Rapport Projet Calcul Sécurisé

Robin Joran

25 Mars 2019

Attaque Par Fautes Sur Le DES

Sommaire

1 Question 1 : Attaque Par Faute Sur Le DES

Le **D**ata **E**ncryption **S**tandard est un algorithme de chiffrement symétrique développé par IBM en 1977 et permettant de chiffrer des données.Nous allons ,dans ce document, étudier comment se déroule une attaque dite "Par Fautes" sur cet algorithme.

Une attaque par faute sur le DES consiste à provoquer une faute volontaire sur l'algorithme afin d'en extraire des informations secrètes(Comme par exemple,des informations sur la clé K). Sachant qu'une attaque par recherche exhaustive sur la clé du DES a une complexité de 2^{56}, le but de cette attaque est donc d'être plus efficace.

Nous allons donc supposer qu'un attaquant est capable d'effectuer une attaque par faute sur la sortie R_{15} du $15^{\text{ème}}$ tour de Feistel du DES et en étudier les conséquences.

Tout d'abord,l'objectif de cette attaque consiste à modifier 1 bit sur les 32 bits qui composent R15 afin d'obtenir une sortie faussée que nous noterons R_{15}^f.La figure ci-dessous permet de voir les répercussions de l'injection d'une telle faute à la sortie du $15^{\text{ème}}$ tour du DES :

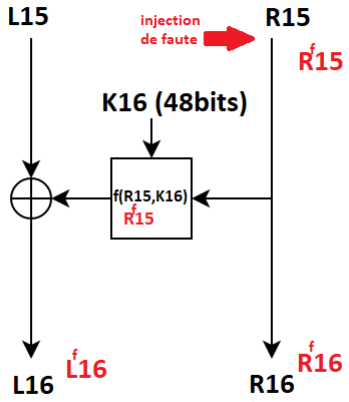

FIGURE 1.1 – Injection d'une Faute sur la sortie R_{15}

Par propagation, on obtient donc un L_{16} et un R_{16} fauté que nous noterons L_{16}^f et R_{16}^f.Nous allons donc étudier les résultats obtenus à la sortie du dernier tour à l'aide des formules connues suivantes :

$$L_{16} = L_{15} \oplus f(R_{15}, K_{16}) \text{ et } R_{16} = R_{15}$$

$$L_{16}^f = L_{15} \oplus f(R_{15}^f, K_{16}) \text{ et } R_{16}^f = R_{15}^f$$

On peut donc voir que K_{16} est commun à L_{16} et L_{16}^f et qu'il suffit de faire un XOR entre les deux pour supprimer L_{15}, ce qui donne :

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

On peut donc regarder comment fonctionne la fonction f du DES afin d'approfondir notre recherche.Voici le détail de son fonctionnement ci-dessous :

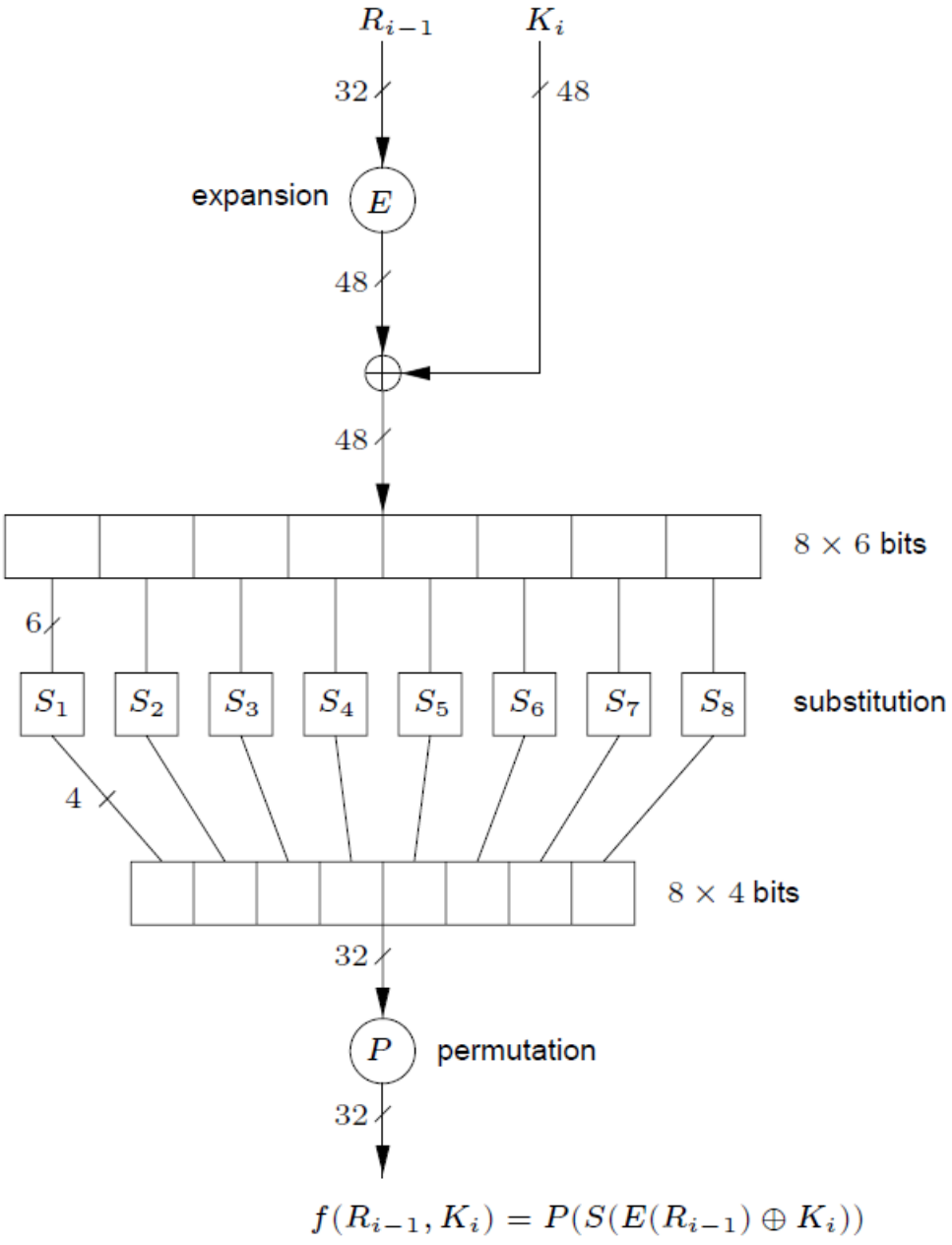

FIGURE 1.2 – Fonction f

On a donc le résultat suivant d'après la figure 1.2 :

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

Ce résultat est obtenu après avoir appliquée l'expansion E à R_{15} qui va permettre de transformer R_{15} de 32bits en 48bits puis on va appliquer le XOR entre E(R_{15}) et la clé K16 qui a aussi une taille de 48bits.
Ensuite, on récupère le résultat de $(E(R_{15}) \oplus K_{16})$ qui fait 48bits et on va le découper en 8blocs de 6bits car il y a 8 SBOXs et chacune d'elle prend 6bits en entrée. On obtient donc les formules suivantes pour l'entrée de chaque SBOX i ($j \to k$ représente des pas de 6bits) :

$$S_i(E(R_{15}) \oplus K_{16 \text{ bits } j \to k}) \text{ et } S_i(E(R_{15}^f) \oplus K_{16 \text{ bits } j \to k})$$

Chacune de ces SBOX va renvoyer un résultat de 4bits que l'on va concaténer ensemble pour obtenir un résultat sur 32 bits que l'on va passer dans la Permutation P,qui donne aussi un résultat sur 32bits, pour enfin obtenir ceci :

$$\begin{aligned} f(R_{15}, K_{16}) &= P(S_i(E(R_{15}) \oplus K_{16 \text{ bits } j \to k})) \text{ et} \\ f(R_{15}^f, K_{16}) &= P(S_i(E(R_{15}^f) \oplus K_{16 \text{ bits } j \to k})) \end{aligned}$$

On a donc obtenue les sorties de la fonction f qui vont être stockées dans L_{16} et L_{16}^f.Comme on sait maintenant comment marche la fonction f, essayons de la parcourir à l'envers en reprenant l'équation :

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

Donc si on applique la permutation P^{-1} sur $L_{16} \oplus L_{16}^f$ d'après ce qu'on a trouvé précédemment,on a pour chaque SBOX i :

$$P^{-1}(L_{16} \oplus L_{16}^f) = S_i(E(R_{15}) \oplus K_{16 \text{ bits } j \to k}) \oplus S_i(E(R_{15}^f) \oplus K_{16 \text{ bits } j \to k})$$

Comme on a procédé dans le sens inverse,on aura donc 32bits réparties sur les 8 SBOX,ce qui va donner un découpage de 4bits par SBOX c'est à dire ($a \to b$ représente des pas de 4bits) :

$$P^{-1}(L_{16} \oplus L_{16}^f)_{\text{ bits } a \to b} = S_i\ (E(R_{15}) \oplus K_{16})_{\text{ bits } a \to b} \oplus S_i\ (E(R_{15}^f) \oplus K_{16})_{\text{ bits } a \to b}$$

Ceci va nous permettre de fixer les 4bits de sortie que nous devons obtenir pour chaque SBOX et sera donc utilisé comme moyen de verification.

On aura juste à comparer le résultat de 4bits pour chaque SBOX obtenu en exécutant la fonction f ,avec le résultat qui sert de vérification.Si les 2 résultats sont égaux alors on garde la valeur de 6bits de la clé K16 qu'on a utilisé. .

Pour trouver cette valeur de la clé K16,on a besoin de faire une recherche exhaustive sur les 6bits d'entrée de la clé de chaque SBOX.Ceci est faisable car du coup on a une recherche exhaustive de 2^6 soit 64 possibilités pour une SBOX et donc 8x2^6 pour toutes les SBOX.Nous allons voir en détail comment cela est possible à la question suivante.

