

Projet Calcul Sécurité

ZHOU Shuyang

Question 1

Les DES, Data Encryption Standard est un algorithme utilisant une clé de 56 bits et des blocs de 64 bits pour effectuer 16 tours de cryptage sur les données. Les règles pour chaque cycle de transformation sont :

$$L_{r+1} = R_r$$

$$R_{r+1} = L_r \oplus f(R_r, K_{r+1})$$

A chaque tour, la fonction f transfère R_r de 32 bits à 48 bits d'après la permutation de E , XOR avec le sous-clé K_{r+1} , puis on obtient la sortie de 32 bits après 8 Sbox. Après la permutation P au final, on obtient la sortie de f .

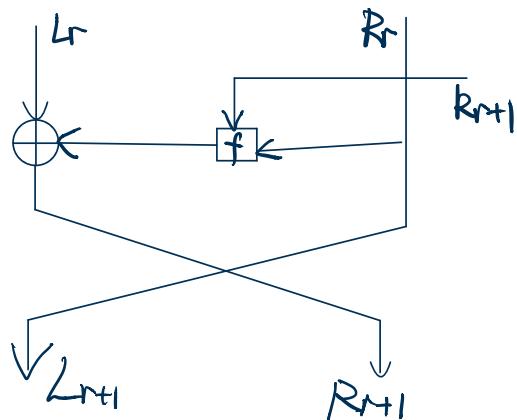


Figure 1 – Tour r

DFA, Differential Fault Analysis, via une erreur d'un bit dans un certain tour de chiffrement, on obtient 2 sorties de texte chiffré différentes C et \tilde{C} du même texte clair. En utilisant la méthode d'analyse différentielle, on peut obtenir la sous-clé K_{16} , puis déchiffrer la clé.

On supposera que l'attaquant est capable d'effectuer une faute sur la valeur de sortie R₁₅ de 15^{ème} tour.

En fait, les attaques DFA visent principalement les cycles ultérieurs de l'algorithme DES, car plus le nombre de cycles où l'erreur se produit est petit, plus la valeur de la différence (entre le message L et le message modifié) est difficile à déterminer et donc moins possible à réussir.

Dans notre cas, cette attaque consiste à modifier 1 bit de R₁₅ pour obtenir une sortie modifiée. On le note R₁₅^f.

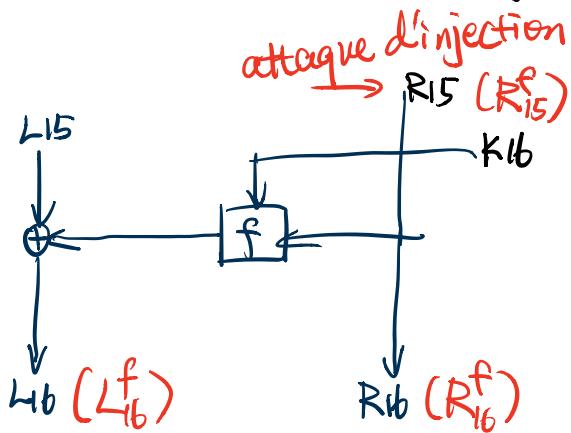


Figure 2 - Injection sur tour 15

Comme la figure 2, on peut étudier les résultats à la sortie :

$$\begin{cases} L_{16} = L_{15} \oplus f(R_{15}, K_{16}) & \textcircled{1} \\ L_{16}^f = L_{15} \oplus f(R_{15}^f, K_{16}) & \textcircled{2} \end{cases}$$

$$\begin{cases} R_{16} = R_{15} \\ R_{16}^f = R_{15}^f \end{cases}$$

$\textcircled{1} \oplus \textcircled{2} =$

$$L_{16} \oplus L_{16}^f = f(R_{15}, K_{16}) \oplus f(R_{15}^f, K_{16}) \quad \textcircled{3}$$

Après on voit comment fonctionne la fonction f dans DES.

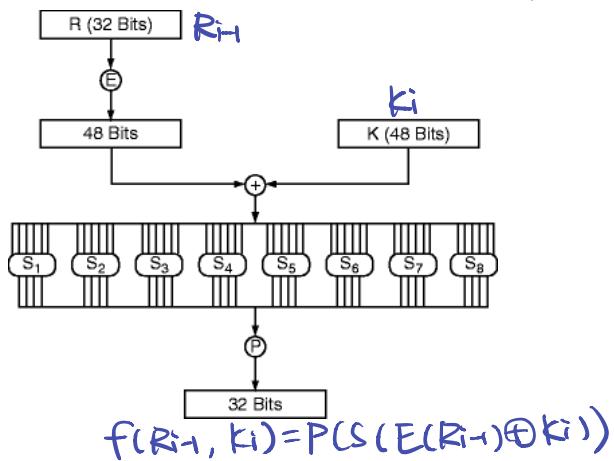


Figure 3 - fonction f dans DES (wikipedia)

On a donc :

$$f(R_{15}, K_{16}) = P(S(E(R_{15}) \oplus K_{16}))$$

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

On récupère le résultat de $E(R_{15}) \oplus K_{16}$ (48 bits). On le coupe en 8 blocs de 6 bits.

