Rapport :

Calcul sécurisé – Contrôle continu
Université Paris-Saclay – M1 Informatique / M1 MINT
26 février 2019

Attaque par fautes sur l'algorithme DES

***par* QEHOUI HAJARE**

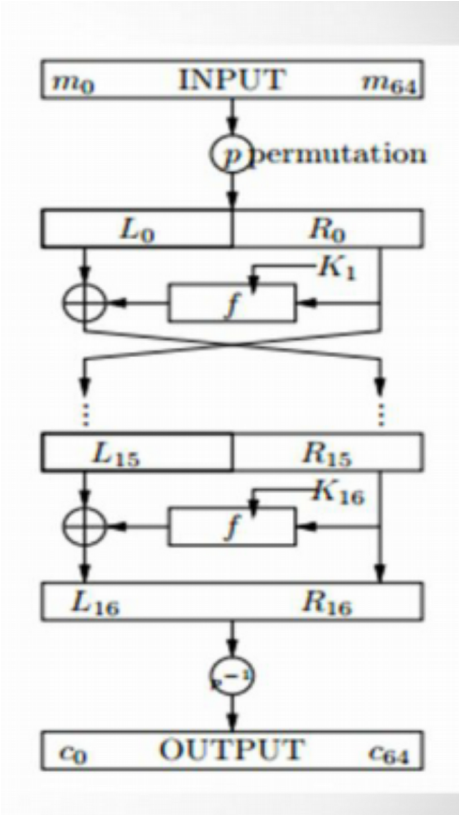
Question 1 (Attaque par faute sur le DES)

Décrire le plus précisément que vous pouvez le principe d'une attaque par fautes contre le DES. On supposera que l'attaquant est capable d'effectuer une faute sur la valeur de sortie R_{15} du 15me tour.

Dans cette partie, on suppose que l'attaquant dispose d'une implémentation de DES, d'un message clair, d'un message chiffré et d'un ensemble de messages faux obtenu grâce à un single bit flip sur R_{15}.

-Rappel sur le DES :

Schema de feistel



$L_i = R_{i-1}$ (32 bits)

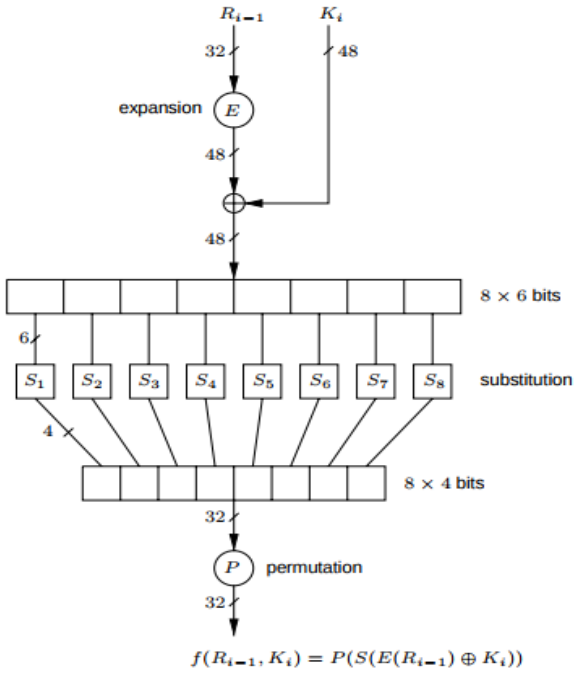
$R_i = L_{i-1} \oplus F(k_i, R_{i-1})$ (32 bits)

$L_{16} = L_{15} \oplus F(k_{16}, R_{15})$ (32 bits)

$R_{16} = R_{15}$ (32 bits)

Fonction F

$F(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i))$



- On a une faute (single bit flip) sur R15 :

$R15^* = R15 \oplus e$ **avec** e (32bits)=(...1...) un seul bit a 1 et les autres a 0.

- Exploiter cette faute :

on a

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

d'où

$$L_{16} \oplus L_{16}^* = F(k_{16}, R_{15}) \oplus F(k_{16}, R_{15} \oplus e)$$
$$= P(S(E(R_{15}) \oplus k_{16})) \oplus P(S(E(R_{15} \oplus e) \oplus k_{16}))$$
$$= P(S(E(R_{15}) \oplus k_{16}) \oplus S(E(R_{15} \oplus e) \oplus k_{16}))$$
$$= P(S(E(R_{15}) \oplus k_{16}) \oplus S(E(R_{15}) \oplus E(e) \oplus k_{16}))$$
$$R_{15} \oplus R_{16}^* = e$$

Car : - $P(x) \oplus P(x^*) = P(x \oplus x^*)$ permutation

- $E(x) \oplus E(x^*) = E(x \oplus x^*)$ (vue au TD)

- Donc on a trouver le bit faux (e) .

- On va maintenant s'intéresser à la fonction interne F du DES :

+ on calcule :

$$A = P^{-1}(L_{16} \oplus L_{16}^*) = S(E(R_{15}) \oplus k_{16}) \oplus S(E(R_{15} \oplus e) \oplus k_{16})$$

Les Inconnues : 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 k16, pour en avoir au final 48 bits.