2 Question 2 : Application Concrète

2.1 Retrouver K16 48bits

Dans cette partie,nous allons donc expliquer concrètement comment nous avons procédé pour retrouver la clé K16 de 48bits en nous appuyant sur le code en Langage C mis à disposition entièrement dans la partie "Annexes" du Rapport.
Dans la partie précédente,nous avons expliqué comment procéder pour faire une attaque par fautes sur le DES.
La première chose qu'il faut faire, pour retrouver la clé K16 de 48bits, est d'analyser pour chaque chiffré faux la position du bit fauté.C'est ce que nous faisons dans cette partie du code :

```c
void analyse_bit_faute(long msg_EncJuste,long msg_EncFaux[],int num)
{
    long msg_Dec_EncJuste,msg_Dec_EncFaux;//Dechiffré OK par rapport au message crypté OK
    long R16,R16f,R15,R15f;
    long position_bit_faute;
    int tmp_pos =33;

    //~ printf("\n########################## PERMUTATION IP Chiffrés Juste ##########################\n\n");

    msg_Dec_EncJuste = permutation(msg_EncJuste, IP, 64, 64);

    R16 = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);
    R16 = decoupage_msgR(R16,0xFFFFFFFF);

    R15=R16;

        //~ printf("\n########################## PERMUTATION IP Chiffrés Faux ##########################\n\n");

        msg_Dec_EncFaux = permutation(msg_EncFaux[num], IP, 64, 64);

        R16f = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);
        R16f = decoupage_msgR(R16f,0xFFFFFFFF);

        R15f=R16f;

        position_bit_faute = or_exclu(R15,R15f);

        //~ printf("\n########################## XOR ##########################\n\n");

        while(position_bit_faute)
        {
            tmp_pos--;
            position_bit_faute = position_bit_faute >> 1;
        }

        //~ printf("\n########################## POSITION BIT FAUTE ##########################\n\n");

        if(num <9)
        {
            printf("Chiffré faux  %d --> Position bit fauté %d \n",num+1,tmp_pos);
        }
        else
        {
            printf("Chiffré faux %d --> Position bit fauté %d \n",num+1,tmp_pos);
        }
}
```

K16_48.c

Comme on sait que l'attaque par faute est injecté seulement sur un seul bit des 32bits de

R_{15},et qu'on connaît R_{15} et R_{15}^{f},il nous suffit de XOR les 2 ensembles pour avoir la position du bit fauté.On récupère donc la position pour les 32 chiffrés faux et on va ensuite regarder où ce bit est propagé à travers la permutation d'expansion : :

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

FIGURE 2.1 – Expansion E

Ceci va donc nous permettre de cibler exactement quelle SBOX est affectée par quel fauté.On peut voir que si le chiffré faux a une faute sur le 1er bit,celui ci va affecter le 2ème et 48ème bits de la sortie de E et sera donc propager en entrée de la SBOX 1 et la SBOX 8.On en déduit donc que pour une faute donnée,elle peut se répartir au maximum sur 2 SBOX.

On peut donc stocker la position des bits fautés de chaque chiffré faux dans un tableau et celui ci nous indiquera pour chaque SBOX les 6 chiffrés faux correspondants :

```
static int pos_bitfaux[8][6] =
{
    {0,31,30,29,28,27},
    {28,27,26,25,24,23},
    {24,23,22,21,20,19},
    {20,19,18,17,16,15},
    {16,15,14,13,12,11},
    {12,11,10,9,8,7},
    {8,7,6,5,4,3},
    {4,3,2,1,0,31}
};
```

TablesPermu.h

Les 8 lignes représentes les 8 SBOX et les 6 colonnes représentent les différents chiffrés dans l'ordre.Il y a donc 6chiffrés par SBOX.Par exemple ici ,le premier chiffré faux a une faute sur

le bit 32 et sera donc injecté dans la SBOX1 et la SBOX 8.Le second a une faute sur le bit 31 et sera aussi injecté dans l SBOX 1 et la SBOX8 ainsi de suite...

Maintenant que nous avons la position du bit fauté pour chaque chiffré faux ,nous allons donc procéder à la recherche de la clé K16 48bits :

```
/*! \fn long trouverK16_48b(long msg_EncJuste, long msg_EncFaux[]);
*  \brief Fonction qui permet de retrouver la clé K16 de 48 bits
*  \param msg_EncJuste : message crypté juste
*  \param msg_EncFaux : messages cryptés faux
*  \return K16 la clé de 48 bits
*/
long trouverK16_48b(long msg_EncJuste, long msg_EncFaux[])
{
    long K16 = 0;//48bits
    long msg_Dec_EncJuste;//Dechiffré OK par rapport au message crypté OK
    long msg_Dec_EncFaux;//Dechiffré Faux OK par rapport au message faux crypté OK
    long check;//Verification de 32bits
    long entree_SBOX,entree_SBOXf;//Entree des Sbox
    long EXP_R15, EXP_R15f;//Expansion
    long L16,R16,R15,L16f,R16f,R15f;//L et R fautés ou non


    int cle[8][6][64] = {{{0}}};//Tableau des 64 solutions pour les 8 Sbox avec les 6 chiffrés fautés correspondants
    int solPossibles[8][6] = {{0}};//Nombre de solutions possibles des 6 chiffrés faux pour les 8 Sbox

    int nbr_faux;

    int check_4bits;//Sbox de verification
    int SBOX_4bits;//Obtenue avec les fautés
    int ligne,colonne;//SBOX
    int ligne_f,colonne_f;//SBOX fautée
    int NUM_SBOX ;//Numéro SBOX
    int RE_k16;//recherche exaustive K16


    //~ printf("\n########################## Analyse Bit Fauté ##########################\n\n");

    //~ for (int analyse=0;analyse<32;analyse++)
    //~ {
        //~ analyse_bit_faute(msg_EncJuste,msg_EncFaux,analyse);
    //~ }

    //printf("\n### PERMUTATION IP Chiffré Juste ### \n");

    msg_Dec_EncJuste = permutation(msg_EncJuste, IP, 64, 64); //Permutation IP avec le chiffré

    //printf("\n### DECOUPAGE L16 et R16 Juste ### \n");

    L16 = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);//On stocke la partie L16

    R16 = decoupage_msgR(msg_Dec_EncJuste,0xFFFFFFFF);//On stocke la partie R16
    R15 = R16;//On stocke R16 dans R15
```

K16_48.c

Premièrement,nous exécutons la permutation IP sur le message clair puis nous découpons le message obtenu en 2 parties L_{16} et R_{16}.On stocke R_{16} dans R_{15}. Comme on sait quel chiffré faux va dans quelle SBOX,On va pouvoir attaquer avec la recherche exhaustive de K_{16} les différentes SBOX avec les 6chiffrés faux correspondants :

```
        //Recherche exhaustive
        for(NUM_SBOX=0;NUM_SBOX<8;NUM_SBOX++)
        {
            for(nbr_faux=0;nbr_faux<6;nbr_faux++)
            {
                msg_Dec_EncFaux = permutation(msg_EncFaux[pos_bitfaux[NUM_SBOX][nbr_faux]], IP, 64, 64);//
    Permutation IP chiffré FAUX

                L16f = decoupage_msgL(msg_Dec_EncFaux,0xFFFFFFFF);//Stockage L16f
                R16f = decoupage_msgR(msg_Dec_EncFaux,0xFFFFFFFF);//Stockage R16f
                R15f=R16f;//Stockage R15f

                check = permutation(or_exclu(L16,L16f),inverse_P,32,32);//Valeur de Verification pour la
    sortie de SBOX

                EXP_R15=permutation(R15,E,32,48);//Expansion R15
                EXP_R15f=permutation(R15f,E,32,48);//Expansion R15f

                for (RE_k16=0;RE_k16<64;RE_k16++)
                {
                    entree_SBOX = or_exclu(shift_droit(decoupe_6bits(EXP_R15,NUM_SBOX),trouver_SBOX_6bits(
    NUM_SBOX)),RE_k16);//Entrée Sbox 6bits XOR k16
                    entree_SBOXf = or_exclu(shift_droit(decoupe_6bits(EXP_R15f,NUM_SBOX),trouver_SBOX_6bits(
    NUM_SBOX)),RE_k16);//Entrée Sboxf 6bits XOR k16

                    ligne = calcul_lignes_SBOX(entree_SBOX);//Ligne SBOX
                    colonne = calcul_col_SBOX(entree_SBOX);//Colonne SBOX

                    ligne_f = calcul_lignes_SBOX(entree_SBOXf);//Ligne SBOXf
                    colonne_f = calcul_col_SBOX(entree_SBOXf);//colonne SBOXf

                    check_4bits =shift_droit(decoupe_4bits(check,NUM_SBOX),trouver_SBOX_4bits(NUM_SBOX));//
    Valeur de Verification pour la sortie de SBOX 4bits
                    SBOX_4bits = or_exclu(Sbox[NUM_SBOX][ligne][colonne],Sbox[NUM_SBOX][ligne_f][colonne_f])
    ;//Valeur de sortie SBOX 4 bits

                    if(check_4bits == SBOX_4bits)//Comparaison
                    {
                        cle[NUM_SBOX][nbr_faux][solPossibles[NUM_SBOX][nbr_faux]] = RE_k16;//Stockage
    valeurs clés
                        ++solPossibles[NUM_SBOX][nbr_faux];//Incrementation
                    }
                }
            }
```