Pour chaque SBOX i :

Si $E(R_{15}) \oplus K_{16}$ 6 bits pas

Si $E(R_{15}^f) \oplus K_{16}$ 6 bits pas

On a donc :

$$f(R_{15}, K_{16}) = P(S_i(E(R_{15}) \oplus K_{16} \text{ bits pas}))$$

$$f(R_{15}^f, K_{16}) = P(S_i(E(R_{15}^f) \oplus K_{16} \text{ bits pas}))$$

On applique la permutation P^+ sur ③, on a pour chaque SBOX i :

$$P^+(L_b \oplus L_b^f) = S_i(E(R_{15}) \oplus K_{16}) \oplus S_i(E(R_{15}^f) \oplus K_{16})$$

Comme on a procédé dans le sens inverse, on aura donc 32 bits réparties sur les 8 SBOX, ce qui va donner un découpage de 4 bits par SBOX

$$P^+(L_b \oplus L_b^f)_{\text{4bits pas}} = S_i(E(R_{15}) \oplus K_{16})_{\text{4bits pas}} \oplus S_i(E(R_{15}^f) \oplus K_{16})_{\text{4bits pas}}$$

On fixe les 4 bits de sortie que nous devons obtenir pour chaque SBOX et le utilise comme moyen de vérification. On compare le résultat de 4 bits pour chaque SBOX obtenu par f , avec le résultat qui sert de vérification. Si les 2 résultats sont égaux, on voit la valeur de 6 bits de K_{16} .

Pour trouver cette valeur de K_{16} , on a besoin de faire une recherche exhaustive sur les 6 bits d'entrée de chaque SBOX.

Question 2

Dans cette question, on explique comment retrouver la clé Kib de 48 bits par un programme C.

Au début du programme, j'ai écrit quelques fonctions de conversion hexadécimales et les tables de remplacement qui doivent être utilisées.

```
//passer d'un long en hexa en un tableau de bit
void hexatobinary(int *tabResult, Long hexa, int nbrHexa){
    Long tmp = hexa;
    int entier;
    int compteur = nbrHexa*4-1;
    for(int j=0;j<nbrHexa;j++){
        entier = tmp & 0xF;
        for(int i=0;i<4;i++){
            tabResult[compteur] = entier % 2;
            entier = entier / 2;
            compteur--;
        }
        tmp = tmp >> 4;
    }
}

//transformer un entier en un tableau de bit
void decimaltobinary(int *tabResult, int decimal, int nbrBit){
    int entier = decimal;
    for(int i=nbrBit-1; i>=0; i--){
        tabResult[i] = entier % 2;
        entier = entier / 2;
    }
}
```

Pour retrouver la clé Kib, il faut qu'on analyse pour chaque chiffre faux la position du bit faute.

```
//applique un XOR entre 2 tableau de bit
void xor(int *tabResult, int *premierK, int *deuxiemeK, int nbrBit){
    for(int i=0;i<nbrBit;i++){
        tabResult[i] = premierK[i] ^ deuxiemeK[i];
    }
}

//trouve la position du bit fauter
int bitFauter(int *tabJuste, int *tabFaux) {
    int tabxor[32] = {0};
    xor(tabxor, tabJuste, tabFaux, 32);
    for(int j=0;j<32;j++){
        if(tabxor[j] == 1){
            return j;
        }
    }
    return -1;
}
```

Comme on sait que l'attaque par faute est injecté sur un seul bit des 32 bits de R_{15} . On connaît déjà R_{15} et R_{15}^f . Donc on XOR les 2 pour avoir la position du bit fauté. Après on voit où ce bit est propagé à travers la permutation d' E .

```
//gère toute les permutation y compris l'expansion
void Permutation(int *resultat, int *aPermuter, const int *tablePermutation, int nbrBit){
    for(int i=0;i<nbrBit;i++){
        if(tablePermutation[i] != 0){
            resultat[i] = aPermuter[tablePermutation[i]-1];
        }
    }
}
```

Après on peut trouver quelle SBOX est affectée par quel fauté.

Exemple: une faute sur 1^{er} bit

↓

sortie d' E : 2^{ème}, 48^{ème} bits

↓

SBOX 1 , SBOX 8

```
void Sboxfonc(int *resultat, int *entrer, int numSbox){
    int row = 0;
    int column = 0;
    row = entrer[0]*2+entrer[5];
    int i=0,j=4;
    while(j>0){
        if(entrer[j] != 0){
            column += puissance(2,i);
        }
        i++;
        j--;
    }
    long resultat4bit = sbox[numSbox][row][column];
    hexatobinary(resultat, resultat4bit,1);
}
```

On recherche la clé. On exécute la permutation IP sur le message clair puis nous découpe le message obtenu en 2 parties L₁₆ et R₁₆. On a déjà trouvé la SBOX de

