Calcul sécurisé - Contrôle continu

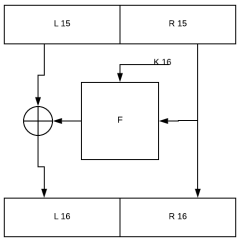
22 mars 2019

Table des matières

1 Question 1

Une attaque par faute contre le DES consiste à introduire une modification sur un bit du chiffré durant l'exécution du chiffrement de manière a compromettre son execution.De ce fait si on dispose d'un chiffré correct et d'un chiffré fauté par l'attaque on peut obtenir des informations sur une partie de la clé utilisée pour le chiffrement. Lors d'une attaque par force brute (recherche exhaustive) sur la clé du DES, la complexité est de 2^{56}. Le but d'une telle attaque est donc de réduire cette complexité.

En supposant que l'attaquant est capable d'effectuer une faute sur la valeur de sortie R_{15} du $15_{ème}$ tour une attaque par faute peut être décrite de la façon suivante :



Lors d'une utilisation normal de DES (sans attaque par faute), on obtiendrais les resultats suivant pour L_{16} et R_{16}.

— $L_{16} = L_{15} \oplus F(R_{15}, K_{16})$
— $R_{16} = R_{15}$

Maintenant, si on introduit une faute sur la valeur de sortie R_{15} du $15_{ème}$ tour, on obtient les valeurs suivantes :

— $R_{15} = R^*_{15}$
— $L_{16} = L^*_{16} = L_{15} \oplus F(R^*_{15}, K_{16})$
— $R_{16} = R^*_{15}$

On a donc une possibilité de retrouver K_{16} en utilisant L_{16} et L^*_{16}.
On va donc utiliser l'opération XOR ($\oplus$) sur L_{16} et L^*_{16} de façon a obtenir :

$$L_{16} \oplus L^*_{16} = L_{15} \oplus F(R_{15}, K_{16}) \oplus L_{15} \oplus F(R^*_{15}, K_{16})$$

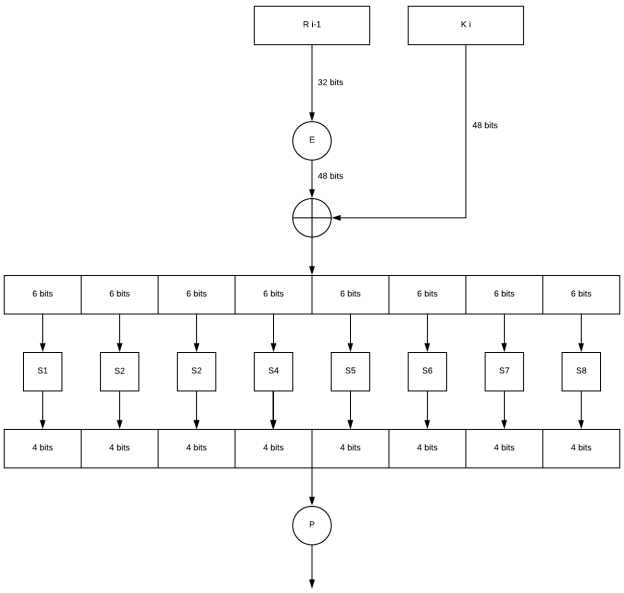
$$= \not{L}_{15} \oplus F(R_{15}, K_{16}) \oplus \not{L}_{15} \oplus F(R^*_{15}, K_{16})$$

On se retrouve donc avec :

$$L_{16} \oplus L^*_{16} = F(R_{15}, K_{16}) \oplus F(R^*_{15}, K_{16})$$

Pour continuer l'attaque on va devoir étudier la fonction F plus en détail soit :

F



On constate que la fonction F prend le bloc R_{i-1} de 32 bits en entrée ainsi que la clé K_i. R_{i-1} passe ensuite par la fonction E qui a pour but d'appliquer une expansion sur le bloc, le passant ainsi de 32 à 48 bits.
Après l'expansion, l'opération XOR est appliquée entre R_{i-1} et K_i.
On a donc 48 bits , qui vont être "découpés" en 8 blocs de 6 bits. Chaque bloc va ensuite passer par une S-Box. Les S-Box prennent en entrée 6 bits et en renvoient 4, ce qui permet de ramener la nombre de bit à 32 (soit la taille initiale de R_{i-1}).
Ce bloc de 32 bits va finalement subir une permutation, la sortie de cette permutation étant le resultat renvoyé par la fonction F.
On peut donc écrire l'équation $L_{16} \oplus L_{16}^* = F(R_{15}, K_{16}) \oplus F(R_{15}^*, K_{16})$ de la façon suivante :