K16_48.c

Dans la boucle de la Recherche exhaustive, pour chaque chiffré faux,on va calculer la permutation IP puis découper en 2 parties L_{16}^f et R_{16}^f.On stocke R_{16}^f dans R_{15}^f.
Puis on calcule la valeur attendue des 4bits à chaque sortie d'une SBOX avec la permutation $P^{-1}(L_{16} \oplus L_{16}^f)$.
Ensuite, on procède au calcul de l'expansion de R_{15} et de R_{15}^f puis on va appliquer un XOR entre l'expansion et les 64 possibilités de clé K_{16} pour l'entrée des 8 SBOX avec $E(R_{15})$ et $E(R_{15}^f)$.
On récupère les valeurs de 4bits de chaque SBOX avec un XOR entre les SBOX du chiffré juste et celles des chiffrés faux puis on compare le résultat avec la valeur de vérification sur 4bits de chaque SBOX.
Si c'est équivalent alors on stocke la possible solution de K16 48bits dans un tableau. Cepen-

dant,on a pas encore trouvé la valeur de la clé car pour chaque Sbox testé avec les 6 fautés correspondant,on aura plusieurs solutions de clé K16 pou chaque fauté.
Il suffit juste de récupérer la solution commune des 6 fautés d'une SBOX pour enfin parvenir à avoir le bout de clé correspondant.
Pour finir,on concatène les valeurs de clé obtenues par chaque SBOX pour trouver la clé K16 48bits,la démarche est indiqué dans cette fonction :

```c
/*! \fn long recupSolution(long cle[][6][64],solPossibles[][6],int NUM_SBOX,long K16)
 *  \brief Fonction qui permet de récupérer la valeur de K16_46bits
 *  \param cle[][6][64] : Tableau des 64 solutions pour les 8 Sbox avec les 6 chiffrés fautés correspondants
 *  \param solPossibles[][6] : Nombre de solutions possibles des 6 fautes pour les 8 Sbox
 *  \param NUM_SBOX : numéro SBOX
 *  \param K16 : K16
 *  \return K16 48bits
 */
long recupSolution(int cle[][6][64],int solPossibles[][6],int NUM_SBOX,long K16)
{
    int test = 0;
    int final_solution;

    long elu = (long) cle[NUM_SBOX][0][test];

        for (int nbr_Faux = 1; nbr_Faux < 6; nbr_Faux++)
        {
            for (final_solution = 0;final_solution<solPossibles[NUM_SBOX][nbr_Faux];final_solution++)
            {
                if (elu == cle[NUM_SBOX][nbr_Faux][final_solution])
                {
                    break;
                }
            }

            if(final_solution == solPossibles[NUM_SBOX][nbr_Faux])
            {
                if(test+1 >= solPossibles[NUM_SBOX][0])
                {
                    printf("\nProblème Sbox numéro %d,chiffré faux %d\n",NUM_SBOX,nbr_Faux);
                    break;
                }
                nbr_Faux = 1;
                ++test;
                elu = (long) cle[NUM_SBOX][0][test];
            }
        }
        printf("\nClé K de 6bits Choisie dans la SBOX %d : %lX donc on concat avec l'ancien K\n",NUM_SBOX+1,
    elu);
        K16 = shift_gauche(K16,6);
        K16 = concat(K16,elu);
        printf("\nK16 courant = %lX\n",K16);

        return K16;
}
```

K16_48.c

```c
        K16=recupSolution(cle,solPossibles,NUM_SBOX,K16);//Récupération de la solution commune des 6
    chiffrés fautés par SBOX puis Concatenation de la clé K16
```

K16_48.c

On obtient donc un recherche exhaustive de complxité $8 * 6 * 2^6$,ce qui donne $3 * 2^{10}$

2.2 48 bits de ma clé K16 obtenus

Voici ma clé K de 48bits : D8 92 78 03 7D 46

```
###################### RECHERCHE K16 48BITS ######################

Clé K de 6bits Choisie dans la SBOX 1 : 36 donc on concat avec l'ancien K

K16 courant = 36

Clé K de 6bits Choisie dans la SBOX 2 : 9 donc on concat avec l'ancien K

K16 courant = D89

Clé K de 6bits Choisie dans la SBOX 3 : 9 donc on concat avec l'ancien K

K16 courant = 36249

Clé K de 6bits Choisie dans la SBOX 4 : 38 donc on concat avec l'ancien K

K16 courant = D89278

Clé K de 6bits Choisie dans la SBOX 5 : 0 donc on concat avec l'ancien K

K16 courant = 36249E00

Clé K de 6bits Choisie dans la SBOX 6 : 37 donc on concat avec l'ancien K

K16 courant = D89278037

Clé K de 6bits Choisie dans la SBOX 7 : 35 donc on concat avec l'ancien K

K16 courant = 36249E00DF5

Clé K de 6bits Choisie dans la SBOX 8 : 6 donc on concat avec l'ancien K

K16 courant = D89278037D46

##################### FIN RECHERCHE K16 48BITS ##############################

K16_48bits = D89278037D46
```

FIGURE 2.2 – Clé K16 de 48bits

3 Question 3 : Retrouver la clé complète du DES

3.1 Retrouver les 8bit manquants

Maintenant qu'on a réussi à retrouver la clé K_{16} de 48 bits, il nous faut retrouver les 8bits manquants pour avoir celle de 56bits ainsi que les 8bits de parités restants pour celle de 64bits.Pour commencer analysons la key schedule :

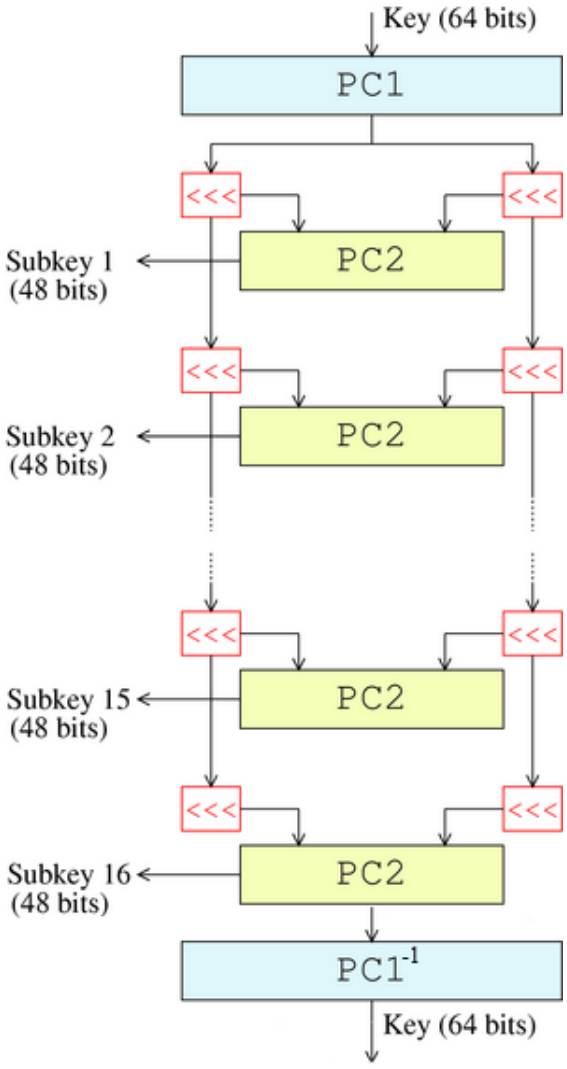

FIGURE 3.1 – Key Schedule

Pour avoir K 64bits,il suffit donc de calculer :

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

Cependant lorsque l'on va effectuer la permutation PC2 inverse,nous allons passer de 48bits à 56bits donc nous allons perdre la valeur de 8bits.
Après avoir étudié ,la permutation PC2 nous avons donc pu déduire la position de ces 8bits dans PC2 inverse :

```
//Les bits 9,18,22,25,35,38,43,54 sont perdus donc on laisse 0
static int inverse_PC2[] =
{
    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
};
```

TablesPermu.h

Dans un premier temps pour récupérer K 56bits,nous avons implémenté l'algorithme du DES pour pouvoir effectuer une recherche exhaustive sur ces 8bits perdus,il y a donc 256 possibilités soit 2^8.On va donc pouvoir tester toutes les positions possibles de ces 8 bits et si le message chiffré obtenu correspond avec celui obtenue avec le message clair alors nous avons la bonne clé de 56bits :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "K16_56.h"
#include "../DES/Permutation.h"
#include "../Fonctions/fonctions.h"
#include "../DES/Tables_Permu.h"
#include "../DES/Des.h"

/*! \file       K16_56.c
 *  \brief      Fichier contenant les fonctiçons necessaires pour retrouver K16 56bits
 *  \author     ROBIN JORAN
 *  \version    1.00
 *  \date       25 Mars 2019
 */

 /*! \fn long bitsPerdus(long mask)
 *  \brief Fonction qui permet de retrouver les bits perdus de la clé K
 *  \param mask : On applique le mask
 *  \return recup : la clé K avec les bits perdus
 */
long bitsPerdus(long mask)
{
    long recup = 0;
    long bits_perdu[] = {14,15,19,20,51,54,58,60};

    for(int i = 0;i<8;i++)
    {
        recup = concat(recup,shift_gauche(et_binaire(shift_droit(mask,i),1),(64 - bits_perdu[i])));
    }

    return recup;
```

```c
}

 /*! \fn long trouverK16_56b(long msg_Clair,long msg_EncJuste,long K16_48b)
  *  \brief Fonction qui permet de retrouver la clé K16 de 56 bits
  *  \param msg_Clair : message Clair
  *  \param msg_EncJuste : message crypté juste
  *  \param K16_48bits : Clé K16 de 48bits
  *  \return K16 la clé de 56 bits
  */
long trouverK16_56b(long msg_Clair,long msg_EncJuste,long K16_48b)
{
    long K48_56;//K après inverse PC1
    long K56_64;//K après inverse PC2

    long mask = 0;
    long RE_K ;//Recherche exhaustive pour trouver les 8bits faux

    K48_56 = permutation(K16_48b,inverse_PC2,48,56);//Permutation PC2 inverse pour recuperer K48 sous forme
    56