chiffre faux, donc on peut obtenir k_6 avec la recherche exhaustive

```
// s'occupe de retrouver le L16 et le L16 fauter
void obtenirR16L16(long hexa, Message *m){
    m->chiffrerHexa = hexa;
    hexatobinary(m->chiffrerBinaire, hexa, 16);
    Permutation(m->chiffrerBinairePermuter, m->chiffrerBinaire, ip, 64);
    splitTab(m->chiffrerBinairePermuter, m->leftChiffrer, m->rightChiffrer, 32);
}

// On extrer 6 bit une position bien defini au niveau des S-box
void extraire6Bits(Message *m, int position) {
    for(int i=0; i<6; i++) {
        m->sbox6Bits[i] = m->rightChiffrerExp[6*position+i];
    }
}

long rechercheExostive(const long LechiffreJuste, const long *LeschiffreFaux){
    Message juste;
    Message faux;
    long aRetourner = 0;
    int resultat[8][64] = {0};
    obtenirR16L16(LechiffreJuste, &juste);
    int leftPmoini[32] = {0};
    int resultatxorLeft[32] = {0};
    for(int w=0; w<32; w++){
        obtenirR16L16(LeschiffreFaux[w], &faux);
        xor(resultatxorLeft, juste.leftChiffrer, faux.leftChiffrer, 32);
        Permutation(leftPmoini, resultatxorLeft, pmoini, 32);
        int bitfaux = bitFauter(juste.rightChiffrer, faux.rightChiffrer);
        printf("bit faux : %d\n", bitfaux);
        Permutation(juste.rightChiffrerExp, juste.rightChiffrer, e, 48);
        Permutation(faux.rightChiffrerExp, faux.rightChiffrer, e, 48);
        int resBox[4] = {0};
        int resLeftJuste[4] = {0};
        int key[6] = {0};
        for(int i=0; i<48; i++){
            if(e[i] == (bitfaux+1)){
                extraire6Bits(&juste, i/6);
                extraire6Bits(&faux, i/6);
                for(int y=0; y<4; y++){
                    resLeftJuste[y] = leftPmoini[4*(i/6)+y];
                }
                for(int j=0; j<64; j++){
                    decimaltobinary(key, j, 6);
                    xor(juste.sbox6bitsXorer, juste.sbox6Bits, key, 6);
                    decimaltobinary(key, j, 6);
                    xor(faux.sbox6bitsXorer, faux.sbox6Bits, key, 6);
                    SboxFonc(juste.sbox4Bits, juste.sbox6BitsXorer, i/6);
                    SboxFonc(faux.sbox4Bits, faux.sbox6BitsXorer, i/6);
                    xor(resBox, juste.sbox4Bits, faux.sbox4Bits, 4);
                    if(egale(resLeftJuste, resBox, 4)) {
                        resultat[i/6][TabToInt(key, 6)]++;
                    }
                }
            }
        }
    }
    aRetourner = k1GenHexa(resultat);
    return aRetourner;
}
```

Dans la boucle, on calcule la valeur attendue des 4 bits à chaque sortie d'une SBOX avec $P^1(L_{16} \oplus L_{16}^f)$
 Ensuite, on calcule $E(R_{15})$, $E(R_{15}^f)$.
 Après, $E(R_{15}) \oplus E(R_{15}^f)$ et 64 possibilités pour l'entrée

des 8 SBOX.

On récupère les valeurs de 4 bits de chaque SBOX avec un XOR entre les SBOX du chiffre juste et celles des chiffres faux. On compare le résultat avec la valeur de vérification sur 4 bits de chaque SBOX.

22008847

M clair = 4A2767F3BB0E9AB

M chiffré juste = FFE5EA421F43F4DE

K₁₆ = FD76CC4E761D.

K-final = AB16DCAB7A7C57E9

```
63
23
27
12
19
39
24
29
K16: fd76cc4e761d
10101010 00010110 11011100 10101010 01111010 01111100 01010110 11101000
10101011 00010110 11011100 10101011 01111010 01111100 01010111 11101001
Cle : ab16dcab7a7c57e9
```

Question 3

On veut retrouver les 8 bits manquants pour avoir 56 bits ainsi que les 8 bits de parités restants pour celle de 64 bits.

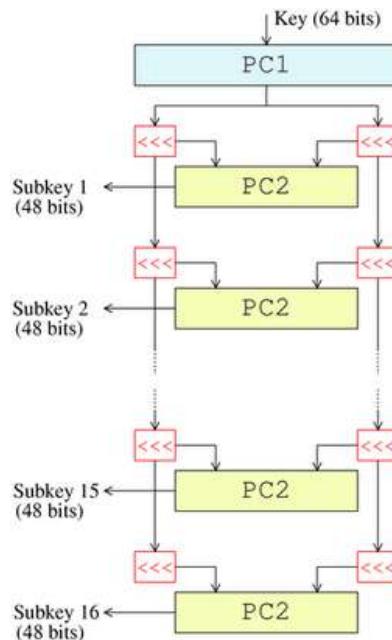


Figure 4 : key-schedule (wikipedia)

$$K_{64} = \text{PC1}^{-1}(\text{PC2}^{-1}(K_{16}))$$

On calcule PC1^{-1} et PC2^{-1} :

```
//permutation 1 inverse dans le key schedule
static const int pc1Moin1[64] = {
    8,16,24,56,52,44,36,0,
    7,15,23,55,51,43,35,0,
    6,14,22,54,50,42,34,0,
    5,13,21,53,49,41,33,0,
    4,12,20,28,48,40,32,0,
    3,11,19,27,47,39,31,0,
    2,10,18,26,46,38,30,0,
    1,9,17,25,45,37,29,0
};
```

```
//permutation 2 inverse dans le key schedule
static const int pc2Moin1[56] = {
    5,24,7,16,6,10,20,
    18,0,12,3,15,23,1,
    9,19,2,0,14,22,11,
    0,13,4,0,17,21,8,
    47,31,27,48,35,41,0,
    46,28,0,39,32,25,44,
    0,37,34,43,29,36,38,
    45,33,26,42,0,30,40
};
```

On implémente l'algorithme du DES pour effectuer une recherche exhaustive sur ces 8 bits. On teste

toutes les 256 positions possibles.

```
//DES : Prend en entrer un message claire et une clef pour donner un message chiffrer
unsigned Long Long fonctionDES(unsigned Long Long claire, unsigned Long Long k64){
    DES d;
    int resultatF[32] = {0};
    int resultatConcat[64] = {0};
    hexatobinary(d.claireBinaire, claire, 16);
    hexatobinary(d.key64Bit, k64, 16);
    Permutation(d.claireBinaireIp, d.claireBinaire, ip, 64);
    splitTab(d.claireBinaireIp, d.left32Bit, d.right32Bit, 32);
    generationSubKey(d.subkey, d.key64Bit);
    for(int i=0; i<16; i++){
        copieTab(d.left32BitPlus1, d.right32Bit, 32);
        f(resultatF, d.right32Bit, d.subKey[i]);
        xor(d.right32BitPlus1, d.left32Bit, resultatF, 32);
        copieTab(d.left32Bit, d.left32BitPlus1, 32);
        copieTab(d.right32Bit, d.right32Bitplus1, 32);
    }
    fusionTab(resultatConcat, d.right32Bit, d.left32Bit, 32);
    Permutation(d.chiffreBinaire, resultatConcat, ipMoin1, 64);
    return TabtoLong(d.chiffreBinaire,64);
}
```