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

On sait également d'après le schéma d'exécution de la fonction F qu'on peut isoler individuellement le résultat de chaque S-box, ce qui permet de transformer cette équation.
$L_{16} \oplus L_{16}^* = F(R_{15}, K_{16}) \oplus F(R_{15}^*, K_{16})$ devient :
$L_{16} \oplus L_{16}^* = P(S_1(E(R_{15}) \oplus K_{16}^{0-5}))||S_2(E(R_{15}) \oplus K_{16}^{6-11}))||...)$

$$\oplus$$

$P(S_1(E(R_{15}^*) \oplus K_{16}^{0-5}))||S_2(E(R_{15}^*) \oplus K_{16}^{6-11}))||...)$
On peut se permettre d'écrire l'équation ainsi car le contenu des différentes S-box est connu,

on peut donc à partir des 6 bits d'entrée trouver les bits de sortie correspondants (par ailleurs on peut également retrouver à partir des 4 bits de sortie d'une S-box plusieurs bloc de 6 bits d'entrée possible).

Maintenant on souhaite se débarrasser de P dans notre équation, pour cela il suffit d'appliquer P^{-1} à $L_{16} \oplus L_{16}^*$ soit (possible car la permutation P est connue) :

$$P^{-1}(L_{16} \oplus L_{16}^*) =$$

$$S_1(E(R_{15}) \oplus K_{16}^{0-5}) \oplus S_1(E(R_{15}^*) \oplus K_{16}^{0-5}) || S_2(E(R_{15}) \oplus K_{16}^{6-11}) \oplus S_2(E(R_{15}^*) \oplus K_{16}^{6-11}) ||$$

...

On peut maintenant mettre en pratique le fait que l'on puisse retrouver les bits d'entrée des 6 box grâce aux bit de sortie pour utiliser cette équation sur les S-box.
On va diviser $P^{-1}(L_{16} \oplus L_{16}^*)$ en 8 blocs de 4 bits. Chaque bloc correspond donc à une sortie de S-box. On aura donc 8 équations :

- $P^{-1}(L_{16} \oplus L_{16}^*)_{0-3} = S_1(E(R_{15}) \oplus K_{16}))_{0-3} \oplus S_1(E(R_{15}^*) \oplus K_{16})_{0-3}$
- $P^{-1}(L_{16} \oplus L_{16}^*)_{4-7} = S_2(E(R_{15}) \oplus K_{16}))_{4-7} \oplus S_2(E(R_{15}^*) \oplus K_{16})_{4-7}$
- ...

K_{16} est la seule valeur inconnue dans ces 8 équations. Chacune de ces équations va nous permettre de retrouver 6 bits de K_{16}, on va donc devoir faire une recherche exhaustive des bits de K_{16} pour chaque S-box afin de retrouver les 48 bits de la sous-clé (6 bits par S-box). Pour cela on va tester toutes les valeurs possibles de K_{16} pour chaque équations.
On fera donc 8 recherches de 6 bits pour une complexité de $6*8*2^6 = 6*2^3*2^6 = 3*2^1*2^9 = 3*2^{10}$.

2 Question 2

2.1 Décrire précisément ce que vous faites pour retrouver la clé

On a précédemment établi 8 équations qui devrait nous permettre de retrouver K_{16}, chaque équation permettant de trouver 6 bits de la clé rentrant dans la S-box correspondante. On va donc maintenant devoir attaquer chaque S-Box pour retrouver les 6 bits de K_{16}. Cependant comme vu précédemment les 4 bits de sorties d'une S-Box peuvent correspondre à plusieurs entrées de 6 bits différents. On ne peut donc pas se contenter d'une seule recherche exhaustive. On va donc procéder de la façon suivante afin de trouver les 6 bon bits de la clé pour chaque S-Box :