    K56_64 = permutation(K48_56,inverse_PC1,56,64);//Permutation PC1 inverse pour recupere K56 sous forme 64

    RE_K = K56_64;

    while(mask<256 && msg_EncJuste != DES(msg_Clair,RE_K) )
    {
        RE_K = concat(K56_64,bitsPerdus(mask));//2^8 =256 possibilités
        mask++;
    }

    if (mask == 256)
    {
        printf("\nErreur K 56bits\n");
    }

    return RE_K;

}
```

K16_56.c

Maintenant que nous avons l clé de 56bits, Il ne nous reste plus qu'à truvr les 8bits de parités.Les bit de parités n'affectent pas le résultat du DES car l DES utilise une clé de 56bits.On a procédé de la façon suivante pou retrouver ces 8Bits de parités :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../K16_56/K16_56.h"
#include "K64.h"
#include "../DES/Permutation.h"
#include "../Fonctions/fonctions.h"
#include "../DES/Tables_Permu.h"
#include "../DES/Des.h"

/*! \file      K64.c
 *  \brief     Fichier contenant les fonctiçons necessaires pour retrouver K 64bits
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

 /*! \fn long bitsParite(long K56b)
  *  \brief Fonction qui permet de retrouver les bits de parité
```

```
 *  \param K56b : clé K 56bits
 *  \return recup : K 64bits
 */
long recup_bits_de_Parite(long K56b)
{
    long recup = K56b;
    long test, parite;

    for(int i=0;i<8;i++)
    {
        parite = 0;

        for(int j=0;j<8;j++)
        {
            if(j!=7)
            {
                test = shift_gauche(1,( ((7 - i) * 8) + (7 - j) ));
                test = et_binaire(K56b,test);

                if(test)
                {
                    parite = or_exclu(parite,1);
                }
                else
                {
                    parite = or_exclu(parite,0);
                }
            }
            else
            {
                test = !parite;
                recup = concat(recup,shift_gauche(test,((7 - i) * 8)));
            }
        }
    }
    return recup;
}

 /*! \fn long trouver_K64(long clair, long chiffre, long K16)
 *  \brief Fonction qui permet de retrouver K 64bits
 *  \param msg_Clair : message clair
 *  \param msg_EncJuste : message chiffré
 *  \param K16_56b : clé K 56bits
 *  \return K64b : K 64bits
 */
long trouver_K64(long msg_Clair,long msg_EncJuste,long K16_56b)
{
    printf("\n########################## RECHERCHE K16 56BITS ##########################\n\n");

    long K56b = trouverK16_56b(msg_Clair, msg_EncJuste, K16_56b);

    printf("\n########################## FIN RECHERCHE K16 56BITS ##########################\n\n");

    printf("\K 56bits = %lX\n", K56b);

    printf("\n########################## RECHERCHE K 64BITS ##########################\n\n");

    long K64b = recup_bits_de_Parite(K56b);

    return K64b;
}
```

K64.c

On a donc découpé la clé de 56bits par blocs de 7 et on a calculé la valeur du 8ème bit de chaque bloc en fonction de la parité du nombre de 1 dans les blocs

de 7bits .Si le nombre de 1 est impair on complète par un 0 sinon par un 1.

3.2 Clé K complète obtenue

Voici ma clé K de 56bits : F6 C2 60 1A 4C 10 42 FA

```
####################### RECHERCHE K16 56BITS ######################


######################## FIN RECHERCHE K16 56BITS ########################

K 56bits = F6C2601A4C1042FA
```

FIGURE 3.2 – Clé K16 de 56bits

Voici ma clé K de 64bits : F7 C2 61 1A 4C 10 43 FB

```
####################### RECHERCHE K 64BITS ######################


K 64bits = F7C2611A4C1043FB

######################## FIN RECHERCHE K 64BITS ########################
```

FIGURE 3.3 – Clé K16 de 64bits

3.3 Vérification de la clé

On vérifie que l'on obtient bien le bon chiffré avec la clé K 64bits que l'on a obtenu

```
1       long verif;
2       long K16_48bits;
3       long K_64bits;
4
5       printf("\n######################## RECHERCHE K16 48BITS ########################\n\n");
6
7       K16_48bits = trouverK16_48b(msg_EncJuste,msg_EncFaux);
8
9       printf("\nK16_48bits = %lX \n",K16_48bits);
10
11
12
13
14      K_64bits=trouver_K64(msg_clair,msg_EncJuste,K16_48bits);
15
16      printf("\nK 64bits = %lX\n", K_64bits);
17
18      printf("\n######################## FIN RECHERCHE K 64BITS ########################\n\n");
19
20      printf("\n######################## VERIFICATION K 64 BITS ########################\n\n");
21
22      verif=DES(msg_clair,K_64bits);
23
```

```
24      printf("Message Clair = %lX \n",msg_clair);
25      printf("Message Chiffré Juste = %lX \n",msg_EncJuste);
26      printf("Chiffré trouvé avec DES et K64 = %lX \n",verif);
27
28      return 0;
```

projet_des.c

```
###################### VERIFICATION K 64 BITS ######################

Message Clair = 4E334626E9DC4BC
Message Chiffré Juste = F3C23DEEF7FE5DCB
Chiffré trouvé avec DES et K64 = F3C23DEEF7FE5DCB
user@debian:~/Bureau/Projet Calcul_Securise/Joran$
```

FIGURE 3.4 – Vérification

Voici les messages qui m'ont été attribués :

```
1  #ifndef Messages_H
2  #define Messages_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  /*! \file      Messages.h
9   *  \brief     Fichier contenant les messages fournis
10  *  \author    ROBIN JORAN
11  *  \version   1.00
12  *  \date      25 Mars 2019
13  */
14
15      long msg_clair = 0x04E334626E9DC4BC;
16
17      long msg_EncJuste =0xF3C23DEEF7FE5DCB;
18
19      long msg_EncFaux[] ={0xF1D73DEAF7FE5DDF,0xF3D03DEEF7FF5DCB,0xF3C23FAEF7FF5DCB,0xF28239A8E7FF5DCB,
20                           0xF3923DEAE5FE5DCB,0xF2823DEEF7FC5DCB,0xF38239EEE7FE5FCB,0xF28239EEB7FA5DC9,
21                           0xFA8239EFA7EA5DCB,0xF3CA3DEEB7EA5DCB,0xF3C235EFF7EE5DCB,0xF3C22DE7F7BA5DCB,
22                           0xF3C22DEEBFEA5DCA,0xB3C22DEEF7B65DCA,0xF3C22DEEF7BE55CB,0xF3C22DEEF3BE5D82,
23                           0X93C22DEEF3BE5D8B,0xF3E23DEEF3FE5C8B,0xF3C21DEEF3FE4D8B,0XE3C27CCEF3FE4C8B,
24                           0xF3C27CEED3FE4D8B,0xE7C27CEEF7DE5DCB,0xF7C27DEEF7FE7DCB,0xE3C27CEEF6FE1DEB,
25                           0x63C27DEEF6FE19CB,0xF3423DFEF6FE5DCB,0xF3C2BDEEF6FE59CB,0XF3C63D7EF7FE19DF,
26                           0xF3C73DFE76FE5DDF,0xF3C73DEEF77E5DCF,0xF3C63DEEF7FEDDDF,0XF3D63DAAF7FF5D5B};
27
28 #endif
```

Messages.h

Rappel : Tout le code source ainsi que les headers (Messages,TablesPermu) ont été mis entièrement à disposition dans l'Annexe.

4 Question 4(plus difficile) : Fautes sur les tours précédents

Dans les questions précédentes ,nous avons trouvé qu'une attaque par fautes sur le $15^{\text{ème}}$ tour a une complexité de $3*2^{10}$.

On peut donc réutiliser les formules établies pour l'attaque sur R_{15}.On va donc devoir analyser la position des bits fautés comme on l'avait fait pour l'attaque du 15ème tour pour qu'il y ait propagation d'un bit faux jusqu'au $16^{\text{ème}}$ tour pour chacune des 8 SBOX à attaquer.

Mais le traçage de la propagation du bit faux devient impossible à cause de la fonction f car on ne connait pas la sous-clé à chaque tour, avant d'arriver au $16^{\text{ème}}$.

On peut essayer de faire une recherche de K_{15} afin d'analyser le traçage de la propagation des bits faux. Cependant nous devons connaitre aussi L_{15} et L_{15}^{f}.

On peut donc en déduire la complexité qui est donc élevée au carré pour chaque tour précédent le $16^{\text{ème}}$.

On obtient donc :
Pour le $14^{\text{ème}}$ tour, la complexité sera environ de 2^{20}.
Pour le $13^{\text{ème}}$ tour, la complexité sera environ de 2^{40}.
Pour le $12^{\text{ème}}$ tour, la complexité sera environ de 2^{80} ce qui est impossible à calculer de nos jours.

Donc la compléxité reste raisonnable jusqu'au $13^{\text{ème}}$ tour.

5 Question 5 : Contre-mesures

L'objectif d'une attaque par faute est la générations de celles-ci dans le circuit d'exécution d'un algorithme. Les contre-mesures contre ce type d'attaques peuvent être déployées à tous les niveaux entre le matériel et l'application mais les plus efficaces sont celles qui utilisent des mécanismes de détection ou correction d'erreur au sein du circuit.Voici différentes contre-mesures possibles sur ce type d'attaques par fautes contre le DES :

Détection ou correction par redondance matérielle :

Le principe de la redondance matérielle est de réaliser la même opération sur plusieurs copies d'un même bloc de calcul et d'en comparer les résultats. Ce principe a été utilisé pour développer plusieurs schémas de protection.

La duplication simple avec comparaison est basée sur l'utilisation de deux copies en parallèle du même bloc, suivies par la comparaison des deux résultats.Dans ce cas 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.

