

# Contrôle continu

Malek Zemni

*Attaque par faute sur DES*

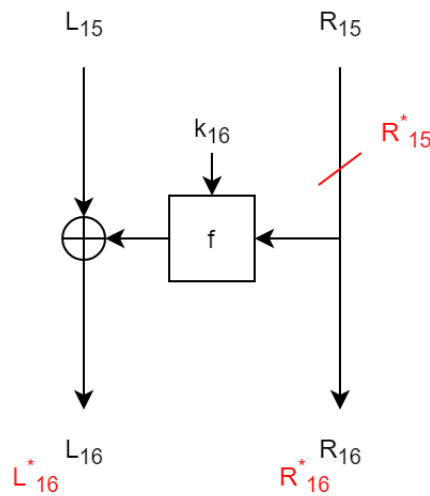
04/04/2018

## Question 1 :

Description d'une attaque par faute contre le DES sur la sortie  $R_{15}$  du 15<sup>ème</sup> tour.

L'attaque par faute contre le DES est une attaque provoquant une faute intentionnelle dans l'algorithme afin de compromettre ses calculs. Cette faute va permettre de révéler une partie de la clé utilisée. Une attaque par recherche exhaustive sur la clé du DES a une complexité de  $2^{56}$ . L'objectif de l'attaque par faute est donc d'accélérer la recherche.

Dans notre cas, la faute est provoquée à la sortie  $R_{15}$  du 15<sup>ème</sup> tour de Feitsel du DES. Il s'agit d'un échange d'un seul bit parmi les 32 bits de  $R_{15}$  (*single bit flip*), ce qui va donner une sortie fautive au 15<sup>ème</sup> tour qu'on note  $R_{15}^*$ . La figure ci-dessous illustre l'injection d'une faute à la sortie 15<sup>ème</sup> tour du DES :



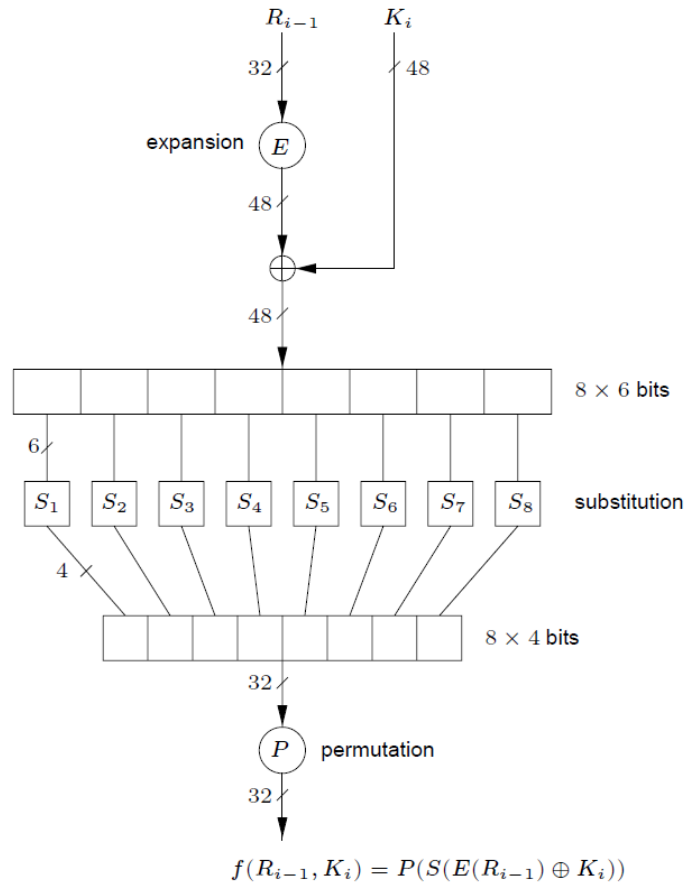
Pour exploiter cette faute, on va analyser les résultats obtenus à la sortie du 16<sup>ème</sup> tour. On a :

- $L_{16} = L_{15} \oplus f(R_{15}, k_{16})$
- $R_{16} = R_{15}$
- $L_{16}^* = L_{15} \oplus f(R_{15}^*, k_{16})$
- $L_{16}^* = R_{15}^*$

