L'étude précise d'une attaque par fautes sur l'algorithme DES

Mohammed Seghir Said
21 60 68 79.

lundi 24 avril 2020.

Table des matières

1 Déscription de l'attaque par fautes sur le DES

1.1 Introduction sur l'attaque

L'attaque par fautes contre le DES permet d'obtenir la clé de chiffrement d'un message chiffré. L'idée de l'attaque par faute sur le DES consiste à perturber le comportement du circuit afin de modifier l'exécution correcte du chiffrement. On peut utiliser d'iférents moyens pour perturber le circuit comme : *la perturbation de l'alimentation*, *le laser*, *des impulsions lumineuses*,*champs magnétiques*.

En pratique on suppose que l'attaquant dispose d'une implémentation de DES, d'un message clair, d'un message chiffré, ainsi qu'un ensemble de messages chiffrés faux obtenus grâce à un single bit flip sur R_{15} du 15ème tour de fiestel , cela veut dire que l'attaquant doit changer un seul bit parmi les 32 bits de R_{15}.

1.2 Rappel sur le DES

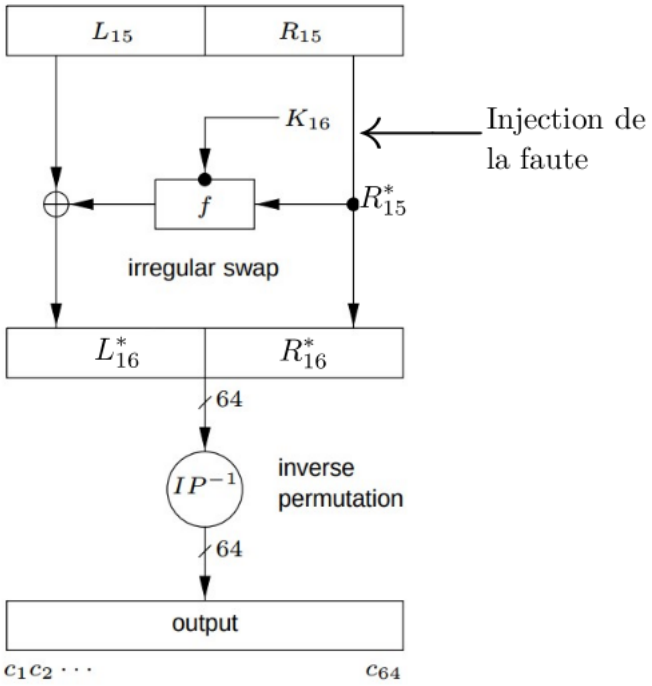

FIGURE 1 – Schéma DES

Comme indiqué dans le schémas figure 1, l'injection d'une faute (single bit flip) au niveau de R_{15} va induire une faute sur R_{16} et L_{16}, qu'on notera R^*_{16} et L^*_{16}.

Le schéma (figure 1) classique sans faute permet de tirer les équations suivantes :

$$\begin{cases} L_{16} &= L_{15} \bigoplus F(R_{15}, K_{16}) \\ R_{16} &= R_{15} \end{cases}$$

En appliquant la faute sur le 15 ème tour on obtient :

$$\begin{cases} L^*_{16} &= L_{15} \bigoplus F(R^*_{15}, K_{16}) \\ R^*_{16} &= R^*_{15} \end{cases}$$

1.3 Exploitation de la faute

Initialement, le but de l'attaque sera de retrouver les 48 bits de la clé K 16. On remarque que L_{15} apparaît dans les deux équations, donc on fait un XOR entre L_{16} et L^*_{16} pour l'éliminer.

d'où :

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

Pour récupérer K_{16}, on peut utiliser un algorithme naïf qui sert à faire une recherche exhaustive sur les 48 bits de la clé, mais en examinant de près la fonction f il est possible de réduire considérablement la complexité de l'attaque.