- En premier lieu on va chercher pour chaque S-Box quels sont les chiffrés faux correspondants de la façon suivante :

chiffré juste	1E F4 9F 41 6D D5 57 8A	0001 1110 1111 0100 1001 1111 0100 0001 0110 1101 1101 0101 0101 0111 1000 1010
chiffré faux	1C E5 9F 45 6D D5 57 8E	0001 1100 1110 0101 1001 1111 0100 0101 0110 1101 1101 0101 0101 0111 1000 1110
Permutation IP	58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8, 57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3, 61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7	
	L16	R16
chiffré juste permuté	0111 1010 0110 0111 0111 0111 0111 1100	1010 0110 0001 0010 1001 0101 1100 0101
	L16*	R16*
chiffré faux permuté	0111 1010 0110 0101 1111 1111 0111 1110	1010 0110 0001 0010 1001 0101 1100 0100
L16 XOR L16*	0000 0000 0000 0010 1000 1000 0000 0010	
P-1	0101 0000 0000 0000 0000 0000 0000 0011	

— On constate ici au résultat de $P^{-1}(L_{16} \oplus L^*_{16})$ que le 1^{er} et le $8^{ème}$ blocs de 4 bits sont différents de 0. On en déduit donc que ce chiffré peut être utilisé pour l'attaque des S-Box 1 et 8.

— La partie précédente nous à permis de définir 8 équations de cette forme :

$$P^{-1}(L_{16} \oplus L^*_{16})_{0-3} = S_1(E(R_{15}) \oplus K_{16})_{0-3} \oplus S_1(E(R^*_{15}) \oplus K_{16})_{0-3} ...$$

On a vu comment identifier quels chiffrés faux utiliser contre quels S-BOX, on va à partir de la devoir trouver tout les couples de 6 bits possibles $R_{15} \oplus K_{16}$ et $R^*_{15} \oplus K_{16}$ tel que :

$$S(E(R_{15}) \oplus K_{16})_{i-i+3} \oplus S(E(R^*_{15}) \oplus K_{16})_{i-i+3} = P^{-1}(L_{16} \oplus L^*_{16})_{i-i+3}$$

On va ensuite devoir isoler K_{16}.
Pour ce faire on va :

— Récupérer le chiffré juste C. On notera
$L_{16} = IP(C)_{0-31}$ et $R_{15} = R_{16} = IP(C)_{32-62}$

— On récupère un chiffré faux FC utilisable sur la s-box a attaquer. On notera
$L^*_{16} = IP(FC)_{0-31}$ et $R^*_{15} = R^*_{16} = IP(FC)_{32-62}$

— Chacune de ces variables est connue et fait 32 bits.
On va donc prendre en référence $P^{-1}(L_{16} \oplus L^*_{16})$
On va également appliquer une expansion E (présente dans la fonction F vue précédemment) à R_{15} et R^*_{15} tel que $E\_R_{15} = E(R_{15})$ et $E\_R^*_{15} = E(R^*_{15})$ avec E_R_{15} et $E\_R^*_{15}$ de 48 bits.

— On effectue ensuite une recherche exhaustive de K_{16} sur 6 bits.
On note $Ctmp = E\_R_{15} \oplus K_{16}$
et $FCtmp = E\_R^*_{15} \oplus K_{16}$,
chacune de ces 2 variables composée de 6 bits. En effet pour réduire la complexité de la recherche exhaustive de K_{16} on va selectionner uniquement les 6 bits de E_R_{15} et $E\_R^*_{15}$ qui nous intéressent sur la S-Box a attaquer (les 6 premiers si on souhaite attaquer S_1 et ainsi de suite).

— On utilise ces 6 bits sur la S-Box à attaquer avec Ctmp et FCtmp. Puis on appliquer un XOR sur le résultat obtenue avec Ctmp et FCtmp ($S(E(R_{15}) \oplus K_{16})_{i-i+3} \oplus S(E(R^*_{15}) \oplus$

$K_{16})_{i-i+3}$) sur la S-Box.
Si le résultat du XOR correspond à $P^{-1}(L_{16} \oplus L^*_{16})$ alors on saura que les 6 bits de K_{16} utilisé pour créer Ctmp et FCtmp sont une solutions possible.

— Enfin on réitère ce processus avec plusieurs chiffrés différents sur une même S-box (les chiffrés identifiés précédemment comme étant utilisable sur cette S-Box).
On aura donc plusieurs solutions de 6 bits possibles pour K_{16}. Les 6 bon bits a conservé seront ceux communs a chaque attaque effectuée sur la S-Box.

— De cette façon on a réussi a identifier 6 bits de K_{16} grâce à une S-Box. Il nous suffit donc d'attaquer les 7 autres de la même manière pour réussir à obtenir 8 * 6 bits = 48 bits soit K_{16}.