La complexité de cette attaque est donc de 8×2^6 .

-Trouver K

Une fois K_{16} trouver on aura 16 bits à retrouver dont 8 bits de parité.

Pour les 8 bits de clé on fait une recherche exhaustive.

La complexité : 2^8

NB :

pour cette attaque il nous faudra 32 chiffrés faux

car : on a 4 possibilité (6bits) de k_{16} pour chaque Sbox et nous avons 8 Sbox donc 4*8=32.

Question 2 (Application concrète)

1/

En suivants les étapes décrites dans le ***QUESTION 1*** **.**

Il faut trouver les Sbox erronés : (méthode utilisée dans mon programme)

-Calculer **Expansion(e)** est après parcourir les 48 bits de résultat (6bit par 6bit).

-Pour chaque 6 bit si c'est différent de 0 alors le Sbox est faux.

-On a au max deux Sbox erronés.

Pour chaque Sbox erroné i :

-On cherche les 6bits de k_{16} pour les quelles

$(S(E(R_{15}) \oplus k_{16}) \oplus S(E(R_{15} \oplus e) \oplus k_{16}))_{4bits}$ est égale au 4 bit de A qui correspond au sortie de Sbox erroné.

l'avantage :

c'est qu'une fois que la Sbox est vérifiée pour une faute on aura les valeurs possibles de k_{16} pour cette Sbox qui seront enregistrés dans une table donc si on a une autre faute sur la même Sbox une autre fois on vas juste tester les valeurs enregistrés pour cette Sbox.

d'ou la **complexité** $2^6\ 2^3 = 2^9$

2^3 = nombre de Sbox

2^6 = pour trouver les 6bits de K_{16}

A la fin après avoir testé 32 fautes pour récupère K_{16} il faut juste prendre les valeurs enregistrés pour chaque Sbox (une valeur par Sbox) puis il suffit de les rassembler pour construire K_{16}.

```
s 1 :
3e
s 2 :
1b
s 3 :
3c
s 4 :
5
s 5 :
27
s 6 :
6
s 7 :
5
s 8 :
e
k16 en binaire
111110011011111100000101100111000110000101001110
k16 = f9bf059c614e
```

2/

K_{16}(BINAIRE) = 111110011011111100000101100111000110000101001110
$K_{16=}$(HÉXA) = F9 BF 05 9C 61 4E

Question 3 (Retrouver la clé complète du DES)

1/Comment on peut trouver les bits manquants

- On a trouver K_{16} (48 bits) , il nous reste 16 bits à retrouver dont 8 bits de parité.

- Pour retrouver la position des 48 bits de K16 dans K :

NB
Lors de la génération des sous clé, voici le cycle de rotation :
1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 soit au total 28 rotations.
On en déduit donc que C16 = C0 et D16 = D0

- Il nous suffit donc d'appliquer deux permutations inverses.
Soit PC1InvF et PC2InvF les permutations inverse respectivement de PC1 et PC2.
Telle que :

k' = PC1InvF(PC2InvF(K_{16}));

- Les positions des bits de parité sont connus :
8, 16, 24, 32, 40, 48, 56, 64

- Les 8 bits manquants de k aurrant les positions suivants :
14, 15, 19, 20, 51, 54, 58, 60

- on fait une recherche exhaustive sur ces 8 bits

complexité : 2^8

meilleur idée : (celle que j'ai appliqué dans mon programme)

au lieu de faire 2^8 calcule pour trouver k(58 bits) puis calculer les bits de parité .
pour chaque 8 bit de la recherche exhaustive j'applique la fonction de Parité qui calcule les bits de parité .
Un fois on aura un K probable, je l'envoie avec le message claire au site (Dans l'énoncer)

http://www.emvlab.org/descalc/

Puis je test si le chiffré est égale au vrai chiffré
si c'est le cas alors on a trouver le K (64 bits).

complexité : (au pire des cas) 2^8

```
//trouver k56 (8 bits incunnues )
long chercher_k(long x,long chiffre,long clair){
        long pos[] = {14, 15, 19, 20, 51, 54, 58, 60};
        long res = 0x0L;
        long tmp = 0x00L;
        long k48 = PC1InvF(PC2InvF(x));
        long k;
        while(tmp < 256){
                for(int j = 0; j < 8; j++) {
                        res |= ((tmp >> j) & 1) << (64 - pos[j]);
                }
                k= k48 | res;
                ++tmp;
                res=0x0L;
                k=Pariter(k);
                if(verifierK(k, clair,chiffre) == 0)
                        return k;
        }
}
```

```
//verification k par le site  http://www.emvlab.org/descalc/
int verifierK(long k,long clair,long chiffre){
        char command[200];
        char output[17];

        snprintf(output, 50, "%lx", k);
        for(int i=0;output[i]!='\0';i++){

          output[i]=toupper(output[i]);
     }

        strcpy( command, " curl \"http://www.emvlab.org/descalc/?key=");
        strcat(command,output);
        strcat(command,"&iv=0000000000000000&input=");
        snprintf(output, 50, "%lx", clair);
        for(int i=0;output[i]!='\0';i++){

          output[i]=toupper(output[i]);
     }
        strcat(command,output);
        strcat(command,"&mode=ecb&action=Encrypt&output=\" | grep \"");
        snprintf(output, 50, "%lx", chiffre);
        for(int i=0;output[i]!='\0';i++){

          output[i]=toupper(output[i]);
     }
        strcat(command,output);
        strcat(command,"\" >> nil");

        int res=system(command);
        return res;
}
```