On remarque que  $L_{16}$  et  $L_{16}^*$  font toutes les deux intervenir la clé  $k_{16}$  qui est l'objet initial de cette attaque. On exploite donc  $L_{16}$  et  $L_{16}^*$  pour construire une équation permettant de retrouver  $k_{16}$ . On effectue un XOR entre  $L_{16}$  et  $L_{16}^*$  pour éliminer  $L_{15}$ , l'équation obtenue est donc :

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

On va maintenant s'intéresser à la fonction interne  $f$  du DES afin de mieux exploiter l'équation obtenue précédemment. Cette fonction est illustrée par la figure ci-dessous :



L'analyse de cette fonction nous permet d'établir que :

$$\begin{aligned}
 - f(R_{15}, k_{16}) &= P [ S_1(E(R_{15}) \oplus k_{16} \text{ bits } 1 \rightarrow 6) \parallel \dots \parallel S_8(E(R_{15}) \oplus k_{16} \text{ bits } 43 \rightarrow 48) ] \\
 - f(R_{15}^*, k_{16}) &= P [ S_1(E(R_{15}^*) \oplus k_{16} \text{ bits } 1 \rightarrow 6) \parallel \dots \parallel S_8(E(R_{15}^*) \oplus k_{16} \text{ bits } 43 \rightarrow 48) ]
 \end{aligned}$$

En effet, la fonction  $f$  prend en entrée le demi-bloc  $R$  de 32 bits auquel elle applique une expansion de 48 bits, ainsi que la clé  $k_{16}$ . Ces deux entrées sont mélangées à l'aide d'un XOR pour obtenir une entité de 48 bits. Ces 48 bits vont être répartis sur 8 boîtes de substitution appelées *S-box*. Chacune des 8 *S-box* prend 6 bits en entrée et en renvoie 4, pour avoir un résultat final de 32 bits. Ces *S-box* vont être l'objet principal de l'attaque. L'équation précédemment établie devient ainsi :

$$\begin{aligned}
 L_{16} \oplus L_{16}^* &= f(R_{15}, k_{16}) \oplus f(R_{15}^*, k_{16}) \\
 &\Leftrightarrow \\
 L_{16} \oplus L_{16}^* &= \\
 &= P [ S_1(E(R_{15}) \oplus k_{16} \text{ bits } 1 \rightarrow 6) \parallel \dots \parallel S_8(E(R_{15}) \oplus k_{16} \text{ bits } 43 \rightarrow 48) ] \\
 &\oplus \\
 &= P [ S_1(E(R_{15}^*) \oplus k_{16} \text{ bits } 1 \rightarrow 6) \parallel \dots \parallel S_8(E(R_{15}^*) \oplus k_{16} \text{ bits } 43 \rightarrow 48) ]
 \end{aligned}$$

En appliquant l'inverse de la permutation P (calculée à la main en prenant le chemin inverse de la permutation P fournie dans la documentation du DES), et en s'appuyant la propriété d'une permutation P quelconque,  $P(a \oplus b) = P(a) \oplus P(b)$ , on obtient :

$$\begin{aligned}
 & P^{-1}(L_{16} \oplus L_{16}^*) \\
 & = \\
 & S_1(E(R_{15}) \oplus k_{16 \text{ bits } 1 \rightarrow 6}) \oplus S_1(E(R_{15}^*) \oplus k_{16 \text{ bits } 1 \rightarrow 6}) \\
 & \quad || \dots || \\
 & S_8(E(R_{15}) \oplus k_{16 \text{ bits } 43 \rightarrow 48}) \oplus S_8(E(R_{15}^*) \oplus k_{16 \text{ bits } 43 \rightarrow 48})
 \end{aligned}$$

Finalement, en répartissant cette équation sur les 8 S-box, on obtient 8 équations dont les membres font 4 bits et les solutions font 6 bits :

$$\begin{aligned}
 & \text{— } P^{-1}(L_{16} \oplus L_{16}^*)_{\text{bits } 1 \rightarrow 4} = S_1(E(R_{15}) \oplus k_{16})_{\text{bits } 1 \rightarrow 4} \oplus S_1(E(R_{15}^*) \oplus k_{16})_{\text{bits } 1 \rightarrow 4} \\
 & \text{— } \dots \\
 & \text{— } P^{-1}(L_{16} \oplus L_{16}^*)_{\text{bits } 29 \rightarrow 32} = S_8(E(R_{15}) \oplus k_{16})_{\text{bits } 29 \rightarrow 32} \oplus S_8(E(R_{15}^*) \oplus k_{16})_{\text{bits } 29 \rightarrow 32}
 \end{aligned}$$

La seule inconnue dans toutes ces équations est  $k_{16}$ . Pour trouver  $k_{16}$ , il faut faire une recherche exhaustive sur chacune des 8 S-box correspondant à chacune des 8 équations. Chaque recherche sur les S-box va permettre de révéler 6 bits de  $k_{16}$ , pour en avoir au final 48 bits. La complexité de cette attaque est donc de  $8 \times 2^6$ .