1.4 Regardont de près ce qui se passe dans la fonction f

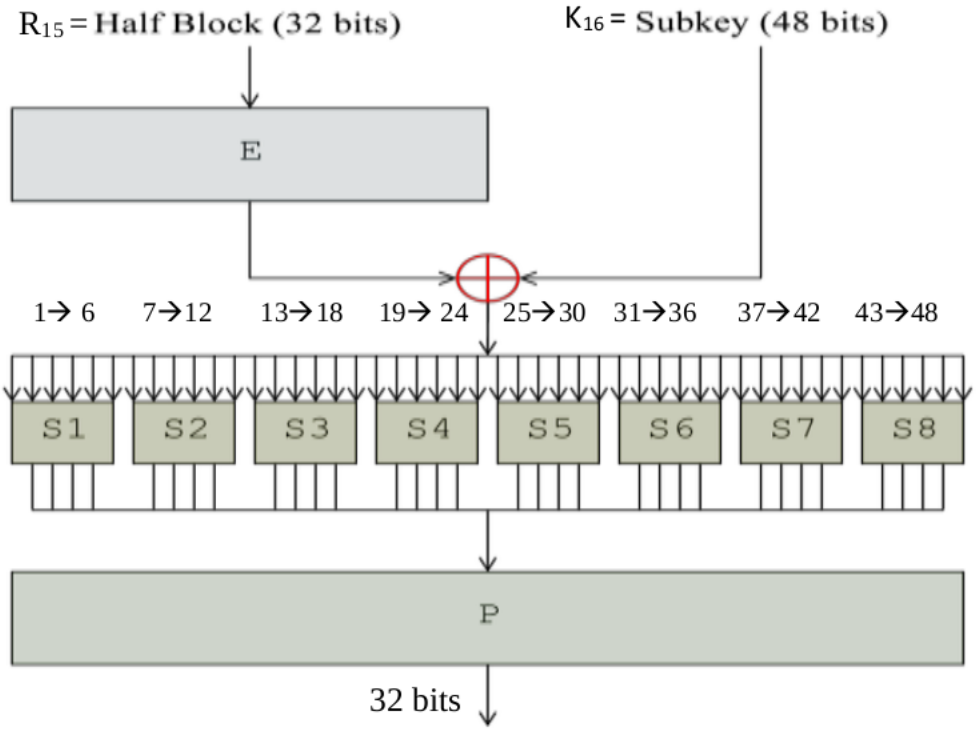

FIGURE 2 – Schéma fonction f

— La fonction f prend en paramètre R_{15} qui fait 32 bits et la clé K_{16} de 48 bits.

— On fait une expantion notée pour R_{15} afin de le faire passer de 32 bits à 48 bits ce qui donne $E(R_{15})$.

— On fait un XOR entre $E(R_{15})$ et K_{16}.

— On obtient un résultat sur 48 bits qui doit passer dans les (S-box) sous forme de 8 paquets de 6 bits ce qui fait 8 boites au total (S1,S2,. . . ..S8), chaque boite prend 6 bits en entrée et renvoie 4 bits en sortie donc on aura un total de 32 bits en sortie.

— Le message de 32 bits en sortie subit une pérmutation P.

Donc d'après le schémas on a les équations suivantes sans faute et avec faute.

$$\begin{cases} F(R_{15},K_{16}) = P(\ S_1([E(R_{15}\oplus K_{16}]_{1\to 6}) \ \| \ S_2([E(R_{15}\oplus K_{16}]_{7\to 12}) \ \| \ \dots\dots \ \| \ S_3([E(R_{15}\oplus K_{16}]_{43\to 48})\ ). \\ F(R^*_{15},K_{16}) = P(\ S_1([E(R^*_{15}\oplus K_{16}]_{1\to 6}) \ \| \ S_2([E(R^*_{15}\oplus K_{16}]_{7\to 12}) \ \| \ \dots\dots \ \| \ S_3([E(R^*_{15}\oplus K_{16}]_{43\to 48})\ ). \end{cases}$$

L'équation peut être simplifiée en appliquant le P^{-1} (la permutation est bijective donc inversible), parce que la boite P fait une permutation entre les bits à la sortie de la S-box, donc va enlever P^{-1} la permutation.

-Donc on obtient :

$$\begin{cases} \mathrm{P^{-1}}(\ \mathrm{F(R_{15},}K_{16})) = S_1([E(R_{15}\oplus K_{16}]_{1\to 6}) \ \| \ S_2([E(R_{15}\oplus K_{16}]_{7\to 12}) \ \| \ \dots\dots \ \| \ S_3([E(R_{15}\oplus K_{16}]_{43\to 48}). \\ \mathrm{P^{-1}}(\ \mathrm{F(R^*_{15},}K_{16})) = S_1([E(R^*_{15}\oplus K_{16}]_{1\to 6}) \ \| \ S_2([E(R^*_{15}\oplus K_{16}]_{7\to 12}) \ \| \ \dots\dots \ \| \ S_3([E(R^*_{15}\oplus K_{16}]_{43\to 48}). \end{cases}$$

On fait un XOR P^{-1} étant linéaire on a : $P^{-1}(a\oplus b)=P^{-1}(a)\oplus P^{-1}(b)$.