2.2 Donnez les 48 bits de clé que vous obtenez grâce à cette attaque par fautes

On à pu, grâce à l'attaque décrite précédement identifier K_{16} comme étant :

— directement en sortie du programme :

$$b|8|3a|21|8|d|2a$$

— converti en binaire :

$$001011|001000|111110|110110|100001|001000|001101|101010$$

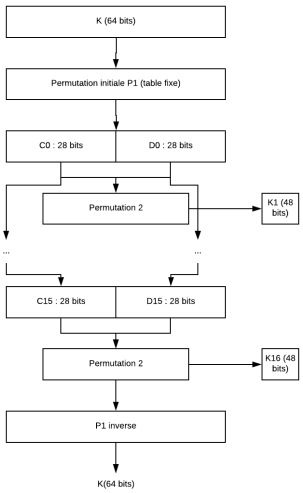
— converti en hexadécimal :

$$2C8FB684836A$$

3 Question 3

3.1 Expliquer comment on peut retrouver les 8 bits manquants

On à retrouvé à la question précédente la sous-clé K_{16}. On va maintenant essayer de retrouver la clé K complète.
Pour cela on devoir analyser le schéma de dérivation des sous-clé K_i à partir de K :



Après analyse de cet algorithme, on constate qu'on peut retrouver K si on à K_{16} car

$$K = P1^{-1}(P2^{-1}(K_{16}))$$

En analysant les données d'entrées de P2, on constate qu'elle prend 56 bits en entrée et en renvoi 48 en sortie.
On va donc avoir un problème pour $P2^{-1}$ qui va prendre en entrée 48 bits pour en renvoyer 56 en sortie, en effet il manque 8 bits.
On va pouvoir cependant retrouver les positions de ces 8 bits en analysant la table de permutation $P2$:

PC-2					
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

En analysant cette table de permutation on peut retrouver les 8 bits manquant qui sont donc :

$$9, 18, 22, 25, 35, 38, 43, 54$$

On va facilement pouvoir reconstituer la permutation inverse $P2^{-1}$ à partir P2 :

$P2^{-1}$					
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				

Les positions identifiées précédemment sont donc mise à zéro dans cette table de permutation car ce sont celles pour lesquelles on n'a pas d'information.
on aura également besoin de $P1^{-1}$ que l'on retrouvera de la même façon que $P2^{-1}$ (à partir de P1 qui est elle aussi connu) :

$P1^{-1}$					
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		

Une fois qu'on à récupéré ces 2 permutations, on va pouvoir construire $K = P1^{-1}(P2^{-1}(K_{16}))$. Cependant ce n'est toujours pas la bonne clé puisque 8 bits sont encore inconnu (les 8 bits mis à 0 par la permutation $P2^{-1}$).
On va donc devoir faire une recherche exhaustive sur ces 8 bits (soit 256 possibilités). On va

simplement prendre K et tester ces possibilités pour les 8 bits manquants.
Après cette recherche on aura donc une clé K de 56 bits. Les 8 bits restants étant des bits de parités,il n'est pas important de les retrouver avant de chercher les 8 bits précédents.
On pourra les retrouver en découpant K en 8 bloc de 7 bits. Pour chaque bloc on va rajouter un bit de parité, de façon à ce que chaque bloc de 8 bits ai un nombre impair de 1.

3.2 Faites-le, et donner ainsi la valeur complète de la clé qui vous a été assignée.

Pour retrouver K, on travaille donc avec $P1^{-1}(P2^{-1}(K_{16}))$.

- $K_{16} = 001011|001000|111110|110110|100001|001000|001101|101010$
- On retrouve facilement $P2^{-1}(K_{16})$:
  $100110100^*011101100^*1100^*100^*1001000000^*000^*10100^*00100011010^*11$
  * : bits perdus en passant par la table de permutations, ce sont les bits qu'on va devoir retrouver pour reconstituer K.
- On retrouve également $P1^{-1}(P2^{-1}(K_{16})$ tout en prenant soin de noter ou sont permutés les bits marqués * :
  $01010000^{\check{r}}111110^*0^*0^{\check{r}}000^*0^*1000^{\check{r}}11010100^{\check{r}}11100000^{\check{r}}01100100^{\check{r}}000^*110^*00^{\check{r}}10^*00^*0010^{\check{r}}$
  * : bits perdus a retrouver
  $^{\check{r}}$: bits de parités

- On remarque que certains bits de parités ne seront pas affectés par les changements sur les bits *, on peut donc déjà écrire :