## Question 2 :

### 2.1 Recherche de la clé $k_{16}$ de 48 bits

Remarque : les manipulations décrites dans cette partie sont implémentées dans le fichier source DES\_K16.c.

Dans la partie précédente, on est arrivé à établir 8 équations dont chacune révélait les 6 bits de  $k_{16}$  qui rentrent dans chaque S-box. Cependant, il en faudrait plus qu'une simple recherche exhaustive sur chaque S-box, et ceci pour 2 raisons :

#### Problème 1 : propagation du bit faux :

Lors de l'attaque, la faute est introduite sur un seul bit parmi les 32 bits du demi-bloc sortant du 15<sup>ème</sup> tour du DES,  $R_{15}$  qui devient  $R_{15}^*$ . Le problème est qu'un bit erroné n'agit pas sur toutes les 8 S-box que l'on va attaquer puisque les données entrantes aux S-box sont réparties par 6 bits.

En effet, pour une S-box  $i$  et des données de 6 bits allant de  $x$  à  $y$  :

$$P^{-1}(L_{16} \oplus L_{16}^*)_{\text{bits } x \rightarrow y} = S_i(E(R_{15}) \oplus k_{16})_{\text{bits } x \rightarrow y} \oplus S_i(E(R_{15}^*) \oplus k_{16})_{\text{bits } x \rightarrow y}$$

Le problème survient lorsque sur ces 6 bits allant de  $x$  à  $y$ , le bit erroné par l'attaque n'est pas propagé. Dans ce cas on aura  $R_{15} = R_{15}^*$ , ce qui induit aussi que  $L_{16} = L_{16}^*$ . L'équation qui permet la recherche sur cette S-box  $i$  sera de la forme  $0 = 0$  ce qui ne révélerait aucune information sur les 6 bits de  $k_{16}$  allant de  $x$  à  $y$ .

La solution pour remédier à ce problème est d'attaquer le DES par plusieurs fautes dont les bits erronés vont nécessairement se propager jusqu'au 6 bits en entrée de chacune des 8 S-box. En effet, pour assurer cela, il faut vérifier la table d'expansion des bits  $E$  :

$E$					
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

Cette table est utilisée pour effectuer une expansion de 32 bits du demi-bloc pris en entrée, en une entité de 48 bits. Ces 48 bits vont être découpées en 8 blocs de 6 bits. Par exemple une faute sur le 32<sup>ème</sup> bit va être propagée au 1<sup>er</sup> et 47<sup>ème</sup> bit de la sortie de l'expansion, et va ainsi se propager à la S-box 1 et 8. Donc pour une faute, on peut avoir une propagation sur une ou deux S-box au plus.

## Problème 2 : nombre de solutions des S-box :

Une équation du type  $y = S\text{-}box(x)$  peut avoir plusieurs solutions pour  $x$ , à cause de la manière dont une S-box est construite. De ce fait, une recherche exhaustive sur une S-box donnera potentiellement plusieurs solutions pour le 6 bits de  $k_{16}$ . Il est donc impossible de conclure quant aux bons 6 bits de  $k_{16}$ .

La solution pour remédier à ce problème est d'attaquer chaque S-box par plusieurs fautes. Pour chaque faute, on va faire une recherche exhaustive des solutions de la S-box concernée, tout en sauvegardant cet ensemble de solutions. La bonne solution qui sera prise en compte pour les 6 bit de  $k_{16}$  est celle qui est en commun à toutes les fautes injectées.

Il faut bien entendu préciser que les fautes utilisées pour attaquer chaque S-box doivent avoir des bits erronés qui se propagent bien sur les 8 S-box.

## Recherche $k_{16}$ :

Compte tenu des problèmes énoncés, on peut maintenant procéder à la recherche de  $k_{16}$ .