La duplication multiple avec comparaison est une extension de la duplication simple à un nombre quelconque de copies du bloc de calcul. La triplication est une des protections les plus utilisées.Dans ce cas les ressources de la carte à puce qui effectue le calcul seront réparties sur 3 calculs, et le temps de calcul va donc être au pire des cas multiplié par 3.

Il en existe aussi pleins d'autres comme la duplication simple avec redondance complémentaire , la duplication dynamique , la duplication hybride ...

Détection ou correction par redondance temporelle :

La redondance temporelle est basée sur la ré-exécution d'un même calcul sur le même bloc matériel et la comparaison des différents résultats obtenus. Ce principe se décline aussi sur plusieurs schémas de protection.

La redondance temporelle simple est basée sur la double exécution d'un calcul sur un même bloc de calcul. Les résultats ainsi obtenus sont donc comparés.Le temps de calcul sera donc multiplié par 2.

La redondance temporelle multiple se base sur l'exécution multiple de la même opération sur la même unité de calcul.Le temps de calcul sera donc multiplié par le nombre d'exécution.

Il en existe aussi d'autres comme la redondance temporelle simple avec opérande inversée,la redondance temporelle simple avec rotation des opérandes ...

Détection ou correction par redondance d'information :

Le dernier type de redondance utilisable est la 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.

6 Annexes(CODE)

6.1 Permutation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Permutation.h"
#include "../Fonctions/fonctions.h"
#include "../DES/Tables_Permu.h"

/*! \file      Permutation.c
 *  \brief     Fichier contenant la fonction de permutation selon les différentes tables
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

 /*!\fn long permutation(long msg, int tab_perm[] ,int tailleBits_in, int tailleBits_out)
 *  \brief Fonction qui permet d'appliquer une Permutation avec le nombre d'éléments de tab_perm[] équivalent à tailleBits_out
 *  \param msg : Message à permuter selon la table
 *  \param tab_perm[] : tableau d'entiers d'une Permutation
 *  \param tailleBits_in : entier qui représente le nombre de bits à l'entrée de la permutation
 *  \param tailleBits_out : entier qui représente le nombre de bits à la sortie de la permutation
 *  \return final contenant le résultat final de la permutation correspondante
 */
long permutation(long msg, int tab_perm[] ,int tailleBits_in, int tailleBits_out)
{
    long tempo;

    long final = 0x0;
    long bit_position = 0x1;

    for (int i=0;i<tailleBits_out;i++)
    {
        tempo = et_binaire(msg, shift_gauche(bit_position,tailleBits_in - tab_perm[i]));//Recuperation du bit

        if (tempo != 0)
        {
            tempo = bit_position;
            tempo = shift_gauche(tempo,(tailleBits_out - i - 1));//On place le bit à sa place
        }

        final = concat(final,tempo);//Concatenation de ce bit avec final
    }
    return final;
}
```

Permutation.c

6.2 K16 48bits

```c
   #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "K16_48.h"
#include "../DES/Permutation.h"
#include "../Fonctions/fonctions.h"
#include "../DES/Tables_Permu.h"

/*! \file      K16_48.c
```

```
10  *  \brief     Fichier contenant les fonctions necessaires pour retrouver K16 48bits
11  *  \author    ROBIN JORAN
12  *  \version   1.00
13  *  \date      25 Mars 2019
14  */
15
16  /*! \fn void affiche_sol(int NUM_SBOX,int t1[],int t2[],int s)
17  *  \brief Fonction qui permet d'afficher les résultats
18  *  \param NUM_SBOX : numéro courant de la SBOX
19  *  \param solPossibles[][6] : Tableau des solutions possibles
20  *  \param cle[][6][64] : Tableau des clé Possibles
21  */
22 void affiche_sol(int NUM_SBOX,int solPossibles[][6],int cle[][6][64])
23 {
24      printf("\n######################### Sbox %d ######################### \n\n", NUM_SBOX + 1);
25
26         for (int nbr_faux=0;nbr_faux<6;nbr_faux++)
27         {
28             printf("Chiffré Faux %d : %d clés K16 ->    ", nbr_faux+1, solPossibles[NUM_SBOX][nbr_faux]);
29
30             for (int i =0;i<solPossibles[NUM_SBOX][nbr_faux];i++)
31             {
32                 printf("%X ", cle[NUM_SBOX][nbr_faux][i]);
33             }
34             printf("\n");
35         }
36 }
37
38  /*! \fn int analyse_bit_faute(long msg_EncJuste,long msg_EncFaux[],int num)
39  *  \brief Fonction qui permet d'analysé la position du bit fauté sur chaque chiffré
40  *  \param msg_EncJuste : Message crypté juste
41  *  \param msg_EncFaux : Message crypté faux
42  *  \param num : numéro du chiffré
43  */
44 void analyse_bit_faute(long msg_EncJuste,long msg_EncFaux[],int num)
45 {
46     long msg_Dec_EncJuste,msg_Dec_EncFaux;//Dechiffré OK par rapport au message crypté OK
47     long R16,R16f,R15,R15f;
48     long position_bit_faute;
49     int tmp_pos =33;
50
51     //~ printf("\n######################### PERMUTATION IP Chiffrés Juste #########################\n\n");
52
53     msg_Dec_EncJuste = permutation(msg_EncJuste, IP, 64, 64);
54
55     R16 = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);
56     R16 = decoupage_msgR(R16,0xFFFFFFFF);
57
58     R15=R16;
59
60         //~ printf("\n######################### PERMUTATION IP Chiffrés Faux #########################\n\n");
61
62         msg_Dec_EncFaux = permutation(msg_EncFaux[num], IP, 64, 64);
63
64         R16f = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);
65         R16f = decoupage_msgR(R16f,0xFFFFFFFF);
66
67         R15f=R16f;
68
69         position_bit_faute = or_exclu(R15,R15f);
70
71         //~ printf("\n######################### XOR #########################\n\n");
72
73         while(position_bit_faute)
74         {
75             tmp_pos--;
76             position_bit_faute = position_bit_faute >> 1;
77         }
```

```
78
79          //~ printf("\n########################## POSITION BIT FAUTE ##########################\n\n");
80
81          if(num <9)
82          {
83              printf("Chiffré faux  %d --> Position bit fauté %d \n",num+1,tmp_pos);
84          }
85          else
86          {
87              printf("Chiffré faux %d --> Position bit fauté %d \n",num+1,tmp_pos);
88          }
89  }
90
91   /*! \fn long recupSolution(long cle[][6][64],solPossibles[][6],int NUM_SBOX,long K16)
92   *  \brief Fonction qui permet de récupérer la valeur de K16_46bits
93   *  \param cle[][6][64] : Tableau des 64 solutions pour les 8 Sbox avec les 6 chiffrés fautés correspondants
94   *  \param solPossibles[][6] : Nombre de solutions possibles des 6 fautes pour les 8 Sbox
95   *  \param NUM_SBOX : numéro SBOX
96   *  \param K16 : K16
97   *  \return K16 48bits
98   */
99  long recupSolution(int cle[][6][64],int solPossibles[][6],int NUM_SBOX,long K16)
100 {
101     int test = 0;
102     int final_solution;
103
104     long elu = (long) cle[NUM_SBOX][0][test];
105
106         for (int nbr_Faux = 1; nbr_Faux < 6; nbr_Faux++)
107         {
108             for (final_solution = 0;final_solution<solPossibles[NUM_SBOX][nbr_Faux];final_solution++)
109             {
110                 if (elu == cle[NUM_SBOX][nbr_Faux][final_solution])
111                 {
112                     break;
113                 }
114             }
115
116             if(final_solution == solPossibles[NUM_SBOX][nbr_Faux])
117             {
118                 if(test+1 >= solPossibles[NUM_SBOX][0])
119                 {
120                     printf("\nProblème Sbox numéro %d,chiffré faux %d\n",NUM_SBOX,nbr_Faux);
121                     break;
122                 }
123                 nbr_Faux = 1;
124                 ++test;
125                 elu = (long) cle[NUM_SBOX][0][test];
126             }
127         }
128         printf("\nClé K de 6bits Choisie dans la SBOX %d : %lX donc on concat avec l'ancien K\n",NUM_SBOX+1,
    elu);
129         K16 = shift_gauche(K16,6);
130         K16 = concat(K16,elu);
131         printf("\nK16 courant = %lX\n",K16);
132
133         return K16;
134 }
135
136  /*! \fn long trouverK16_48b(long msg_EncJuste, long msg_EncFaux[]);
137  *  \brief Fonction qui permet de retrouver la clé K16 de 48 bits
138  *  \param msg_EncJuste : message crypté juste
139  *  \param msg_EncFaux : messages cryptés faux
140  *  \return K16 la clé de 48 bits
141  */
142 long trouverK16_48b(long msg_EncJuste, long msg_EncFaux[])
143  {
144      long K16 = 0;//48bits
```