-Donc l'équation $(\star)$ devient :

$$P^{-1}(L_{16}\oplus L^*_{16}) = P^{-1}(F(R_{15},K_{16})\oplus F(R^*_{15},K_{16})) = P^{-1}(F(R_{15},K_{16}))\oplus P^{-1}(F(R^*_{15},K_{16})) \iff$$

$$\begin{aligned} P^{-1}(L_{16}\oplus L^*_{16}) = (\ & [S_1([E(R_{15}\oplus K_{16}]_{1\to 6})]\oplus[S_1([E(R^*_{15}\oplus K_{16}]_{1\to 6})] \ \| \ [S_2([E(R_{15}\oplus K_{16}]_{7\to 12})] \ \oplus \\ & [S_2([E(R^*_{15}\oplus K_{16}]_{7\to 12})] \ \| \ \dots\dots\dots \ \| \ [S_3([E(R_{15}\oplus K_{16}]_{43\to 48})]\oplus[S_3([E(R^*_{15}\oplus K_{16}]_{43\to 48})]\ ). \end{aligned}$$

1.5 Résoudre un système d'équations à 1 inconnu

$$\begin{cases} [P^{-1}(L_{16}\oplus L^*_{16})]_{1\to 4} = S_1([E(R_{15})]_{1\to 6}\oplus[K_{16}]_{1\to 6}) \bigoplus S_1([E(R^*_{15})]_{1\to 6}\oplus[K_{16}]_{1\to 6}) \\ [P^{-1}(L_{16}\oplus L^*_{16})]_{5\to 8} = S_1([E(R_{15})]_{7\to 12}\oplus[K_{16}]_{7\to 12}) \bigoplus S_1([E(R^*_{15})]_{7\to 12}\oplus[K_{16}]_{7\to 12}) \\ \vdots \\ \vdots \\ [P^{-1}(L_{16}\oplus L^*_{16})]_{29\to 32} = S_1([E(R_{15})]_{43\to 48}\oplus[K_{16}]_{43\to 48}) \bigoplus S_1([E(R^*_{15})]_{43\to 48}\oplus[K_{16}]_{43\to 48}) \end{cases}$$

On a 8 équations à 1 inconnu, la recherche des 48 bits de la clé K_{16} va consister à faire uniquement une recherche exhaustive sur les blocs de 6 bits des S-box.

Chaque recherche sur les S-box va permettre de révéler 6 bits de K_{16}, ce qui donne une compléxité de 2^6. Au total on a 8 boites S-Box ce qui fait une compléxité de $8\times 2^6 = 2^9$ pour trouver la clé K_{16} .

-Trouver la clé K à partir de K_{16} :

Une fois K_{16} trouvée on aura 16 bits à retrouver dont 8 bits de parité. Pour les 8 bits de clé on fait une recherche exhaustive ce qui donne une complexité de 2^8.

2 Application concrète de l'attaque injection par faute

A l'aide des 32 messages chiffrés faux, on va cibler les S-box à utiliser afin de déterminer la valeur de la clé en effectuant des comparaisons successives.

2.1 Décrire précisément ce que j'ai fait pour retrouver la clé

2.1.1 Introduction au principe de l'attaque

On a le message clair suivant en hexa et en binaire :

$$\begin{cases} (message\ clair)_{15} & = 91\ BF\ 3D\ 7C\ 0B\ 1A\ 1C\ 02. \\ (message\ clair)_2 & = 1001\ 0001\ 1011\ 1111\ 0011\ 1101\ 0111\ 1100\ 0000\ 1011\ 0001\ 1010\ 0001\ 1100\ 0000\ 0010. \end{cases}$$

On a le message chiffré suivant en hexa et en binaire :

$$\begin{cases} (message\ chiffre)_{15} & = & 95\ 3E\ D6\ AA\ D0\ D0\ C1\ F5. \\ (message\ chiffre)_2 & = & 1001\ 0101\ 0011\ 1110\ 1101\ 0110\ 1010\ 1010\ 1101\ 0000\ 1101\ 0000\ 1100\ 0001\ 1111\ 0101. \end{cases}$$

On possède aussi 32 messages chiffrés avec contenant des injections de fautes.