- On commence d'abord par récupérer les 32 chiffrés faux fournis. Après analyse de ces chiffrés, on a pu récupérer pour chaque S-box 6 chiffrés faux dont on s'est assuré que le bit erroné va bien se propager dans la S-box concernée lors de la recherche exhaustive. Pour chacune des 8 S-box, on indique dans une table la position de ces 6 chiffrés faux :

```
1  int faux[8][6] = {
2      {0, 31, 30, 29, 28, 27},
3      {28, 27, 26, 25, 24, 23},
4      {24, 23, 22, 21, 20, 19},
5      {20, 19, 18, 17, 16, 15},
```

```

6      {16, 15, 14, 13, 12, 11},
7      {12, 11, 10, 9, 8, 7},
8      {8, 7, 6, 5, 4, 3},
9      {4, 3, 2, 1, 0, 31}
10 };

```

Listing 1 – DES\_K16.c

— Ensuite, on calcule les éléments vrais de l'équation ( $R_{15}$  et  $L_{16}$ ) :

```

1      //On cherche L16 et R16 à partir d'un chiffre juste
2      dechiffJuste = permutation(chiffreJuste, IP, 64, 64);
3      L16 = (dechiffJuste >> 32) & 0xFFFFFFFFL;
4      R15 = dechiffJuste & 0xFFFFFFFFL;    //car R16 = R15

```

Listing 2 – DES\_K16.c

— On attaque maintenant attaque chacune de 8 S-box par 6 fautes, tout en recalculant à chaque fois les éléments faux de l'équation ( $R_{15}^*$  et  $L_{16}^*$ ) :

```

1      for (int s = 0; s < 8 ; s++)
2      {
3          //Chacune des 8 Sbox va etre attaquée par 6 chiffrés faux
4          for (int f = 0; f < 6 ; f++)
5          {
6              dechiffFaux = permutation(chiffreFaux[faux[s][f]], IP, 64, 64);
7              L16f = (dechiffFaux >> 32) & 0xFFFFFFFFL;
8              R15f = dechiffFaux & 0xFFFFFFFFL;
9
10             //On calcule les éléments de l'équation
11             verif = permutation(L16 ^ L16f, Pinv, 32, 32);
12             E_R15 = permutation(R15, E, 32, 48);
13             E_R15f = permutation(R15f, E, 32, 48);
14
15             //On va tester toutes les valeurs possibles de k16 sur 6 bits :
16             recherche exhaustive de 6 bits
17             for (int k16i = 0 ; k16i < 64 ; k16i++)
18             {
19                 //Valeurs de 6 bits qu'on va rentrer dans la Sbox numéro s
20                 puis vérifier avec les 4 bits correspondants de verif
21                 tmp = ((E_R15 & mask6[s]) >> ((7 - s) * 6)) ^ k16i;
22                 tmpf = ((E_R15f & mask6[s]) >> ((7 - s) * 6)) ^ k16i;
23
24                 //Calcul des lignes et colonnes de la Sbox
25                 r = 2 * ((tmp & 0x20) >> 5) + (tmp & 0x1);
26                 c = (tmp & 0x1E) >> 1;
27
28                 rf = 2 * ((tmpf & 0x20) >> 5) + (tmpf & 0x1);
29                 cf = (tmpf & 0x1E) >> 1;
30
31                 //Vérification de k16i : on compare les 4 bits de verif et
32                 les 4 bits du XOR de Sbox
33                 int PP = (verif & mask4[s]) >> ((7 - s) * 4);

```

```

31         int SS = Sbox[s][r][c] ^ Sbox[s][rf][cf];
32
33         if ( PP == SS )
34         {
35             sol[s][f][nbSol[s][f]] = k16i;
36             ++nbSol[s][f];
37         }
38     }
39 }

```

Listing 3 – DES\_K16.c

- L'attaque va donner un ensemble de solutions possibles pour chacune des 6 fautes. Il faudra finalement récupérer la solution unique qui correspond aux bons 6 bits de  $k_{16}$ , et ceci pour chacune des 8 S-box :

```

1 //Récupération de la solutions unique de chaque Sbox pour les 6
2 fautes et concaténation dans K16
3 int numSolf0 = 0;
4 long candidat = (long) sol[s][0][numSolf0];
5 for (int f = 1; f < 6; f++)
6 {
7     int numSolf;
8     for (numSolf = 0; numSolf < nbSol[s][f]; numSolf++)
9         if (candidat == sol[s][f][numSolf])
10             break;
11
12     if (numSolf == nbSol[s][f]){
13         if (numSolf0+1 >= nbSol[s][0]){
14             printf("\nProblème Sbox %d faute %d pour trouver
15 K16\n", s, f);
16             break;
17         }
18         f = 1;
19         ++numSolf0;
20         candidat = (long) sol[s][0][numSolf0];
21     }
22 }
23 printf("Solution S%d = %lx | ", s+1, candidat);
24 K16 = K16 << 6;
25 K16 = K16 | candidat;
26 printf("K16 actuel = %lx\n", K16);