```
145     long msg_Dec_EncJuste;//Dechiffré OK par rapport au message crypté OK
146     long msg_Dec_EncFaux;//Dechiffré Faux OK par rapport au message faux crypté OK
147     long check;//Verification de 32bits
148     long entree_SBOX,entree_SBOXf;//Entree des Sbox
149     long EXP_R15, EXP_R15f;//Expansion
150     long L16,R16,R15,L16f,R16f,R15f;//L et R fautés ou non
151
152
153     int cle[8][6][64] = {{{0}}};//Tableau des 64 solutions pour les 8 Sbox avec les 6 chiffrés fautés
    correspondants
154     int solPossibles[8][6] = {{0}};//Nombre de solutions possibles des 6 chiffrés faux pour les 8 Sbox
155
156     int nbr_faux;
157
158     int check_4bits;//Sbox de verification
159     int SBOX_4bits;//Obtenue avec les fautés
160     int ligne,colonne;//SBOX
161     int ligne_f,colonne_f;//SBOX fautée
162     int NUM_SBOX ;//Numéro SBOX
163     int RE_k16;//recherche exaustive K16
164
165
166     //~ printf("\n########################## Analyse Bit Fauté ##########################\n\n");
167
168     //~ for (int analyse=0;analyse<32;analyse++)
169     //~ {
170         //~ analyse_bit_faute(msg_EncJuste,msg_EncFaux,analyse);
171     //~ }
172
173   //printf("\n### PERMUTATION IP  Chiffré Juste ### \n");
174
175   msg_Dec_EncJuste = permutation(msg_EncJuste, IP, 64, 64); //Permutation IP avec le chiffré
176
177   //printf("\n### DECOUPAGE L16 et R16 Juste ### \n");
178
179   L16 = decoupage_msgL(msg_Dec_EncJuste,0xFFFFFFFF);//On stocke la partie L16
180
181   R16 = decoupage_msgR(msg_Dec_EncJuste,0xFFFFFFFF);//On stocke la partie R16
182   R15 = R16;//On stocke R16 dans R15
183
184       //~ printf("\n### PERMUTATION IP 32 chiffrés Faux ### \n");
185       //~ printf("\n### DECOUPAGE 32 Chiffrés Faux L16F et R16F ### \n");
186
187       //~ printf("\n### PERMUTATION INVERSE DE P avec L16 ^ L16f ### \n");
188
189       //~ printf("\n### EXPANSION R15 et R15F ### \n");
190       //~ printf("\n### DECOUPAGE EXPANSION en 8blocs de 6bits ### \n");
191       //~ printf("\n### XOR AVEC RECHERCHE EXHAUSTIVE K16 ### \n");
192       //~ printf("\n### INJECTION DANS LES BONNES SBOX### \n");
193       //~ printf("\n### CALCUL LIGNES ET COLONNES DES SBOX ### \n");
194
195       //~ printf("\n### DECOUPAGE PERMUTATION INVERSE DE P avec L16 ^ L16f en 8blocs 4bits ### \n");
196       //~ printf("\n### CORRESPONDANCE AVEC LES BONNES SBOX ### \n");
197       //~ printf("\n### RECUPERATION DES VALEURS SBOX CHIFFRE JUSTE ET 32 FAUX### \n");
198       //~ printf("\n### XOR DES 2 SBOX ### \n");
199       //~ printf("\n### COMPARAISON ENTRE CHECK ET VALEUR SBOX DE SORTIE ### \n");
200       //~ printf("\n### SI EGAL ALORS STOCKAGE DE K16 6bits par SBOX ### \n");
201       //~ printf("\n### STOCKAGE ELEMENT COMMUN SUR 6 FAUTE PAR SBOX ### \n");
202       //~ printf("\n### CONCATENATION DES MORCEAUX DE K16 ### \n");
203       //~ printf("\n### AFFICHAGE ### \n");
204
205       //Recherche exhaustive
206       for(NUM_SBOX=0;NUM_SBOX<8;NUM_SBOX++)
207       {
208           for(nbr_faux=0;nbr_faux<6;nbr_faux++)
209           {
210               msg_Dec_EncFaux = permutation(msg_EncFaux[pos_bitfaux[NUM_SBOX][nbr_faux]], IP, 64, 64);//
    Permutation IP chiffré FAUX
```

```
211
212                 L16f = decoupage_msgL(msg_Dec_EncFaux,0xFFFFFFFF);//Stockage L16f
213                 R16f = decoupage_msgR(msg_Dec_EncFaux,0xFFFFFFFF);//Stockage R16f
214                 R15f=R16f;//Stockage R15f
215
216                 check = permutation(or_exclu(L16,L16f),inverse_P,32,32);//Valeur de Verification pour la
    sortie de SBOX
217
218                 EXP_R15=permutation(R15,E,32,48);//Expansion R15
219                 EXP_R15f=permutation(R15f,E,32,48);//Expansion R15f
220
221                 for (RE_k16=0;RE_k16<64;RE_k16++)
222                 {
223                     entree_SBOX = or_exclu(shift_droit(decoupe_6bits(EXP_R15,NUM_SBOX),trouver_SBOX_6bits(
    NUM_SBOX)),RE_k16);//Entrée Sbox 6bits XOR k16
224                     entree_SBOXf = or_exclu(shift_droit(decoupe_6bits(EXP_R15f,NUM_SBOX),trouver_SBOX_6bits(
    NUM_SBOX)),RE_k16);//Entrée Sboxf 6bits XOR k16
225
226                     ligne = calcul_lignes_SBOX(entree_SBOX);//Ligne SBOX
227                     colonne = calcul_col_SBOX(entree_SBOX);//Colonne SBOX
228
229                     ligne_f = calcul_lignes_SBOX(entree_SBOXf);//Ligne SBOXf
230                     colonne_f = calcul_col_SBOX(entree_SBOXf);//colonne SBOXf
231
232                     check_4bits =shift_droit(decoupe_4bits(check,NUM_SBOX),trouver_SBOX_4bits(NUM_SBOX));//
    Valeur de Verification pour la sortie de SBOX 4bits
233                     SBOX_4bits = or_exclu(Sbox[NUM_SBOX][ligne][colonne],Sbox[NUM_SBOX][ligne_f][colonne_f])
    ;//Valeur de sortie SBOX 4 bits
234
235                     if(check_4bits == SBOX_4bits)//Comparaison
236                     {
237                         cle[NUM_SBOX][nbr_faux][solPossibles[NUM_SBOX][nbr_faux]] = RE_k16;//Stockage
    valeurs clés
238                         ++solPossibles[NUM_SBOX][nbr_faux];//Incrementation
239                     }
240                 }
241             }
242
243             //affiche_sol(NUM_SBOX,solPossibles,cle);//Affichage Possibilités par SBOX
244
245             K16=recupSolution(cle,solPossibles,NUM_SBOX,K16);//Récupération de la solution commune des 6
    chiffrés fautés par SBOX puis Concatenation de la clé K16
246         }
247
248     printf("\n###################### FIN RECHERCHE K16 48BITS ##############################\n");
249
250     return K16;
251 }
```

K16_48.c

6.3 DES

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "Permutation.h"
5  #include "Des.h"
6  #include "fonction_f.h"
7  #include "algo_keyschedule.h"
8  #include "../Fonctions/fonctions.h"
9  #include "Tables_Permu.h"
10
11 /*! \file       Des.c
12  *  \brief      Fichier contenant l'algorithme de DES.
```

```
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

/*! \fn long DES(long msg_clair, long cle_K)
 *  \brief Fonction qui permet execute l'algorithme de DES
 *  \param msg_clair :  message clair
 *  \param cle_K : clé K
 *  \return un message crypté
 */
long DES(long msg_clair,long K64)
{
    long sous_cles[16]={0};//16 sous-clés K48 de K64
    long L,R;//Partie L et R du msg clair apres permutation IP
    long Li,Ri;//les Li et Ri de chaque tour

    DES_sousCle(sous_cles,K64);//Calcule des 16 sous-clés K48

    L = decoupage_msgL(permutation(msg_clair,IP,64,64),0xFFFFFFFF);//Calcule de L
    R = decoupage_msgR(permutation(msg_clair,IP,64,64),0xFFFFFFFF);//Calcule de R

    for (int i=0;i<16;i++)
    {
        Li = R; // Li recoit R
        Ri = or_exclu(L,f(R,sous_cles[i])); //Ri recoit L xor f

        L = Li;//on stocke le nouvel Li dans L
        R = Ri;//on stocke le nouvel Ri dans R
    }

    return permutation(concat(shift_gauche(R,32),L),inverse_IP,64,64);//Permutation IP inverse

}
```

Des.c

6.4 Key Schedule

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Permutation.h"
#include "Tables_Permu.h"
#include "algo_keyschedule.h"
#include "../Fonctions/fonctions.h"

/*! \file      algo_keyschedule.h
 *  \brief     Fichier contenant l'algorithme des sous clés.
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

 /*! \fn long Rotation_Gauche(long K56)
 *  \brief Fonction qui permet la Rotation à gauche de chaque moitié de K56
 *  \param K56 :  clé 56bits
 *  \return un message crypté
 */
long Rotation_Gauche(long K56)
{
    long C28,D28;
    long test;