En comparant les différents R_{15}^* (avec faute), on remarquer que l'erreur qui se trouve sur un bit, se propage au niveau de R_{15} sur 1 seul bit à la fois, décale à gauche afin de balayer les 32 bits de R_{15} , d'où les 32 chiffrés faux.

La différence entre le R_{15} juste et quelques R_{15}^* (faux) :

$$R_{15}(juste) = 1100\ 0101\ 0111\ 1011\ 0101\ 1110\ 1011\ 0110.$$

$$\begin{cases} R_{15}^* & = 1100\ 0001\ 0011\ 1001\ 0101\ 1110\ 1111\ 0011. \\ R_{15}^* & = 1100\ 0101\ 0111\ 1011\ 0101\ 1110\ 1011\ 0100. \\ R_{15}^* & = 1100\ 0101\ 0111\ 1011\ 0101\ 1110\ 1011\ 0010. \\ R_{15}^* & = 1100\ 0101\ 0111\ 1011\ 0101\ 1110\ 1011\ 1110. \\ & \vdots \\ & \vdots \\ R_{15}^* & = 0100\ 0101\ 0111\ 1011\ 0101\ 1110\ 1011\ 0110. \end{cases}$$

2.2 Description précise de la méthode.

2.2.1 Cibler les S-box

Comme on vient de voir que l'attaque par faute est injecté seulement sur un seul bit des 32bits de R_{15} ,et R_{15} et R_{15}^* sont connus ,il suffit de XOR les 2 ensembles pour avoir la position du bit fauté. On récupère donc la position du bit faux pour les 32 chiffrés faux et on va ensuite regarder où ce bit est propagé à travers la permutation d'expansion 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

FIGURE 3 – Expansion E

Donc à l'entré de S-Box on va pouvoir cibler exactement la S-Box affectée, et par quel fauté. On peut voir que si le chiffré faux a une faute sur le 32 ème bit, elle va être propagée au 1 er et 47 ème bit de la sortie de E, donc elle va aussi se propager en entrée de la S1 et la S8, alors pour une faute donnée, elle peut se répartir au maximum sur 2 S-Box.

On récupère donc la position des bits fautés et les stoque dans une structure de Type matrice(8×6), de sort à ce que le numéro de ligne correspond au numéro de la S-box, et que le numéro de la colonne correspond au numéro du message chiffré faux.

32	0	31	30	29	28
29	28	27	26	25	24
25	24	23	22	21	20
21	20	19	18	17	16
17	16	15	14	13	12
13	12	11	10	9	8
9	8	7	6	5	4
5	4	3	2	32	0

FIGURE 4 – Matrice des BitsFaux

Par exemple ici le 2ème chiffré faut à une faute sur le 1er bit et elle sera injéctée dans la 1er S-Box et la 8ème S-Box.

2.2.2 La recherche de la clé K_{16} 48 bits

On enlève la permutation IP^{-1} qui est s'effectue en sortie du DES sur le message clair en faisant IP, puis nous allons découper le message obtenu en deux parties, L_{16} et R_{16}, comme on a L_{16}=R_{16}, nous allons stocker R_{16} dans R_{15}, comme on a quel chiffré faux va dans quelle SBOX, on peut faire une attaque par recherche exhaustive de K_{16} de la manière suuivante :

— Dans la boucle de la Recherche exhaustive, pour chaque chiffré faux, on calculera la permutation IP puis découper en 2 parties L^*_{16} et R^*_{16}.

— On va stocker R^*_{16} dans R^*_{15}, car $R^*_{16} = R^*_{15}$.

— On calcule la valeur attendue des 4 bits à chaque sortie d'une S-Box avec la permutation $P^{-1}(L_{16} \oplus L^*_{16})$.

— Ensuite, on attaque le calcul de l'expansion de R_{15}^* et de R_{15}.

— 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 $E(R_{15})$.

— On récupère les valeurs de 4 bits de chaque S-Box avec un XOR entre les S-Box du chiffré juste et celles des chiffrés faux, puis on compare le résultat avec la valeur de vérification sur 4 bits de chaque SBOX.