On obtient les 56 bit juste de la clé de chiffrement. Après on trouve les bit de parité. On découpe la clé de 56 bits par blocs de 7 et on calcule la valeur du 8ème bit de chaque bloc en fonction de la parité du nombre de 1 dans les blocs de 7 bits. Si le nombre de 1 est impair on complète par un 0, sinon par 1.

```
unsigned Long Long getK56bit(unsigned long long claire, unsigned long long chiffre, unsigned long long k16){
    Key k;
    initTab(k.key48bit,48);
    initTab(k.keys6bit,56);
    initTab(k.key64bitb,64);
    hexatobinary(k.key48bit,k16,12);
    Permutation(k.key56bit, k.key48bit, pc2Moin1, 56);
    Permutation(k.key64bitb, k.key56bit, pc1Moin1, 64);
    int position8bit[8] = {14,15,19,20,51,54,58,68};
    int i=0;
    while(i<256) {
        decimaltobinary(k.key8bit,i,8);
        for(int j=0;j<8;j++){
            k.key64bitb[position8bit[j]-1] = k.key8bit[j];
        }
        unsigned long long clef = TabtoLong(k.key64bitb,64);
        if(chiffre == fonctionDES(claire,clef)){
            return clef;
        }
        i++;
    }
    return 0;
}

unsigned Long long get64bitParite(unsigned long long claire, unsigned long long chiffre, unsigned Long Long k16){
    int compteur = 0;
    int tabClefB[64] = {0};
    hexatobinary(tabClefB, getK56bit(claire, chiffre, k16), 16);
    printf(" parite:\n");
    for(int i=0; i<64; i++){
        printf("%d ", tabClefB[i]);
        if((i+1)>=8)
            printf("\n");
    }
    printf("\n");
    for(int i=1; i<65; i++) {
        if(((i+8) == 8) {
            if((compteur%2 == 1){
                tabClefB[i-1] = 0;
            }else {
                tabClefB[i-1] = 1;
            }
            compteur = 0;
        }else {
            compteur += tabClefB[i-1];
        }
    }
    for(int i=0; i<64;i++){
        printf("%d ", tabClefB[i]);
        if((i+1)>=8)
            printf("\n");
    }
    printf("\n");
}
```

Résultat dans
page précédente

Question 4

D'après la question 2, on peut obtenir la complexité du 15^{ème}-tour-attaque par le recherche exhaustive on a écrit, est 3×2^{10} .

Pour discuter la faute sur 14^{ème} tour, on définit un symbole^[1]:

On note N le nombre de "1" dans x(binaire)

$$N \{ i | x \gg i \& 1 = 1, 1 \leq i \leq 32 \}.$$

On le note $N\{x\}$. On note n la position du

On note le nombre moyen $\bar{N}\{x\}$. faute.

$\bar{N}(a \oplus b)$ peut mesurer la nombre du différence entre a et b.
(de bits)

Soit 2 ensembles:

$$T_1 = \{2, 3, 6, 7, 10, 11, 14, 15, 18, 19, 22, 23, 26, 27, 30, 31\}$$

$$T_2 = \{1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29, 32\}.$$

Si les fautes sont sur R_{14} ou L_{13} , soit $L_{14} \oplus L_{14}^f \neq 0$,
 $R_{14} \oplus R_{14}^f = 2^n$ ($1 \leq n \leq 32$), on voudrait calculer $\bar{N}(L_{16} \oplus L_{16}^f)$
et $\bar{N}(L_{16} \oplus L_{16}^f)$.

D'après recherche, on a la tableau qui décrit la relation entre la position du faute et le SBOX avec son entrée modifiée.

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

Tableau 1

$(E(R_{14} \oplus R_{14}^f))$

On peut trouver que si n est aléatoire, après la permutation d' E , au moulte 1.5 Sbox change, son

l'entrée. Si $n \in T_1$, $E(R_{14}^f \oplus R_{14}^f)$ change au moyenne 1 SBOX.

D'après recherche^[2], on trouve tableau 2, qui signifie le nombre de bit "1" SBOX a changé pour l'entrée.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x1	0	0	0	6	0	2	4	4	0	10	12	4	10	6	2	4
0x2	0	0	0	8	0	4	4	4	0	6	8	6	12	6	4	2
0x4	0	0	0	6	0	10	10	6	0	4	6	4	2	8	6	2
0x8	0	0	0	12	0	8	8	4	0	6	2	8	8	2	2	4
0x10	0	0	0	0	0	0	2	14	0	6	6	12	4	6	8	6
0x20	0	0	0	10	0	12	8	2	0	6	4	4	4	2	0	12

tableau 2 [2]

On peut voir que le nombre moyen de "1" qui change pendant la sortie de SBOX 1 est 2.48.

Dans la même façons on calcule tous les 8 SBOX.

Sbox	1	2	3	4	5	6	7	8
N	2.48	2.53	2.65	2.46	2.53	2.60	2.63	2.50

D'après la transformation de f dans DES, le faute va modifier en moyenne $1.5 \times 2.55 = 3.83$ bits de changement. C-à-d, $\bar{N}(R_{15} \oplus R_{15}^f) = 3.83$, $\bar{N}(L_{16} \oplus L_{16}^f) = 3.83$. Si $n \in T_1$, $\bar{N}(R_{15} \oplus R_{15}^f) = 2.55$, $\bar{N}(L_{16} \oplus L_{16}^f) = 2.55$.