  $K_{provisoire} = 01010001^{\check{r}}111110^*0^*0^{\check{r}}000^*0^*1000^{\check{r}}11010101^{\check{r}}11100000^{\check{r}}01100100^{\check{r}}000^*110^*00^{\check{r}}10^*00^*0010^{\check{r}}$

On a ci-dessus toutes les donnes que l'on va devoir utiliser pour retrouver la clé K utilisé pour ce chiffrement. On teste donc toutes les possibilités (soit 2^8). On vérifie* ensuite chaque résultat en chiffrant le message clair avec la clé. trouvée. Si on retrouve le même chiffré juste que celui fourni dans l'énoncé, alors on aura retrouvé K :

$$01010001111110000001100011010101111000000110010000111100110000 10$$

soit $51F818D5E0643CC2$
* : vérification des clés avec le lien fourni dans l'énoncé.

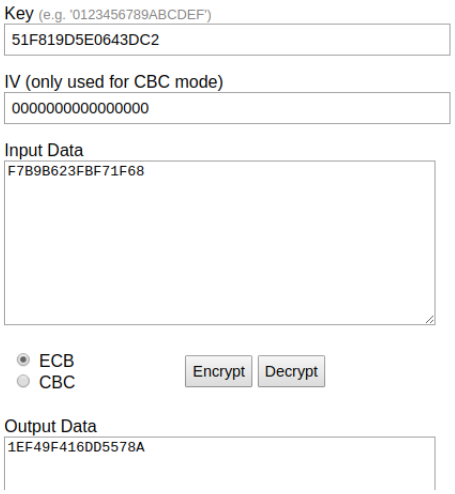
Cependant il manque encore les bits de parité à modifier(ces bits n'intervenant pas dans le calcul du chiffrement, on peu verifier la clé K sans les avoir modifier au préalable). On obtient donc K :

$$01010001111110000001100111010101111000000110010000111101110000 10$$

soit K =

$$51F819D5E0643DC2$$

- Message clair : $F7B9B623FBF71F68$
- Chiffré juste : $1EF49F416DD5578A$



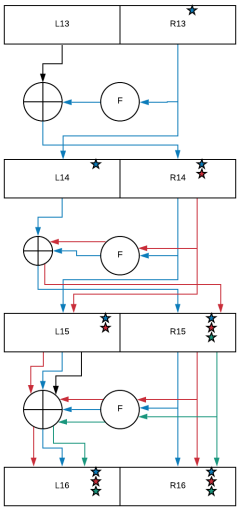
4 Question 4

Pour retrouver la clé K grâce à l'injection d'une faute sur la sortie R_{15} du $15^{ème}$ tour il à fallu :

- retrouver K_{16} : $6 * 8 * 2^6 = 6 * 2^3 * 2^6 = 3 * 2^1 * 2^9 = 3 * 2^{10}$ (complexité de la recherche exhaustive sur les 8 S-BOX).
- retrouver K : 2^8.

On a donc une complexité $3 * 2^{10} + 2^8$ que l'on peut approximer comme $O(2^{10})$. Mais que ce passerait-il si la faute était injectée avant le $15^{ème}$ tour ?

...



Voici un aperçu du cheminement d'une faute si l'attaque avait été réalisé sur la sortie R_{14} du $14^{ème}$ tour (en rouge) ou encore celle R_{13} du $13^{ème}$ tour (en bleu).
On a également en vert l'attaque que nous avons réalisé.

4.1 attaque sur R_{14} du $14^{ème}$ tour

On va s'intéresser à une attaque R_{14} le $14^{ème}$ tour.
On aura donc les équations suivantes :

— $L^*_{15} = R^*_{14} = R_{14} \oplus e$
— $R^*_{15} = L_{14} \oplus F(R^*_{14} \oplus K_{15}) = R_{15} \oplus e'$
— $L^*_{16} = L^*_{15} \oplus F(R^*_{15} \oplus K_{16})$
$= (R_{14} \oplus e) \oplus F(R^*_{15} \oplus K_{16})$