2/

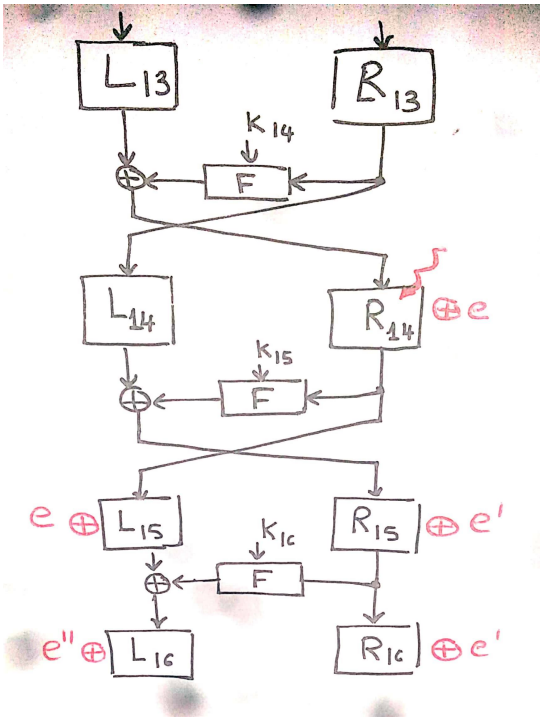
```
100  4402    0  4402    0     0  93659      0 --:--:-- --:--:-- --:--:-- 93659
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  4402    0  4402    0     0   102k      0 --:--:-- --:--:-- --:--:--  102k
k en binaire
00110010011101100110100000110111111100010110100011100000010111111
k = 32766837f168e0bf
hj@hj-Aspire-E1-572:~/Bureau$
```

K (binaire) =00110010011101100110100000110111111100010110100011100000010111111

K (Héxa) = 32 76 68 37 F1 68 E0 BF

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

En supposant cette fois que la faute est provoquée sur la valeur de sortie R14 du 14ème tour



on a :

$L_{16} = L_{15} \oplus F(k_{16}, R_{15})$ $\quad L_{16}^* = e \oplus L_{15} \oplus F(k_{16}, R_{15} \oplus e')$

$R_{16} = R_{15}$ $\quad R_{16}^* = R_{15} \oplus e'$

On remarque alors que :

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

$R_{15} \oplus R_{16}^* = e' = F(k_{15}, R_{14}) \oplus F(k_{15}, R_{14} \oplus e)$

-On chercher le Sbox erroné :

on a :

$$A = P^{-1}(e') = S(E(R_{14}) \oplus k_{15}) \oplus S(E(R_{14} \oplus e) \oplus k_{15})$$

↳ si on découpe la valeur de A en 8 partie de 4bits on vas trouver les Sbox erroné (au maximum 2)

↳une fois le Sbox erroné trouver on a 6 valeur possible (au maximum) pour e sur cette Sbox

000001
ou 000010
ou 000100
ou 001000
ou 010000
ou 100000

↳pour chacune de ces valeurs on fais un recherche exhaustive sur les 6bits de k_{16} dans cette Sbox (2^6) Telle que : $(L_{16} \oplus L_{16}^*)_{6bits} = (e \oplus F(k_{16}, R_{15}) \oplus F(k_{16}, R_{15} \oplus e'))_{6bits}$
soit vérifier

Complexité : $6*8*2^6$

pour cette attaque il nous faudra 192 chiffrés faux

car : on a 4 possibilité (6bits) de k_{16} pour chaque Sbox et nous avons 8 donc 4*8=32

et on fait le calcule pour 6 valeur possibles de e pour chaque Sbox donc 32*6=192

En supposant cette fois que la faute est provoquée sur la valeur de sortie R13 du 13ème tour

↳

Question 5 (Contre-mesures)

Un 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, etc.). 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 s'appuient sur des mécanismes de détection d'erreur au sein du circuit.

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. Un exemple est 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.

Redondance temporelle : basée sur la ré-exécution d'un même calcul sur le même bloc matériel et la comparaison des différents résultats obtenus. La redondance temporelle simple est basée sur la double exécution d'un calcul sur un même bloc de calcul. Les résultats ainsi obtenus sont donc comparés. Le temps de calcul va être multiplié par 2 dans tous les cas.

Solutions algorithmiques : ajout de portions de code de test, exécutés en même temps que l'algorithme. Si une erreur est détectée, on incrémente un compteur, au delà d'une certaine valeur, le système est réinitialisé. Les programmes de test sont très courts et se résument généralement à des lectures/écritures en mémoire, des opérations arithmétiques simples ou toute fonction facile à vérifier. Il rajoutent donc un certain facteur (nombre d'opérations de test) au temps de calcul.