Ci-dessus est la situation pour les fautes sur $R_{14}(L_{13})$. Maintenant on calcule faute sur $R_{13}(L_{12})$, et la position du faute n aléatoire, on veut calculer $\bar{N}(L_{16} \oplus L_{16}^f)$.

Soit $N(L_r \oplus L_r^f) = n_1$ ($1 \leq n_1 \leq 32$), $N(R_r \oplus R_r^f) = n_2$ soit les bits "1" sont uniforme, $\bar{N}(E(R_r) \oplus E(R_r^f)) = 1.5n_2$. Dans ce cas, le nombre ^{moyenne} du SBOX qui change pour l'entrée est:

$$\begin{cases} 1.5n_2, \text{ si } n_2 < \frac{16}{3} \\ 8, n_2 \geq \frac{16}{3}. \end{cases}$$

$$\bar{N}\{S(E(R_r) \oplus E(R_r^f))\} = \begin{cases} 2.55 \times 1.5 \times n_2, n_2 < \frac{16}{3} \\ 2.55(1.5n_2 - 8) + 2(16 - 1.5n_2), \frac{16}{3} \leq n_2 < \frac{32}{3} \\ 16, n_2 \geq \frac{32}{3} \end{cases}$$

$$= \bar{N}\{f(R_r, k_{r+1}) \oplus f(R_r^f, k_{r+1})\}.$$

Note $\bar{N}_{r,f}$.

Et puis $R_{r+1} = L_r \oplus f(R_r, k_{r+1})$, on a donc :

$$\bar{N}\{R_{r+1} \oplus R_{r+1}^f\} = n_1 + \bar{N}_{r,f} - \frac{n_1 \cdot \bar{N}_{r,f}}{16},$$

$$\text{Et } \bar{N}\{L_{r+1} \oplus L_{r+1}^f\} = n_2$$

Si la faute est sur 12^{ème} tour on avant.

$$\bar{N}\{L_{14} \oplus L_{14}^f\} = 3.83.$$

$$\bar{N}\{R_{14} \oplus R_{14}^f\} = 14.73.$$

$$\bar{N}\{S(E(R_{15}) \oplus E(R_{15}^f))\} = 16.$$

$$\Rightarrow \bar{N}\{L_{16} \oplus L_{16}^f\} = 16$$

Donc on a montré :

La position du faute: R_{15}/L_{14} , R_{14}/L_{13} , R_{13}/L_{12} , Avant

$\bar{N}\{L_{16} \oplus L_{16}^f\}$	1	3.83	14.73	16
$\bar{N}\{R_{16} \oplus R_{16}^f\}$	3.83	14.73	16	16

Comme on a mentionné avant, $\bar{N}\{L_{16} \oplus L_{16}^f\}$ peut mesurer les nombres de bits qui sont changé à cause du faute. Lorsque ce nombre est petit,

nous pouvons réduire la portée et exécuter moins de recherche exhaustive.

On estime la complexité par le nombre de recherches exhaustive :

Pour le 14ème tour, $O(DFA_{14}) \approx 2^{20}$

$O(DFA_{13}) \approx 2^{40}$

$O(DFA_{12}) \approx 2^{80}$

Question 5

Contre-mesure sur l'attaque par fautes contre DES :

(1) Détection ou correction par redondance matérielle.

Cette méthode est de réaliser la même opération sur plusieurs copies d'un même bloc de calcul et d'en comparer les résultats.

La duplication simple avec comparaison est basée sur l'utilisation de 2 copies en parallèle du même bloc, suivies par la comparaison des 2 résultats. Les ressources de la carte à puce qui effectue le calcul seront réparties sur 2 calculs, et le temps de calcul va donc être au pire des cas multiplié par 2.

(2) Détection ou correction par redondance d'information
Cette technique permet, par l'utilisation de codes

déTECTEURS ou CORRECTEURS d'erreur, d'obtenir une certaine protection sans nécessiter d'exécution complète supplémentaire.

3) La redondance spatiale ou temporelle

L'idée est de calculer au moins 2 fois la valeur, c-à-d la ré-exécution d'un même calcul sur le même bloc matériel et la comparaison des différents résultats obtenus. La redondance temporelle simple est basée sur la double exécution d'un calcul sur un même bloc de calcul. La redondance temporelle multiple se base sur l'exécution multiple de la même opération sur la même unité de calcul.

Référence :

[1] WANG X, ZHANG T. Judgemental method of differential fault position in DFA attack against DES. Computer Engineering and Applications, 2011, 47 (19) : 75-77.

[2] Kim C H, Quisquater J J. New differential fault analysis on AES key schedule: Two faults are enough[C]//Grimaud G, Standaert F X. LNCS 5189: CARDIS 2008. Heidelberg: Springer, 2008: 48-60

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//utile dans la recherche de K16
typedef struct message {
    unsigned long long chiffrerHexa;
    int chiffrerBinaire[64];
    int chiffrerBinairePermuter[64];
    int leftChiffrer[32];
    int rightChiffrer[32];
    int rightChiffrerExp[48];
    int sbox6Bits[6];
    int sbox6BitsXorer[6];
    int sbox4Bits[4];
} Message;

//utilise dans les recherche des 16 sous cles
typedef struct clef {
    int key48bit[48];
    int key56bit[56];
    int key64bitb[64];
    int key8bit[8];
} Key;

//utilise dans le DES, elle regroupe toute les variables necessaire au bon deroulement du DES
typedef struct des {
    int claireBinaire[64];
    int key64Bit[64];
    int claireBinaireIp[64];
    int right32Bit[32];
    int left32Bit[32];
    int right32BitPlus1[32];
    int left32BitPlus1[32];
    int right48Bit[48];
    int subKey[16][48];
    int chiffrerBinaire[64];
} DES;