Essayons de retrouver la même équation que pour l'attaque sur le $15^{ème}$ tour :
$L_{16} \oplus L^*_{16} =$
$L_{15} \oplus F(R_{15} \oplus K_{16}) \oplus L^*_{15} \oplus F(R^*_{15} \oplus K_{16})$
$= R_{14} \oplus F(R_{15} \oplus K_{16}) \oplus R_{14} \oplus e \oplus F(R^*_{15} \oplus K_{16})$
On peut retrouver e de la façon suivante :
$e = R_{14} \oplus R^*_{14} = L_{15} \oplus L^*_{15}$ Avec cette valeur on peut donc obtenir l'équation suivante :
$L_{16} \oplus L^*_{16} \oplus e =$
$\cancel{R_{14}} \oplus F(R_{15} \oplus K_{16}) \oplus \cancel{R_{14}} \oplus \cancel{e} \oplus F(R^*_{15} \oplus K_{16}) \oplus \cancel{e}$

On retrouve ici la même équation que pour une attaque sur le dernier tour de DES. On peut donc l'exploiter de la même façon pour retrouver la clé K. Il faut cependant réussir a identifier R_{14} et R^*_{14}. La différence de complexités est donc celle de la recherche de ces 2 inconnus.
En supposant que l'attaquant connaisse R_{14}, il devra faire une recherche sur R^*_{14} soit 2^5 R^*_{14} possibles. Ensuite pour chaque possibilité faire une recherche exhaustive sur K_{15}, sur le même principe que celle que nous avons effectué sur R_{15}. On va donc cherché pour chaque possibilité 4 bits sur les 8 S-BOX de la fonction F soit un total de $2^5 * 8 * 4 * 2^4 = 2^{14}$. On aura donc une complexité pour retrouver K_{16} de $2^{14} * 3 * 2^{10} = O(2^{24})$.
2^5 représente les possibilités de R^*_{14}, 8 les 8 S-box, 4 les 4 entrées possibles pour les S-Box et 2^4 les possibilités de sortie de la S-BOX.

4.2 attaque sur R_{13} du $13^{ème}$ tour

On va s'intéresser à une attaque R_{13} le $13^{ème}$ tour.
On peut appliquer le même raisonnement que pour l'attaque sur le $14^{ème}$, sauf que cette fois si il faudra également retrouver R^*_{13}. Donc on aura une complexité de la forme $2^{14} * 2^{14} * 3 * 2^{10} = O(2^{38})$

4.3 au delà

On constate que plus on remonte dans les tours, plus les valeurs dont on aura besoin pour retrouver K seront compliquées à obtenir. De ce fait cette attaque ne sera pas réaliste sur tout les tour de DES. Si une attaque à lieu sur le $10^{ème}$ tour elle aura une complexité en $O(2^{80})$ et ne sera donc pas réalisable en temps raisonnable.

5 Question 5

Pour pouvoir contrer une attaque par faute, on peux imaginer certaines contre-mesures.

5.1 contre-mesure par répétition

On peut par exemple exécuter les calculs 2 fois. En émettant l'hypothèse que la faute induite n'affecte qu'une exécution, on peut alors effectuer le même calcul après un certain intervalle de temps. On compare ensuite le résultat de la première exécution avec celui de la seconde. Si les résultat sont les mêmes, on peu considérer le chiffré comme étant bon. S'ils sont différents alors dans ce cas on peu en déduire qu'une faute à été induite.
Ce procédé est cela dit coûteux car il nécessite d'effectuer plusieurs fois le chiffrement. Si on ne décide de faire qu'un vérification, on ira au mieux 2 fois plus lentement que sans contre-mesures.

5.2 contre-mesure matérielle

On peut également effectuer le calcul sur 2 unités différentes et comparer le résultat. En émettant l'hypothèse que la faute induite n'affecte qu'un unité de calcul, on peut alors comparer les résultats en sortie pour détecter s'il y a eu une erreur ou non.De plus on peu paralléliser les calculs ce qui n'augmente donc pas le temps d'exécution.

5.3 autre

Enfin on peut décider d'inclure dans le code des fonctions de test pour vérifier à chaque calcul si le résultat trouvé correspond au résultat attendu. Cette méthode aurait un impact sur le temps de calcul en fonction du nombre de tests effectués durant le chiffrement.