— Si c'est équivalent alors on stocke la possible solution de K_{16} sur 48 bits dans un tableau.

`Pour un exemple sur une S-box consultez l'annexe de l'exécution voir figure` n `:6.2.`

`Pour voir le code consultez l'annexe code de get_K16 voir figure` n `:6.6.`

On a donc une recharche exhaustive de compléxité de $8 \times 6 \times 2^6$ *,ce qui donne* 3×2^{10}.

2.3 Les 48 bits de K16 obtenus grâce à cette attaque par fautes.

`Pour voir l'affichage du résultat consultez l'annexe du Résultat K16 voir figure` n `:6.1.`

La valeur de K_{16} **:** $(84\ E1\ 3B\ D3\ 76\ D8)_{16}$.

La valeur de K_{16} **:** $(1000\ 0100\ 1110\ 0001\ 0011\ 1011\ 1101\ 0011\ 0111\ 0110\ 1101\ 1000)_2$.

3 Trouver la clé complète du DES

3.1 Trouver les 8 bits manquants

Examinons le shéma suivant de Key Schedule

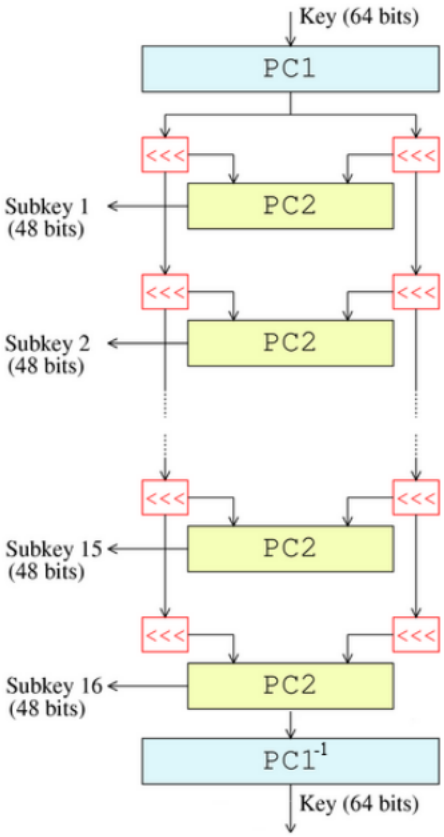

FIGURE 5 – Key Schedule

La permutation PC1 prend la clé key sur 64 bits élimine les 8 bits de parité et fournie en sortie une clé sur 56 bits, la clé subit des rotations de (left shift), Ainsi qu'une permutation PC2 qui donne une subKey sur 48 bits, la clé K16 est donc obtenue après application de plusieurs fois de ce processes (rotations (left shift) et de permutation PC2).

On pu retrouver la clé K 16 de 48 bits dans la question précédente (2.2), il nous faut retrouver les 8 bits manquants pour avoir celle de 56 bits ainsi que les 8 bits de parités restants pour celle de 64 bits. Donc pour avoir la clé K sur 64 bits il faut calculer :

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

3.1.1 Trouver la clé sur 56bits

Cependant lorsque l'on va effectuer la permutation $PC2^{-1}$ inverse, nous allons passer de 48 bits à 56 bits on va avoir 8 bits de perdu, donc nous allons d'abord essayer de déduire la position de ces 8 bits en étudiant la permutation $PC2$, donc on aura une permutation $PC2^{-1}$ qui donne la position des 8 bits, mais pas leur valeur donc on va la mettre à 0 dans un premier temps.

Pour récupérer K_{56} bits, on a implémenté l'algorithme du DES pour pouvoir effectuer une recherche exhaustive sur ces 8 bits perdus, il y a donc 256 possibilités soit 2^8 . On va donc pouvoir tester toutes les positions possibles de ces 8 bits.

Si le message chiffré obtenu correspond avec celui obtenue avec le message clair alors nous avons la bonne clé sur 56bits.

`Pour voir le code consultez l'annexe code de get_K56 voir figure` n `:6.4.`

3.1.2 Trouver la bonne clé sur 64 bits

On a une clé de 56 bits, Il ne nous reste plus qu'à trouver les 8 bits de parités. Les bits de parités n'affectent pas le résultat du DES car le DES utilise une clé de 56 bits. Pour trouver les 8 bits de parité on procède de la façon suivante :

On découpé la clé de 56 bits par blocs de 7, et on calcule la valeur du 8 ème bit de chaque bloc en fonction de la parité du nombre de un dans les blocs de 7 bits, si le nombre de 1 est impair on complète par un 0 sinon par un 1.