//Initial permutation
static const int ip[64] = {
    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
}

```



```

        {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}
    },
    {
        {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}
    },
    {
        {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}
    },
    {
        {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}
    },
    {
        {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}
    },
    {
        {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}
    },
    {
        {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}
    },
    {
        {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}
    }
}
};
```

```

//permutation 1 dans le key schedul
static const int pc1[56] = {
    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
};

//permutation 1 inverse dans le key schedul
static const int pc1Moin1[64] = {
    8,16,24,56,52,44,36,0,
    7,15,23,55,51,43,35,0,
    6,14,22,54,50,42,34,0,
    5,13,21,53,49,41,33,0,
    4,12,20,28,48,40,32,0,
    3,11,19,27,47,39,31,0,
    2,10,18,26,46,38,30,0,
    1,9,17,25,45,37,29,0
};

//permutation 2 dans le key schedul qui fait passer la clef de 56 a 48 bits
static const int pc2[48] = {
    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
};

//permutation 2 inverse dans le key schedul
static const int pc2Moin1[56] = {
    5,24,7,16,6,10,20,
    18,0,12,3,15,23,1,
    9,19,2,0,14,22,11,
    0,13,4,0,17,21,8,
    47,31,27,48,35,41,0,
    46,28,0,39,32,25,44,
    0,37,34,43,29,36,38,
    45,33,26,42,0,30,40
};

```

```

static const unsigned long long messageClaire = 0x04A2767F3BB0E9AB;

static const unsigned long long chiffrerJuste = 0xFFE5EA421F43F4DE;

static const unsigned long long chiffrerFaux[32] = {
    0xFDE1EA461F42F4DA,
    0xFFFF7EA021F43F4DE,
    0xFFFF5E8461F43F4DE,
    0xFEAE5EA001F42F4DE,
    0xFEAE5EA460D42F4DE,
    0xFFA5EE421F41F4DE,
    0xFEAE5EE420F43F6DE,
    0xFEAE5EE435F57F4DC,
    0xF6A5EA434F53F4DE,
    0xFFEDEA435F57F4DE,
    0FFE5E2435F43F4DE,
    0xBFE5FA4A1F17F4DF,
    0FFE5EA425713F4DF,
    0FFE5FA421F0BF4DF,
    0BFE5EA421F43FCDF,
    0BFE5FA421B03F496,
    0DFE5FA421B03F49E,
    0FFC5EA421B43F59E,
    0FE5CA421B43E49E,
    0EBE5EB621B43E59E,
    0FBE5AB423B43E49E,
    0EBE5EA421F63F4DE,
    0FBE5EB421F43D4DE,
    0EFE5AA521F43B4FE,
    06FE5AB521E43B0DE,
    0FF65EA521E43F0DE,
    0FFE56A521F43B4DE,
    0FFE4EAD21F43F0DA,
    0FFE0EA429E43B4DE,
    0FFE0EA421FC3F4CE,
    0FFE4EA421F4374DA,
    0FFF0EA421F42F44A
};

//passer d'un unsigned long long en hexa en un tableau de bit
void hexatobinary(int *tabResult, unsigned long long hexa, int nbrHexa) {
    unsigned long long tmp = hexa;
    int entier;
    int compteur = nbrHexa*4-1;
    for(int j=0;j<nbrHexa;j++) {
        entier = tmp & 0xF;
        for(int i=0;i<4;i++) {
            tabResult[compteur] = entier % 2;
            entier = entier / 2;
        }
    }
}

```

```

        compteur--;
    }
    tmp = tmp >> 4;
}
}

//transformer un entier en un tableau de bit
void decimaltobinary(int *tabResult, int decimal, int nbrBit){
    int entier = decimal;
    for(int i=nbrBit-1; i>=0; i--) {
        tabResult[i] = entier % 2;
        entier = entier /2;
    }
}

//élevé à la puissance
unsigned long long puissance(int a,int b) {
    if(b == 0) {
        return 1;
    } else {
        return a*puissance(a,b-1);
    }
}

//transformer un tableau de bit en un entier
int TabtoInt(int *tab,int nbrBit) {
    int nombre=0;
    int i=0,j=nbrBit-1;
    while(j>=0) {
        if(tab[j] != 0) {
            nombre += puissance(2,i);
        }
        i++;
        j--;
    }
    return nombre;
}

//transformer un tableau de bit en un hexadecimale
unsigned long long TabtoLong(int *tab,int nbrBit) {
    unsigned long long nombre=0;
    int i=0,j=nbrBit-1;
    while(j>=0) {
        if(tab[j] != 0) {
            nombre += puissance(2,i);
        }
        i++;
        j--;
    }
}
```

```

    }

    return nombre;
}

//gère toute les permutation y compris l'expansion
void Permutation(int *resultat,int *aPermuter,const int *tablePermutation, int nbrBit) {
    for(int i=0;i<nbrBit;i++) {
        if(tablePermutation[i] != 0) {
            resultat[i] = aPermuter[tablePermutation[i]-1];
        }
    }
}

//divise un tableau de bit en 2 tableau de même longueur
void splitTab(int *completTab,int *leftTab,int *rightTab, int nbrBit) {
    for(int i=0;i<nbrBit;i++) {
        leftTab[i] = completTab[i];
        rightTab[i] = completTab[i+nbrBit];
    }
}

//applique un XOR entre 2 tableau de bit
void xor(int *tabResult, int *premierK, int *deuxiemeK, int nbrBit) {
    for(int i=0;i<nbrBit;i++) {
        tabResult[i] = premierK[i] ^ deuxiemeK[i];
    }
}