6 Annexe

6.1 SBoxFinder

```
//Fonction permettant d'identifer quels messages utiliser pour attaquer les
S-BOX
void SBOX FINDER (int* FC,int* res){
        IPpermut(FC);
        int tmpC[64];
        for (int i = 0; i < 64; i ++){
            tmpC[i] = C[i];
        }
        IPpermut(tmpC);
        int p_FC32[32], p_C32[32];
        for (int i = 0; i < 32; i ++){
                p_FC32[i] = FC[i];
                p_C32[i] = tmpC[i];
        }
        for (int i = 0; i < 32; i++){
                res[i] = (p_FC32[i] + p_C32[i]) % 2 ;
        }
        PINVpermut(res);
```

```
        //division de res en 8 tableau de 4 bits
        //chaque tableau correspond a une S-BOX
        int S[8][4];
        for (int i = 0; i < 32; i ++){
            if (i < 4)
                S[0][i%4] = res[i];
            else if (i < 8)
                S[1][i%4] = res[i];
            else if (i < 12)
                S[2][i%4] = res[i];
            else if (i < 16)
                S[3][i%4] = res[i];
            else if (i < 20)
                S[4][i%4] = res[i];
            else if (i < 24)
                S[5][i%4] = res[i];
            else if (i < 28)
                S[6][i%4] = res[i];
            else if (i < 32)
                S[7][i%4] = res[i];
        }
        for (int i = 0; i < 8; i ++){
            if (S[i][0] + S[i][1] + S[i][2] + S[i][3] > 0){
                printf("\t SBOX %d ", i+1);
            }
        }
}
```

6.2 subKey16Finder

```
//Fonction permettant de retrouver la sous clé K16
void SubKey16finder (){
    //variables
    int CL16[32], CR16[32], FCL16[32], FCR15[32], REFERENCE[32] /* P-1(L16 XOR L16*) */;
    int E_CR15[48], E_FCR15[48];
    int tmpREF[4];
    int tmpC[6], tmpFC[6]; //variables à utiliser pour les S-BOX
    int SBOX_res[8][6][64] = {{{0}}}; //resultat des attaques sur les S-BOX
    int nbSolviables[8][6] = {{0}}; //nombre de solutions viables par S-BOX
    int extTmpC[2], extTmpFC[2], intTmpC[4], intTmpFC[4];
    //utilisé pour chaques S-BOX
    L16_R16_assignation(CL16,CR16);
    for (int i = 0; i < 8 ; i++){
        printf("############  SBOX %d #########\n", i +1);
        for (int j = 0; j < 6; j ++){
        //ASSIGNATIONS
            int usedFC = SBOX_opponent[i][j]; //FC a utiliser
            int tmpFC64[64];
            FC_FCL16_FCR15_assignation(tmpFC64, FCL16, FCR15, usedFC); //assignation du FC
            REFERENCE_assignation(REFERENCE, CL16, FCL16); //assignation de REFERENCE = P-1(CL16 XOR FCL16)
        //assignation E_CR15 et E_FCR15
            Expansion(E_CR15, CR16);
            Expansion(E_FCR15, FCR15);
```

```
//recherche exhaustive de K16
    for (int k = 0; k < 64; k ++){
//assignation K16tmp, tmpC, tmpFC
        for (int l = 0; l < 6; l++){
            tmpC[l] = (E_CR15[i*6+l] + K16[k][l]) % 2;
            tmpFC[l] = (E_FCR15[i*6+l] + K16[k][l]) % 2;
        }
        //passage dans la S-box
        //on résupère d'abords les bits externes de tmpC et tmpFC et leurs bits internes
        extTmpC[0] = tmpC[0]; extTmpC[1] = tmpC[5];
        extTmpFC[0] = tmpFC[0]; extTmpFC[1] = tmpFC[5];
        for (int l = 1; l < 5; l++){
            intTmpC[l-1] = tmpC[l];
            intTmpFC[l-1] = tmpFC[l];
        }
        //calcul des lignes et colonnes de la S-BOX
        int rowC = TabToDec(extTmpC, 2), columnC = TabToDec(intTmpC, 4);
        int rowFC = TabToDec(extTmpFC, 2), columnFC = TabToDec(intTmpFC, 4);
        int tmpsol = Sbox[i][rowC][columnC] ^ Sbox[i][rowFC][columnFC];
        RefIsolation(tmpREF, REFERENCE, i);
        int ref = TabToDec(tmpREF, 4);
        int k16 = TabToDec(K16[k], 6);
        if (tmpsol == ref){
            SBOX_res[i][j][nbSolviables[i][j]] = k16;
            nbSolviables[i][j] ++;
        }
    }
```