```

Listing 4 – DES\_K16.c

Voici un aperçu des différentes étapes de l'exécution de cette recherche de  $k_{16}$  :

```

Sbox 4
Faute 1 : 12 solutions 1 3 b 10 17 1a 21 23 2b 30 37 3a
Faute 2 : 12 solutions b d f 1b 1d 1f 20 22 24 30 32 34
Faute 3 : 8 solutions 1 9 30 34 37 38 3c 3f
Faute 4 : 12 solutions b f 13 17 1b 1f 20 24 30 34 38 3c
Faute 5 : 8 solutions 0 2 5 7 20 22 30 32
Faute 6 : 16 solutions 2 3 6 7 e f 1a 1b 24 25 26 27 30 31 36 37
Solution S4 = 30 | K16 actuel = e1d9b0

Sbox 5
Faute 1 : 8 solutions 9 e f 17 29 2e 2f 37
Faute 2 : 4 solutions 22 2f 32 3f
Faute 3 : 8 solutions 0 2 8 a 12 1a 27 2f
Faute 4 : 10 solutions 11 15 1a 1e 22 26 29 2b 2d 2f
Faute 5 : 6 solutions 29 2b 2c 2d 2e 2f
Faute 6 : 12 solutions e f 1a 1b 22 23 2c 2d 2e 2f 3c 3d
Solution S5 = 2f | K16 actuel = 38766c2f

Sbox 6
Faute 1 : 4 solutions 1b 1f 3b 3f
Faute 2 : 6 solutions d f 1d 1f 2c 3c
Faute 3 : 16 solutions 0 1 4 5 8 9 c d 17 1f 30 33 36 38 3b 3e
Faute 4 : 8 solutions 1b 1f 30 32 34 36 3b 3f
Faute 5 : 10 solutions c d e f 1d 1f 39 3b 3c 3e
Faute 6 : 6 solutions 1e 1f 34 35 3c 3d
Solution S6 = 1f | K16 actuel = e1d9b0bdf

Sbox 7
Faute 1 : 8 solutions 5 15 19 1b 25 35 39 3b
Faute 2 : 6 solutions 5 15 25 2e 35 3e
Faute 3 : 8 solutions 5 7 d f 16 1e 36 3e
Faute 4 : 6 solutions 1 5 12 16 32 36
Faute 5 : 14 solutions 5 7 d f 10 12 20 22 28 2a 30 32 38 3a
Faute 6 : 4 solutions 4 5 a b
Solution S7 = 5 | K16 actuel = 38766c2f7c5

Sbox 8
Faute 1 : 14 solutions 3 6 c 15 16 19 1e 23 26 2c 35 36 39 3e
Faute 2 : 8 solutions 9 e f 19 1e 1f 27 37
Faute 3 : 12 solutions 13 16 17 1b 1e 1f 33 36 37 3b 3e 3f
Faute 4 : 8 solutions 1a 1e 29 2d 33 37 3b 3f
Faute 5 : 8 solutions 1c 1d 1e 1f 34 35 36 37
Faute 6 : 16 solutions 4 5 e f 12 13 14 15 16 17 1c 1d 1e 1f 2a 2b
Solution S8 = 1e | K16 actuel = e1d9b0bdf15e

K16 = e1d9b0bdf15e

```

## 2.2 Valeur de la clé $k_{16}$ de 48 bits

E1D9B0BDF15E

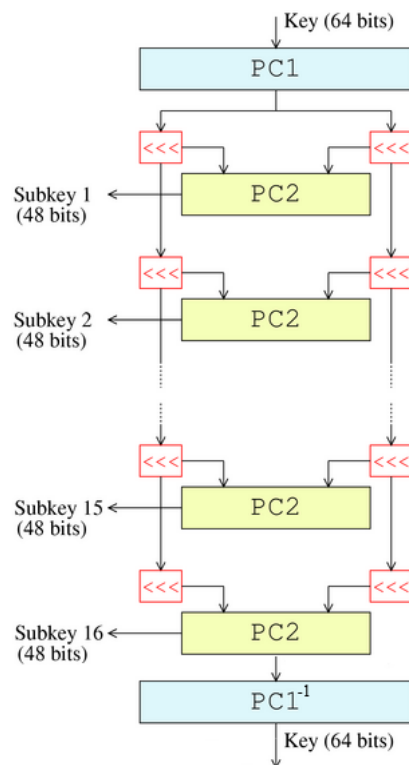
## Question 3 :

Remarque : les manipulations décrites dans cette partie sont implémentées dans le fichier source DES\_K.c.