//trouve la position du bit fauter
int bitFauter(int *tabJuste, int *tabFaux) {
    int tabxor[33] = {0};
    xor(tabxor, tabJuste, tabFaux, 32);
    for(int j=0;j<32;j++) {
        if(tabxor[j] == 1) {
            return j;
        }
    }
    return -1;
}

//la boîte de substitution qui transforme 6 bit en 4 bit selon la S-box choisie
void Sboxfond(int *resultat, int *entrer, int numSbox) {
    int row = 0;
    int column = 0;
    row = entrer[0]*2+entrer[5];
    int i=0,j=4;
    while(j>0) {
        if(entrer[j] != 0) {
            column += puissance(2,i);
        }
        i++;
        j--;
    }
    resultat[0] = column;
}

```

```

        }
        i++;
        j--;
    }

    unsigned long long resultat4bit = sbox[numSbox][row][column];
    hexatobinary(resultat, resultat4bit,1);
}

//s'occupe de retrouver le L16 et le L16 fauter
void obtenirR16L16(unsigned long long hexa,Message *m) {
    m->chiffrerHexa = hexa;
    hexatobinary(m->chiffrerBinaire,hexa,16);
    Permutation(m->chiffrerBinairePermuter, m->chiffrerBinaire, ip, 64);
    splitTab(m->chiffrerBinairePermuter,m->leftChiffrer,m->rightChiffrer, 32);

}

//On extrer 6 bit une position bien defini au niveau des S-box
void extraire6Bits(Message *m, int position) {
    for(int i=0; i<6; i++) {
        m->sbox6Bits[i] = m->rightChiffrerExp[6*position+i];
    }
}

//Compare deux tableau de meme taille
int egale(int *tab1, int *tab2, int nbrBit){
    for(int i=0; i<nbrBit; i++){
        if(tab1[i] != tab2[i])
            return 0;
    }
    return 1;
}

//traite les resultat de la recherche exhaustive pour trouver le bon K16
unsigned long long k16enHexa(int tabK16[8][64]){
    unsigned long long resultat = 0;
    int tab[8] = {0};
    int tabclef[6] = {0};
    int tabresult[64] = {0};
    for(int i=0; i<8; i++){
        for(int j=0;j<64;j++) {
            if(tabK16[i][j] == 6)
                tab[i] = j;
        }
        printf("%d\n", tab[i]);
        decimaltobinary(tabclef, tab[i], 6);
        for(int j=0; j<6;j++){
            tabresult[i*6+j] = tabclef[j];
        }
    }
}

```

```

    }

    resultat = TabtoLong(tabresultat, 48);

    return resultat;
}

//Fait la recherche exhaustive pour trouver la clef K15
unsigned long long rechercheExostive(const unsigned long long LechiffreJuste, const unsigned
long long *LeschiffreFaux) {
    Message juste;
    Message faux;
    unsigned long long aRetourner = 0;
    int resultat[8][64] = {0};
    obtenirR16L16(LechiffreJuste, &juste);
    int leftPmoin1[32] = {0};
    int resultatxorLeft[32] = {0};
    for(int w=0;w<32;w++) {
        obtenirR16L16(LeschiffreFaux[w], &faux);
        xor(resultatxorLeft, juste.leftChiffre, faux.leftChiffre, 32);
        Permutation(leftPmoin1, resultatxorLeft, pMoin1, 32);
        int bitfaux = bitFauter(juste.rightChiffre, faux.rightChiffre);
        //printf("bit faux : %d\n", bitfaux);
        Permutation(juste.rightChiffreExp, juste.rightChiffre, e, 48);
        Permutation(faux.rightChiffreExp, faux.rightChiffre, e, 48);
        int resSbox[4] = {0};
        int resLeftJuste[4] = {0};
        int key[6] = {0};
        for(int i=0; i<48; i++) {
            if(e[i] == (bitfaux+1)) {
                extraire6Bits(&juste, i/6);
                extraire6Bits(&faux, i/6);
                for(int y=0; y<4; y++) {
                    resLeftJuste[y] = leftPmoin1[4*(i/6)+y];
                }
                for(int j=0; j<64; j++) {
                    decimaltobinary(key, j, 6);
                    xor(juste.sbox6BitsXorer, juste.sbox6Bits, key, 6);
                    decimaltobinary(key, j, 6);
                    xor(faux.sbox6BitsXorer, faux.sbox6Bits, key, 6);
                    Sboxfonc(juste.sbox4Bits, juste.sbox6BitsXorer, i/6);
                    Sboxfonc(faux.sbox4Bits, faux.sbox6BitsXorer, i/6);
                    xor(resSbox, juste.sbox4Bits, faux.sbox4Bits, 4);
                    if(egale(resLeftJuste, resSbox, 4)) {
                        resultat[i/6][TabtoInt(key, 6)]++;
                    }
                }
            }
        }
    }
    aRetourner = k16enHexa(resultat);
}

```

```

    return aRetourner;
}

//initialise le tableau de bit a 0
void initTab(int *tab, int nbrBit) {
    for(int i=0;i<nbrBit;i++) {
        tab[i] = 0;
    }
}

//S'occupe de shifter les bit de nbrShift fois vers la gauche
void shiftGauche(int *resultat, int *tabAshifter, int nbrShift, int nbrBit) {
    for(int i=-nbrShift; i<nbrBit-nbrShift; i++) {
        if(i<0) {
            resultat[i+nbrBit] = tabAshifter[i+nbrShift];
        } else {
            resultat[i] = tabAshifter[i+nbrShift];
        }
    }
}