`Pour voir le code consultez l'annexe code de get_K voir figure` n `:6.5.`

3.2 La clé K complète obtenue

`Pour voir l'affichage du résultat consultez l'annexe du Résultat K voir figure` n `:6.1.`

La valeur de K **:** $(49\ C7\ FE\ FE\ 8C\ 15\ 83\ 0D)_{16}$.

La valeur de K **:** $(0100\ 1001\ 1100\ 0111\ 1111\ 1110\ 1111\ 1110\ 1000\ 1100\ 0001\ 0101\ 1000\ 0011\ 0000\ 1101)_2$.

4 Attaque DFA (differential fault attack) sur plusieurs tours

L'attaque réalisée ci-dessus montre une attaque DFA à 1 tour. En effet, on a supposé que l'attaquant introduit une erreur sur la valeur du 15 ème tour R_{15}, l'attaque a pu se faire en une complexité 3×2^{10}.

Si on produit une faute sur la valeur de sortie R 14 du 14ème tour on aura :

$$\begin{cases} L_{15} &= L_{14} \bigoplus F(R_{14}, K_{15}) \\ R_{15} &= R_{14} \end{cases} \Longrightarrow \begin{cases} L^*_{15} &= L^*_{14} \bigoplus F(R^*_{14}, K_{15}) \\ R^*_{15} &= R^*_{14} \end{cases}$$

$$\begin{cases} L_{16} &= L_{15} \bigoplus F(R_{15}, K_{16}) \\ R_{16} &= R_{15} \end{cases} \Longrightarrow \begin{cases} L^*_{16} &= L^*_{15} \bigoplus F(R^*_{15}, K_{16}) \\ R^*_{16} &= R^*_{15} \end{cases}$$

On va réutiliser les formules établies pour l'attaque DFA sur L_{15}. Il faut analyser la position des bits faux comme on a fait pour l'attaque du 15 ème tour pour avoir une propagation d'un bit faux jusqu'au 16 ème tour pour chacune des 8 S-Box à attaquer.

La fonction f rend le traçage du bit faux difficile, car la sous-clé à chauqe tour n'est pas connue, jusqu'à ce qu'on arrive au 16 ème tour. Si on fait une recherche de K_{15} afin d'analyser le traçage de la propagation des bits faux, on doit connaître aussi L_{15} et L^*_{15}.

De ce fait la complexité est élevée au carré à chaque fois qu'on essaye de remonter au tour $i-1$.

-Donc :

Pour le 14ème tour la complexité $= 2^{20}$

Pour le 13ème tour la complexité $= 2^{40}$

Pour le 12ème tour la complexité = 2^{80}

La compléxité reste intéréssante jusqu'au 13 ème tour.

5 Contre-mesures

Une attaque par faut a pour principe de générer des fautes dans le circuit d'exécution d'un algorithme. Ces fautes sont généralement réalisées par une modification des conditions environnementales ou la modification des signaux de contrôle (tensions alimentation, champs magnétiques, ...), donc on peut par exemple renforcer le matériel et prendre des mesure à ce que il ne soit pas affecté par ces attaques mais cela peut être très couteux et inadaptable pour la plus part des apareils éléctroniques.

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.

Différentes contre-mesures envisageable sur les attaques par fautes contre le DES :

La redondance temporelle : On peut ré-exécuter le calcul sur le même bloc matériel, puis comparer des différents résultats obtenus. 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 va être multiplié par 2.

Il y a d'autres redondance temporelle : (simple avec opérande inversée,la redondance temporelle simple avec rotation des opérandes . . .).

Détection ou correction par redondance d'information : Cette technique consiste à , ajouter des tests de code qui seront exécutés en même temps que l'algorithme sans nécessiter d'exécution complète supplémentaire. Si une erreur est détectée, on incrémente un compteur, au delà d'une certaine valeur, le système est réinitialisé.

Il rajoutent donc un certain facteur (nombre d'opérations de test) au temps de calcul.

Détection ou correction par redondance matérielle : Cette contre-mesure a pour principe de réaliser la même opération sur plusieurs copies d'un même bloc de calcul et d'en comparer les résultats. Par exemple la duplication simple avec comparaison qui 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.