6.3 keyFinder

```
//Fonction permettant de trouver les 256 K possibles a partir de la sous clé K16
void KeyFinder(){
    int  tmpK[64] = {0,1,0,1,0,0,0,1,
                     1,1,1,1,1,0,0,0,
                     0,0,0,0,1,0,0,0,
                     1,1,0,1,0,1,0,1,
                     1,1,1,0,0,0,0,0,
                     0,1,1,0,0,1,0,0,
                     0,0,0,1,1,0,0,0,
                     1,0,0,0,0,0,1,0};
    int LostEightPos[8] = {13, 14, 18, 19, 50, 53, 57, 59};
    for(int i = 0; i < 256; i ++){
        for(int j = 0; j < 8; j ++){
            tmpK[LostEightPos[j]] = theLostEight[i][j];
        }
        printf("*");

        for (int j = 0; j < 64; ++j)
        {
            printf("%d", tmpK[j]);
        }
        printf("\n");
    }
}
```

6.4 autres fonction

```
//Fonction appliquant une Expansion au tableau x passé en argument à partir des valeurs du tableau y
void Expansion(int* x, int* y){
    for (int i = 0; i < 48; i ++){
        x[i] = y[E[i]];
    }
}
```

```
//Fonction appliquant la permutation IP au tableau x passé en argument
void IPpermut(int* x){
    int tmp;
    int res[64];
    for (int i = 0; i < 64; i++){
        tmp = ip[i];
        res[i] = x[tmp-1];
    }
    for (int i = 0; i < 64; i ++){
        x[i] = res[i];
    }
}
```

```
//Fonction appliquant la permutation IP-1 au tableau x passé en argument
void PINVpermut(int* x){
    int tmp;
    int res[32];
    for (int i = 0; i < 32; i++){
        tmp = pinv[i];
        res[i] = x[tmp-1];
    }
    for (int i = 0; i < 32; i ++){
        x[i] = res[i];
    }
}
```

```c
//Fonction utilisée pour retrouver la clé K, associe au tableau x la valeur L16 de FC et celle R15 au tableau y
void FC_FCL16_FCR15_assignation(int* x, int* y, int* z, int i){
    for (int k = 0; k < 64; k++){
        x[k] = FC[i][k];
    }
    IPpermut(x);
    for (int k = 0; k < 64; k ++){
        if (k < 32)
            y[k] = x[k]; //FCL16 = 32 premiers bits de FC
        if (k > 31)
            z[k-32] = x[k]; //FCR15 = 32 derniers bit de C
    }
}
```

```c
//Fonction utilisée pour retrouver la clé K, associe au tableau x la valeur L16 de C et celle R16 au tableau y
void L16_R16_assignation (int* x, int *y){
    int tmpC[64];
    for (int i = 0; i < 64; i ++){
        tmpC[i] = C[i];
    }
    IPpermut(tmpC);
    for (int i = 0; i < 64; i ++){
        if (i < 32)
            x[i] = tmpC[i]; //L16 = 32 premiers bits de C
        if (i > 31)
            y[i-32] = tmpC[i]; //R16 = 32 derniers bits de C
    }
}
```

```c
//Fonction utilisée pour retrouver la clé K, sauvegarde dans le tableau x la valeur P-1(y xor z)
void REFERENCE_assignation(int* x, int* y, int* z){
    for (int k = 0; k < 32; k ++){
        x[k] = (y[k] + z[k]) % 2;
    }
    PINVpermut(x);
}
```

```c
//Fonction utilisée pour retrouver la clé K, isole les 4 bits de References a verifier
void RefIsolation(int* x, int* y, int i){
    for(int j = 0; j < 4; j ++){
        x[j] = y[i*4+j];
    }
}
```

```c
//Convertie le tableau tab en int
int TabToDec(int* tab, int taille){
    int val = 0;
    int tmp = 1;
    for(int i = taille - 1; i >= 0; i --){
        if(tab[i] == 1){
            val += tmp;
        }
        tmp = tmp * 2;
    }
    return val;
}
```

6.5 données

```
#ifndef DATA
#define DATA
//possibilités pour les 8 bits perdus
int theLostEight[256][8] = {...
};
//possibilités pour les 6 bits de K16
int K16[64][6] = {...
};
int Sbox[8][4][16] = {...
};
//table de permutation TP
int ip[] = {...
    };
//table de permutaion P-1
int pinv[] = {...
    };
//table d'expenssion
int E[] = {...
    };
//chiffré juste
    int C[] = { 0,0,0,1,1,1,1,0,...
//ensemble des chiffrés faux
    int FC[32][64] = {...
    };
//chiffrés faux à utiliser contre les SBOX identifié grâce à la fonction SBOX_FINDER
    int SBOX_opponent[8][6] = {...
    };
#endif
```