//On fusionne deux tableau de meme dimension
void fusionTab(int *resultat, int *leftTab, int *rightTab, int nbrBit) {
    for(int i=0; i<nbrBit; i++) {
        resultat[i] = leftTab[i];
        resultat[i+nbrBit] = rightTab[i];
    }
}

//Genere les 16 clef de K0 a K15 dans le DES
void generationSubKey(int Les16SubKey[][48], int *key64Bit) {
    int v[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
    int key56bit[56] = {0};
    int key48bit[48] = {0};
    int tabSplit[2][28] = {0};
    int tabSplitRes[2][28] = {0};
    Permutation(key56bit, key64Bit, pc1, 56);
    for(int i=0; i<16; i++) {
        splitTab(key56bit, tabSplit[0], tabSplit[1], 28);
        shiftGauche(tabSplitRes[0], tabSplit[0], v[i], 28);
        shiftGauche(tabSplitRes[1], tabSplit[1], v[i], 28);
        fusionTab(key56bit, tabSplitRes[0], tabSplitRes[1], 28);
        Permutation(Les16SubKey[i], key56bit, pc2, 48);
    }
}

//Fonction interne F du DES
void f(int *resultat, int *Ri, int *Ki) {
    int right48b[48] = {0};

```

```

int resultatXor[48] = {0};
int entrerSbox[6] = {0};
int sortiSbox[4] = {0};
int sorti32bit[32] = {0};
Permutation(right48b,Ri,e,48);
xor(resultatXor,right48b,Ki,48);
for(int j=0;j<8;j++) {
    for(int i=0; i<6; i++) {
        entrerSbox[i] = resultatXor[6*j+i];
    }
    Sboxfond(sortiSbox, entrerSbox, j);
    for(int i=0;i<4;i++) {
        sorti32bit[j*4+i] = sortiSbox[i];
    }
}
Permutation(resultat, sorti32bit, p, 32);
}

//Sert à copier un tableau dans un autre de même dimension
void copieTab(int *resultat, int *aCopier, int nbrBit){
    for(int i=0; i<nbrBit; i++){
        resultat[i] = aCopier[i];
    }
}

//DES : Prend en entrer un message claire et une clef pour donner un message chiffrer
unsigned long long fonctionDES(unsigned long long claire, unsigned long long k64) {
    DES d;
    int resultatF[32] = {0};
    int resultatConcat[64] = {0};
    hexatobinary(d.claireBinaire, claire, 16);
    hexatobinary(d.key64Bit, k64, 16);
    Permutation(d.claireBinaireIp, d.claireBinaire, ip, 64);
    splitTab(d.claireBinaireIp, d.left32Bit, d.right32Bit, 32);
    generationSubKey(d.subKey, d.key64Bit);
    for(int i=0; i<16; i++) {
        copieTab(d.left32BitPlus1, d.right32Bit, 32);
        f(resultatF, d.right32Bit, d.subKey[i]);
        xor(d.right32BitPlus1, d.left32Bit, resultatF, 32);
        copieTab(d.left32Bit, d.left32BitPlus1, 32);
        copieTab(d.right32Bit, d.right32BitPlus1, 32);
    }
    fusionTab(resultatConcat, d.right32Bit, d.left32Bit, 32);
    Permutation(d.chiffreBinaire, resultatConcat, ipMoin1, 64);
    return TabtoLong(d.chiffreBinaire, 64);
}

//Sert à obtenir les 56 bit juste de la clef de chiffrement
//(il restera que les bit de parité à trouver)

```

```

unsigned long long getK56bit(unsigned long long claire, unsigned long long chiffrer, unsigned
long long k16) {
    Key k;
    initTab(k.key48bit, 48);
    initTab(k.key56bit, 56);
    initTab(k.key64bitb, 64);
    hexatobinary(k.key48bit, k16, 12);
    Permutation(k.key56bit, k.key48bit, pc2Moin1, 56);
    Permutation(k.key64bitb, k.key56bit, pclMoin1, 64);
    int position8bit[8] = {14, 15, 19, 20, 51, 54, 58, 60};
    int i=0;
    while(i<256) {
        decimaltobinary(k.key8bit, i, 8);
        for(int j=0;j<8;j++) {
            k.key64bitb[position8bit[j]-1] = k.key8bit[j];
        }
        unsigned long long clef = TabtoLong(k.key64bitb, 64);
        if(chiffrer == fonctionDES(claire,clef)) {
            return clef;
        }
        i++;
    }
    return 0;
}

```

```

//Renvoi la clef de chiffrement complete de 64 bits
unsigned long long getK64bitParite(unsigned long long claire, unsigned long long chiffrer,
unsigned long long k16) {
    int compteur = 0;
    int tabClefB[64] = {0};
    hexatobinary(tabClefB, getK56bit(claire, chiffrer, k16), 16);
    for(int i=0; i<64;i++) {
        printf("%d", tabClefB[i]);
        if((i+1)%8 == 0)
            printf(" ");
    }
    printf("\n");
    for(int i=1; i<65; i++) {
        if((i%8) == 0) {
            if(compteur%2 == 1) {
                tabClefB[i-1] = 0;
            }else {
                tabClefB[i-1] = 1;
            }
            compteur = 0;
        }else {
            compteur += tabClefB[i-1];
        }
    }
}
```

```
for(int i=0; i<64;i++) {
    printf("%d", tabClefB[i]);
    if((i+1)%8 == 0)
        printf(" ");
}
printf("\n");
return TabtoLong(tabClefB, 64);
}

int main() {

    unsigned long long K_16 = rechercheExostive(chiffrerJuste,chiffrerFaux);
    printf("K16: %llx\n", K_16);
    printf("Cle : %llx\n", getK64bitParite(messageClaire, chiffrerJuste, K_16));
}

return 0;
}
```