On peut aussi trouver d'autres duplications comme la duplication dynamique, la duplication hybride ...).

6 Annexes code

6.1 Affichage au terminal de l'attaque et du test.

```
Remonter vers le résultat de K16 ref 2.3
```

Remonter vers le résultat de K ref 3.1.2

```
<------------------------------------------------------------------------>

Recherche de la Cle du chiffrement K :

        Cle K16 trouvée avec succes : 84e13bd376d8.

        La clé K sur 48 bits trouvée avec succes K48 : 48c0cefe8c14820c.

        Clé K trouvée avec succes : 49c7fefe8c15830d.

<------------------------------------------------------------------------>

        Le Message Chiffré donné est : 953ed6aad0d0c1f5.

        Le Chiffré trouvé avec la clé K est : 953ed6aad0d0c1f5.

        Succes test.

<------------------------------------------------------------------------>
```

FIGURE 6 – Terminal

6.2 Exemple d'attaque sur un S-Box

Remonter vers l'attaque K16 ref 2.2.2

```
Resultat de recherche S-Box n°8 :

Le chiffré faux n°5 : 16 solutions      4 7 b c 13 17 18 1c 24 27 2b 2c 33 37 38 3c
Le chiffré faux n°4 : 8 solutions       8 9 f 18 19 1f 21 31
Le chiffré faux n°3 : 6 solutions       10 18 31 32 39 3a
Le chiffré faux n°2 : 10 solutions      2 6 18 1c 2a 2e 30 34 38 3c
Le chiffré faux n°1 : 10 solutions      5 7 18 1a 30 32 35 37 39 3b
Le chiffré faux n°32 : 16 solutions     0 1 2 3 4 5 8 9 a b 12 13 18 19 3c 3d

Solution S8 est : 18            |       K16 evolution : 84e13bd376d8
```

FIGURE 7 – Exemple sur S-Box

6.3 Récupération de la clé K_{48} sur 48 bits.

```c
uint64_t get_K48(uint64_t K16){

    /*
     *On remet les 48 bits de K16 à la bonne position pour K
     *On passe de 48 à 56 bits avec 8 bits faux,On applique l'inverse de PC2.
     *On sauvegarde la position des bits faux.
     *On passe de 56 à 64 bits avec 8 autres bits de parité faux,en appliquant l'inverse de PC1
     *changement de position pour de 8 bits perdus par PC2
     */
    uint64_t K48 = Permutation(Permutation(K16, PC2_Inv, 48, 56), PC1_Inv, 56, 64);
    return K48;

}
```

FIGURE 8 – get_K48

6.4 Récupération de la clé K_{56} sur 56 bits.

Remonter vers l'attaque sur de K56 ref 3.1.1

```c
uint64_t get_K56(uint64_t clair, uint64_t chiffre, uint64_t K16)
{
    /*
     *Attaque par recherche exhaustive sur les 8 bits perdus manquants de K avec à l'aide de DES
     *Recuperer les bits perdus par PC1_Inv(PC2_Inv) dans K48, avec leur position : pos[]
     *Les 8 bits de parite n'interviennent pas dans le DES
    */

    uint64_t K48=get_K48(K16);
    long MASK = 0x00L;
    uint64_t K_test = K48;

    //Tables pour differents positions des bits a charcher
    long pos[] = {14, 15, 19, 20, 51, 54, 58, 60};

    //On va tester toutes les possibilités pour les valeurs des 8 bits perdus dans les positions sauvegardées,
    while( MASK < 256 && chiffre != DES(clair, K_test) )
    {
        //Pour retrouver les bits a 0 qu'on a perdu
        long res = 0x0L;

    for(int i = 0; i < 8; i++)
        res = res | ( ((MASK >> i) & 1) << (64 - pos[i]) );
        K_test = K48 | res;
        MASK = MASK + 1;
    }
    //si on arrive pas a trouver la cle K apre le test de 256 pissibilites
    if (MASK == 256)
        printf("\nimpossible de trouver K 56 bits apres le test des 256 possibilites\n");

    return K_test;
}
```