    C28 = shift_droit(et_binaire(K56,0xFFFFFFF0000000),28);
```

```
27      D28 = et_binaire(K56,0xFFFFFFF);
28
29      test = shift_droit(C28,27);//Choix du bit le plus à droite
30      C28 = shift_gauche(C28,1);//Décalage des bits à gauche
31      C28 = et_binaire(C28,0xFFFFFFF);//Suppression du bit de poids fort
32      C28 = concat(C28,test);//Mis à la position voulue
33
34      test = shift_droit(D28,27);//Choix du bit le plus à droite
35      D28 = shift_gauche(D28,1);//Décalage des bits à gauche
36      D28 = et_binaire(D28,0xFFFFFFF);//Suppression du bit de poids fort
37      D28 = concat(D28,test);//Mis à la position voulue
38
39      C28 = shift_gauche(C28,28);
40
41      return et_binaire(concat(C28,D28),0xFFFFFFFFFFFFFF);
42  }
43
44   /*! \fn void DES_sousCle(long sous_cles[], long K)
45   *  \brief Fonction qui permet de calculer chaque sous clé
46   *  \param sous_cles[] :  Tableau des sous-clés
47   *  \param K :  Clé K
48   */
49  void DES_sousCle(long sous_cles[],long K64)
50  {
51      //Permutation Key Schedule
52      int perm_ks[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
53
54      long K56 = permutation(K64,PC1,64,56);// Permutation de K64 dans PC1 donc on a K56
55
56      for(int i=0;i<16;i++)
57      {
58          if (perm_ks[i] == 1)
59          {
60              K56 = Rotation_Gauche(K56);
61          }
62          else
63          {
64              K56 = Rotation_Gauche(Rotation_Gauche(K56));
65          }
66
67          sous_cles[i] = permutation(K56,PC2,56,48);//Permutation de K56 dans PC2 pour obtenir les 16 sous clé
    s K48
68      }
69  }
```

algo_keyschedule.c

6.5 Fonction f

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "Permutation.h"
5  #include "../Fonctions/fonctions.h"
6  #include "Tables_Permu.h"
7
8   /*! \file      fonction_f.c
9    *  \brief     Fichier contenant la fonction f de DES.
10   *  \author    ROBIN JORAN
11   *  \version   1.00
12   *  \date      25 Mars 2019
13   */
14
15   /*! \fn long f(long R, long sous_cle)
16   *  \brief Fonction qui permet d'appliquer la fonction f du DES
```

```
 *  \param R :  partie R du message
 *  \param sous_cle : clé sous_cle
 *  \return un message 48bits
 */
long f(long R, long sous_cle)
{
    long Enter_Sbox;//Entree SBOX
    long Sortie_Sbox=0x0;//Sortie SBOX
    long expansion;// Expansion
    long ligne,colonne;//Ligne et Colonne SBOX
    long bloc_6bits;// Decoupage 6bits pour entréee des SBOX

    expansion = permutation(R,E,32,48);// Expansion de R

    Enter_Sbox = or_exclu(expansion,sous_cle);//Enter_Sbox = R xor les sous-clés sous_cle

    for (int i=0;i<8;i++)
    {
        bloc_6bits = shift_droit(Enter_Sbox,et_binaire((48 - 6 - i*6),0x3F));//Recuperation des 6bits pour chaque SBOX

        ligne = calcul_lignes_SBOX(bloc_6bits);//Calcul Ligne SBOX
        colonne = calcul_col_SBOX(bloc_6bits);//Calcul Colonne SBOX

        Sortie_Sbox = concat(Sortie_Sbox,shift_gauche(Sbox[i][ligne][colonne],(32 - 4 - i*4)));//Concatenation des sorties de chaque SBOX
    }

    return permutation(Sortie_Sbox,P,32,32);
}
```

fonction_f.c

6.6 K16 56bits

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "K16_56.h"
#include "../DES/Permutation.h"
#include "../Fonctions/fonctions.h"
#include "../DES/Tables_Permu.h"
#include "../DES/Des.h"

/*! \file      K16_56.c
 *  \brief     Fichier contenant les fonctiçons necessaires pour retrouver K16 56bits
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

 /*! \fn long bitsPerdus(long mask)
 *  \brief Fonction qui permet de retrouver les bits perdus de la clé K
 *  \param mask : On applique le mask
 *  \return recup : la clé K avec les bits perdus
 */
long bitsPerdus(long mask)
{
    long recup = 0;
    long bits_perdu[] = {14,15,19,20,51,54,58,60};

    for(int i = 0;i<8;i++)
    {
        recup = concat(recup,shift_gauche(et_binaire(shift_droit(mask,i),1),(64 - bits_perdu[i])));
    }
```

```
31
32      return recup;
33  }
34
35   /*! \fn long trouverK16_56b(long msg_Clair,long msg_EncJuste,long K16_48b)
36   *  \brief Fonction qui permet de retrouver la clé K16 de 56 bits
37   *  \param msg_Clair : message Clair
38   *  \param msg_EncJuste : message crypté juste
39   *  \param K16_48bits : Clé K16 de 48bits
40   *  \return K16 la clé de 56 bits
41   */
42  long trouverK16_56b(long msg_Clair,long msg_EncJuste,long K16_48b)
43  {
44      long K48_56;//K après inverse PC1
45      long K56_64;//K après inverse PC2
46
47      long mask = 0;
48      long RE_K ;//Recherche exhaustive pour trouver les 8bits faux
49
50      K48_56 = permutation(K16_48b,inverse_PC2,48,56);//Permutation PC2 inverse pour recuperer K48 sous forme
        56
51
52      K56_64 = permutation(K48_56,inverse_PC1,56,64);//Permutation PC1 inverse pour recupere K56 sous forme 64
53
54      RE_K = K56_64;
55
56      while(mask<256 && msg_EncJuste != DES(msg_Clair,RE_K) )
57      {
58          RE_K = concat(K56_64,bitsPerdus(mask));//2^8 =256 possibilités
59          mask++;
60      }
61
62      if (mask == 256)
63      {
64          printf("\nErreur K 56bits\n");
65      }
66
67      return RE_K;
68
69  }
```

K16_56.c

6.7 K 64bits

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "../K16_56/K16_56.h"
5   #include "K64.h"
6   #include "../DES/Permutation.h"
7   #include "../Fonctions/fonctions.h"
8   #include "../DES/Tables_Permu.h"
9   #include "../DES/Des.h"
10
11  /*! \file       K64.c
12   *  \brief      Fichier contenant les fonctiçons necessaires pour retrouver K 64bits
13   *  \author     ROBIN JORAN
14   *  \version    1.00
15   *  \date       25 Mars 2019
16   */
17
18   /*! \fn long bitsParite(long K56b)
19   *  \brief Fonction qui permet de retrouver les bits de parité
20   *  \param K56b : clé K 56bits
```

```
21   *  \return recup : K 64bits
22   */
23  long recup_bits_de_Parite(long K56b)
24  {
25      long recup = K56b;
26      long test, parite;
27
28      for(int i=0;i<8;i++)
29      {
30          parite = 0;
31
32          for(int j=0;j<8;j++)
33          {
34              if(j!=7)
35              {
36                  test = shift_gauche(1,( ((7 - i) * 8) + (7 - j) ));
37                  test = et_binaire(K56b,test);
38
39                  if(test)
40                  {
41                      parite = or_exclu(parite,1);
42                  }
43                  else
44                  {
45                      parite = or_exclu(parite,0);
46                  }
47              }
48              else
49              {
50                  test = !parite;
51                  recup = concat(recup,shift_gauche(test,((7 - i) * 8)));
52              }
53          }
54      }
55      return recup;
56  }
57
58   /*! \fn long trouver_K64(long clair, long chiffre, long K16)
59   *  \brief Fonction qui permet de retrouver K 64bits
60   *  \param msg_Clair : message clair
61   *  \param msg_EncJuste : message chiffré
62   *  \param K16_56b : clé K 56bits
63   *  \return K64b : K 64bits
64   */
65  long trouver_K64(long msg_Clair,long msg_EncJuste,long K16_56b)
66  {
67      printf("\n########################## RECHERCHE K16 56BITS ##########################\n\n");
68
69      long K56b = trouverK16_56b(msg_Clair, msg_EncJuste, K16_56b);
70
71      printf("\n########################## FIN RECHERCHE K16 56BITS ##########################\n\n");
72
73      printf("\K 56bits = %lX\n", K56b);
74
75      printf("\n########################## RECHERCHE K 64BITS ##########################\n\n");
76
77      long K64b = recup_bits_de_Parite(K56b);
78
79      return K64b;
80  }
```

K64.c

6.8 Fonctions

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fonctions.h"
#include "../DES/Tables_Permu.h"

/*! \file      fonctions.c
 *  \brief     Fichier contenant les différentes fonctions communes aux autres fichiers
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

 /*! \fn  long et_binaire(long a, long b)
 *  \brief Fonction qui permet d'appliquer un ET binaire entre a et b
 *  \param a : long a
 *  \param b : long b
 *  \return (a & b)
 */
long et_binaire(long a, long b)
{
    return (a & b);
}

 /*! \fn  long shift_droit(long a, int b)
 *  \brief Fonction qui permet de shifter à droite de b decalage
 *  \param a : long a
 *  \param b : int b
 *  \return (a >> b)
 */
long shift_droit(long a, int b)
{
    return (a >> b);
}

 /*! \fn  long shift_gauche(long a, int b)
 *  \brief Fonction qui permet de shifter à gauche de b decalage
 *  \param a : long a
 *  \param b : int b
 *  \return (a << b)
 */
long shift_gauche(long a, int b)
{
    return (a << b);
}

 /*! \fn  long concat(long a,long b)
 *  \brief Fonction qui permet de concatener 2 long
 *  \param a : long a
 *  \param b : long b
 *  \return (a | b) la ocncatenation des 2
 */
long concat(long a,long b)
{
    return (a | b);
}

 /*! \fn  long or_exclu(long a,long b)
 *  \brief Fonction qui permet de xor 2 long
 *  \param a : long a
 *  \param b : long b
 *  \return (a ^ b) la concatenation des 2
 */
long or_exclu(long a,long b)
{
    return (a ^ b);
}

```