## 3.1 Recherche des 8 bits manquants

On a obtenu précédemment une clé  $k_{16}$  de 48 bits. On va utiliser cette clé pour reconstituer la clé mère K de 64 bits. Pour cela, on va analyser l'algorithme de dérivation de clés qui est illustré dans la figure ci-dessous :





On remarque qu'en appliquant les inverses des 2 fonctions de permutation PC1 et PC2 à partir de  $k_{16}$ , on retombe sur la clé K de 64 bits. En effet,

$$K = PC1^{-1}( PC2^{-1}( k_{16} ))$$

Cependant, un problème se pose à cause de la nature de la permutation PC2. En effet, PC2 est une permutation qui prend 56 bits en entrée et en renvoie 48. Quand on applique la permutation inverse  $PC2^{-1}$ , on prend 48 bits en entrée et on en renvoie 56. On perd les informations de 8 bits.

Ces 8 bits perdus ont des positions bien précises qu'on trouve par analyse de la permutation appliquer par PC2. Ces positions celles des bits 9, 18, 22, 25, 35, 38, 43 et 54. Lorsqu'on reconstruit la table de la permutation inverse  $PC2^{-1}$ , on met la permutation à 0 pour ces positions.

### Recherche :

La première étape de la recherche des 8 bits perdus était de reconstituer une clé qu'on va appeler  $K^*$  de 64 bits telle que  $K^* = PC1^{-1}( PC2^{-1}( k_{16} ))$ .

Ensuite, une fois qu'on a récupéré les positions bits perdus par  $PC2^{-1}$  (9, 18, 22, 25, 35, 38, 43 et 54), on effectue une recherche exhaustive pour ces 8 bits (complexité au pire de  $2^8$ ). Cette recherche va appliquer un masque différent sur la clé à chaque itération pour alterner les 8 bits aux positions précises sur les 64 bits de  $K^*$  :

```

1  long mask = 0x00L;
2  long Ktest = K48b;
3
4  //On va tester toutes les possibilités pour les valeurs des 8 bits perdus
   dans les positions sauvegardées, donc 256 possibilités

```

```

5  while( mask < 256 && chiffre != DES(clair, Ktest) )
6  {
7      Ktest = K48b | bitsPerdus(mask);
8      mask = mask + 1;
9  }
10 //Si on testé les 256 possibilités pour les 8 bits perdus, on n'arrive
    donc pas à trouver les 56 bits de la clé K
11 if (mask == 256)
12     printf("\nProblème : impossible de trouver K 56 bits\n");

```

Listing 5 – DES\_K.c

```

1  long bitsPerdus(long mask)
2  {
3      long pos[] = {14, 15, 19, 20, 51, 54, 58, 60};
4      long res = 0x0L;
5
6      for(int i = 0; i < 8; i++)
7          res = res | ( ((mask >> i) & 1) << (64 - pos[i]) );
8
9      return res;
10 }

```

Listing 6 – DES\_K.c

### Bits de parité :

La recherche des 8 bits perdus a permis de retrouver 56 bits parmi les 64 bits de la clé mère K. Les 8 bits qui restent sont les bits de parités. Ils n'interviennent pas dans l'opération de chiffrement du DES et n'ont donc pas d'impact lors de la recherche des 8 bits perdus.

Voici un aperçu des différentes étapes de l'exécution de la reconstitution de la clé mère K de 64 bits :

```

Recherche K

48 bons bits de K = 52f8cc169c4a502e
56 bons bits de K = 52f8fc169c4a707e
K 64 bits = 52f8fd169d4a707f
#####
Vérification de la clé trouvée
Chiffré donné = 8fa98ee0e9f5cbaf
Chiffré trouvé = 8fa98ee0e9f5cbaf

```

## 3.2 Valeur de la clé K de 64 bits

52F8FD169D4A707F