FIGURE 9 – get_K56

6.5 Récupération de la clé K_{64} sur 64 bits.

Remonter vers l'attaque sur de K ref 3.1.2

```c
uint64_t get_K(uint64_t clair, uint64_t chiffre, uint64_t K16)
{
    //Retrouver les 56 bits de K a partir de K16 i.e les 8 bits manquant
    uint64_t K56 = get_K56(clair, chiffre, K16);
    //Calcul des 8 bits de parite restants
    uint64_t K64 = bitsParite(K56);

    return K64;
}
```

FIGURE 10 – get_K

6.6 Récupération de la clé K_{16} sur 48 bits.

```
//Recherche exhaustive pour trouver la cle K16 sur 56 bits
uint64_t get_K16(uint64_t chiffreJuste, long chiffreFaux[])
{
    int num_Sbox;
    int k16i;
    int f;
    int r, c, rf, cf;

    uint64_t K16,dechiffJuste,dechiffFaux;
    uint32_t L16, R15, L_f_16, R_f_15,verif;
    uint64_t E_R15, ER_f_15;
    uint32_t  tmp, tmpf;
    int num_chiffre_faux;

    //2^6 solutions possibles  des 6 fautes pour les 8 Sbox
    int Solution[8][6][64] = {{{0}}};
    //Nombre de solutions possibles pour 6 chiffres faux sur les 8 Sbox
    int Nbr_Solutions[8][6] = {{0}};

    //Calcule de L16 et R16 a partir d'un chiffre juste
    dechiffJuste = Permutation(chiffreJuste, IP, 64, 64);
    L16 = (dechiffJuste >> 32) & 0xFFFFFFFFL;
    // R16 = R15
    R15 = dechiffJuste & 0xFFFFFFFFL;

    /*Recherche de la cle K16 par une recharche exhaustive a l'aide les 8 equation
     *chaque attaque va nous permettre de connaitre 8 bits de K16 l'aide de la func Calcul_true_K16
    */
    for (num_Sbox = 0; num_Sbox < 8 ; num_Sbox++)
    {

        for (f = 0; f < 6 ; f++)
        {
            //Attaque d'une S-Box par 6 chiffres faux

            dechiffFaux = Permutation(chiffreFaux[BitsFaux[num_Sbox][f]], IP, 64, 64);
            L_f_16 = (dechiffFaux >> 32) & 0xFFFFFFFFL;
            R_f_15 = dechiffFaux & 0xFFFFFFFFL;
```

```
        // calcule des éléments de l'équation
        verif = Permutation(L16 ^ L_f_16, P_Inv, 32, 32);
        E_R15 = Permutation(R15, E, 32, 48);
        ER_f_15 = Permutation(R_f_15, E, 32, 48);

        //recherche exhaustive de K16 sur 6 bits
        for (k16i = 0 ; k16i < 64 ; k16i++)
        {
            //Valeurs de 6 bits qu'on va rentrer dans la Sbox numéro s puis vérifier avec les 4 bits corréspondants de verif
            tmp = ((E_R15 & get_6bits[num_Sbox]) >> ((7 - num_Sbox) * 6)) ^ k16i;
            tmpf = ((ER_f_15 & get_6bits[num_Sbox]) >> ((7 - num_Sbox) * 6)) ^ k16i;

            //Trouver numero de la ligne et de colonne de la S-Box
            r = 2 * ((tmp & 0x20) >> 5) + (tmp & 0x1);
            c = (tmp & 0x1E) >> 1;

            rf = 2 * ((tmpf & 0x20) >> 5) + (tmpf & 0x1);
            cf = (tmpf & 0x1E) >> 1;

            //On va Verfifier la k16i en on comparant les 4 bits de verifvcation avec les 4 bits du XOR de S-Box
            int PP = (verif & get_4bits[num_Sbox]) >> ((7 - num_Sbox) * 4);
            int SS = Sbox[num_Sbox][r][c] ^ Sbox[num_Sbox][rf][cf];

            if ( PP ==  SS )
            {
                Solution[num_Sbox][f][Nbr_Solutions[num_Sbox][f]] = k16i;
                ++Nbr_Solutions[num_Sbox][f];
            }
        }
    }

    //Affichage des solutions pour chaque faute de chaque Sbox
    affiche_Solutin_SBox(Nbr_Solutions,Solution,num_Sbox,chiffreFaux,BitsFaux);
    //Pour trouver la bonne cle K16
    K16=Calcul_true_K16(Solution, Nbr_Solutions,num_Sbox,K16);
}
    return K16;
```

FIGURE 11 – get_K16