```
69  /*! \fn long decoupage_msgL(long msg,long mask)
70   *  \brief Fonction qui permet de stocker la partie L du message
71   *  \return L16 fauté ou non selon le paramètre
72   */
73  long decoupage_msgL(long msg,long mask)
74  {
75      return (msg >> 32) & mask;
76  }
77
78  /*! \fn long decoupage_msgR(long msg,long mask)
79   *  \brief Fonction qui permet de stocker la partie R du message
80   *  \return R16 fauté ou non selon le paramètre
81   */
82  long decoupage_msgR(long msg,long mask)
83  {
84      return msg & mask;
85  }
86
87  /*! \fn int calcul_col_SBOX(int bloc6bits)
88   *  \brief Fonction qui permet de calculer les colonnes des SBOX
89   *  \return la valeur de la colonne
90   */
91  int calcul_col_SBOX(int bloc6bits)
92  {
93      return ((bloc6bits & 0x1E) >> 1);
94  }
95
96  /*! \fn int calcul_lignes_SBOX(int bloc6bits)
97   *  \brief Fonction qui permet de calculer les lignes des SBOX
98   *  \return la valeur de la ligne
99   */
100 int calcul_lignes_SBOX(int bloc6bits)
101 {
102     return (2 * ((bloc6bits & 0x20) >> 5) + (bloc6bits & 0x1));
103 }
104
105 /*! \fn long decoupe_4bits(long check,int NUM_SBOX)
106  *  \brief Fonction qui permet de découper le check en blocs de 4bits pour chaque SBOX
107  *  \param check : XOR entre L16 et L16_fauté
108  *  \param NUM_SBOX : numéro courant de la SBOX
109  *  \return les 4bits de sortie pour chaque SBOX
110  */
111 long decoupe_4bits(long check,int NUM_SBOX)
112 {
113     return check & get_4bits[NUM_SBOX];
114 }
115
116 /*! \fn long decoupe_6bits(long expansion,int NUM_SBOX)
117  *  \brief Fonction qui permet de découper l'expansion en blocs de 6bits pour chaque SBOX
118  *  \param expansion : Expansion De R15 ou R15f
119  *  \param NUM_SBOX : numéro courant de la SBOX
120  *  \return les 6bits pour chaque SBOX
121  */
122 long decoupe_6bits(long expansion,int NUM_SBOX)
123 {
124     return expansion & get_6bits[NUM_SBOX];
125 }
126
127 /*! \fn int trouver_SBOX_4bits(int NUM_SBOX)
128  *  \brief Fonction qui permet de trouver la bonne SBOX où se trouve les 4bits de sortie
129  *  \param NUM_SBOX : numéro courant de la SBOX
130  *  \return la position des 4 bits pour la bonne SBOX
131  */
132 int trouver_SBOX_4bits(int NUM_SBOX)
133 {
134     return (7 - NUM_SBOX) * 4;
135 }
136
```

```c
/*! \fn int trouver_SBOX_6bits(int NUM_SBOX)
 *  \brief Fonction qui permet de trouver la bonne SBOX où injecté les 6bits
 *  \param NUM_SBOX : numéro courant de la SBOX
 *  \return la position des 6bits pour la bonne SBOX
 */
int trouver_SBOX_6bits(int NUM_SBOX)
{
    return (7 - NUM_SBOX) * 6;
}

/*! \fn int recup_4bits_fonctionF(int SBOX)
 *  \brief Fonction qui permet de recuperer la sortie 4bits après SBOX
 *  \param SBOX : numéro courant de la SBOX
 *  \return les 4bits de chaque SBOX
 */
int recup_sortieSBOX_fonctionF(int SBOX)
{
    return (32 - 4 - SBOX*4);
}
```

fonctions.c

6.9 Main

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "K16_48/K16_48.h"
#include "K16_56/K16_56.h"
#include "K64/K64.h"
#include "DES/Des.h"
#include "DES/Messages.h"

/*! \file      projet_des.c
 *  \brief     Fichier contenant la fonction principale
 *  \author    ROBIN JORAN
 *  \version   1.00
 *  \date      25 Mars 2019
 */

/*! \fn int main()
 *  \brief Fonction qui exécute le programme
 *  \return retourne 0 une fois terminé
 */
int main()
{
    long verif;
    long K16_48bits;
    long K_64bits;

    printf("\n########################## RECHERCHE K16 48BITS ##########################\n\n");

    K16_48bits = trouverK16_48b(msg_EncJuste,msg_EncFaux);

    printf("\nK16_48bits = %lX \n",K16_48bits);




    K_64bits=trouver_K64(msg_clair,msg_EncJuste,K16_48bits);

    printf("\nK 64bits = %lX\n", K_64bits);

    printf("\n########################## FIN RECHERCHE K 64BITS ##########################\n\n");

```

```
42      printf("\n########################## VERIFICATION K 64 BITS ##########################\n\n");
43
44      verif=DES(msg_clair,K_64bits);
45
46      printf("Message Clair = %lX \n",msg_clair);
47      printf("Message Chiffré Juste = %lX \n",msg_EncJuste);
48      printf("Chiffré trouvé avec DES et K64 = %lX \n",verif);
49
50      return 0;
51  }
```

projet_des.c

6.10 Tables Permutations

```
1   #ifndef Tables_Permu_H
2   #define Tables_Permu_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <string.h>
7
8   /*! \file       Tables_Permu.h
9    *  \brief      Fichier contenant les différentes constantes nécessaires pour les permutations du DES
10   *  \author     ROBIN JORAN
11   *  \version    1.00
12   *  \date       25 Mars 2019
13   */
14
15  static int IP[] =
16  {
17      58,50,42,34,26,18,10,2,
18      60,52,44,36,28,20,12,4,
19      62,54,46,38,30,22,14,6,
20      64,56,48,40,32,24,16,8,
21      57,49,41,33,25,17,9,1,
22      59,51,43,35,27,19,11,3,
23      61,53,45,37,29,21,13,5,
24      63,55,47,39,31,23,15,7
25  };
26
27  static int inverse_IP[] =
28  {
29      40,8,48,16,56,24,64,32,
30      39,7,47,15,55,23,63,31,
31      38,6,46,14,54,22,62,30,
32      37,5,45,13,53,21,61,29,
33      36,4,44,12,52,20,60,28,
34      35,3,43,11,51,19,59,27,
35      34,2,42,10,50,18,58,26,
36      33,1,41,9,49,17,57,25
37  };
38
39  static int P[] =
40  {
41      16,7,20,21,
42      29,12,28,17,
43      1,15,23,26,
44      5,18,31,10,
45      2,8,24,14,
46      32,27,3,9,
47      19,13,30,6,
48      22,11,4,25
49  };
50
```

```
51  static int inverse_P[] =
52  {
53      9,17,23,31,
54      13,28,2,18,
55      24,16,30,6,
56      26,20,10,1,
57      8,14,25,3,
58      4,29,11,19,
59      32,12,22,7,
60      5,27,15,21
61  };
62
63  static int E[] =
64  {
65      32,1,2,3,4,5,
66      4,5,6,7,8,9,
67      8,9,10,11,12,13,
68      12,13,14,15,16,17,
69      16,17,18,19,20,21,
70      20,21,22,23,24,25,
71      24,25,26,27,28,29,
72      28,29,30,31,32,1
73  };
74
75  static int Sbox[8][4][16] =
76  {{//SB1
77      {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},
78      {0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
79      {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},
80      {15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
81  },
82
83  {//SB2
84      {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10},
85      {3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5},
86      {0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15},
87      {13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9}
88  },
89
90  {//SB3
91      {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
92      {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
93      {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
94      {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
95  },
96
97  {//SB4
98      {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
99      {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
100     {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
101     {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
102 },
103
104 {//SB5
105     {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
106     {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
107     {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
108     {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
109 },
110
111 {//SB6
112     {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
113     {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
114     {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
115     {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
116 },
117
118 {//SB7
```

```
    {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}
},


{//SB8
    {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}
}};

static int 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
};

static int inverse_PC1[] =
{
    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
};

static int 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
};

//Les bits 9,18,22,25,35,38,43,54 sont perdus donc on laisse 0
static int inverse_PC2[] =
{
    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
};

```

```
187 static long get_4bits[] =
188 {
189     0xF0000000,
190     0x0F000000,
191     0x00F00000,
192     0x000F0000,
193     0x0000F000,
194     0x00000F00,
195     0x000000F0,
196     0x0000000F
197 };
198
199 static long get_6bits[] =
200 {
201     0xFC0000000000,
202     0x03F000000000,
203     0x000FC0000000,
204     0x00003F000000,
205     0x000000FC0000,
206     0x00000003F000,
207     0x000000000FC0,
208     0x00000000003F
209 };
210
211 static int pos_bitfaux[8][6] =
212 {
213     {0,31,30,29,28,27},
214     {28,27,26,25,24,23},
215     {24,23,22,21,20,19},
216     {20,19,18,17,16,15},
217     {16,15,14,13,12,11},
218     {12,11,10,9,8,7},
219     {8,7,6,5,4,3},
220     {4,3,2,1,0,31}
221 };
222
223 #endif
```

TablesPermu.h

6.11 Messages

```
1  #ifndef Messages_H
2  #define Messages_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  /*! \file       Messages.h
9   *  \brief      Fichier contenant les messages fournis
10  *  \author     ROBIN JORAN
11  *  \version    1.00
12  *  \date       25 Mars 2019
13  */
14
15     long msg_clair = 0x04E334626E9DC4BC;
16
17     long msg_EncJuste =0xF3C23DEEF7FE5DCB;
18
19     long msg_EncFaux[] ={0xF1D73DEAF7FE5DDF,0xF3D03DEEF7FF5DCB,0xF3C23FAEF7FF5DCB,0xF28239A8E7FF5DCB,
20                          0xF3923DEAE5FE5DCB,0xF2823DEEF7FC5DCB,0xF38239EEE7FE5FCB,0xF28239EEB7FA5DC9,
21                          0xFA8239EFA7EA5DCB,0xF3CA3DEEB7EA5DCB,0xF3C235EFF7EE5DCB,0xF3C22DE7F7BA5DCB,
22                          0xF3C22DEEBFEA5DCA,0xB3C22DEEF7B65DCA,0xF3C22DEEF7BE55CB,0xF3C22DEEF3BE5D82,
23                          0X93C22DEEF3BE5D8B,0xF3E23DEEF3FE5C8B,0xF3C21DEEF3FE4D8B,0XE3C27CCEF3FE4C8B,
```

```
24                              0xF3C27CEED3FE4D8B, 0xE7C27CEEF7DE5DCB, 0xF7C27DEEF7FE7DCB, 0xE3C27CEEF6FE1DEB,
25                              0x63C27DEEF6FE19CB, 0xF3423DFEF6FE5DCB, 0xF3C2BDEEF6FE59CB, 0XF3C63D7EF7FE19DF,
26                              0xF3C73DFE76FE5DDF, 0xF3C73DEEF77E5DCF, 0xF3C63DEEF7FEDDDF, 0XF3D63DAAF7FF5D5B};
27
28  #endif
```

Messages.h