

01100
10110
11110



Data Encryption Standard Breaker

CHAHY, Rabie Ala-Eddine
rabie_chahi@yahoo.fr
Numéro étudiant 21505593

[Site web personnel](#)

[lien du projet Github](#)

Master : SeCReT

Enseignant : Mr Goubin Louis

30 Avril 2020

Abstract

L'association du hardware et du software bas niveau ouvre la porte à des vulnérabilités sur l'ensemble et permet aux attaquants d'accéder aux noyaux des programmes en passant les frontières matérielles à l'aide d'attaques dites physiques. Ces attaques sont découpées principalement en deux grandes classes les attaques par canaux auxiliaires analysant les fuites physiques de l'implémentation matérielle du code et les attaques par injection de fautes qui elles se traduisent par la perturbation du fonctionnement des circuits. Nous nous intéresserons dans ce projet à cette deuxième classe.

Pour réussir une telle attaque, il faut disposer du matériel nécessaire et pouvoir affecter son état interne; il est également important d'être en mesure de comprendre parfaitement le déroulement de chaque étape des processus de chiffrement/déchiffrement de l'algorithme cryptographique lié au programme; cette étape peut parfois prendre plusieurs semaines voir plusieurs mois. Une fois l'algorithme maîtrisé, l'attaquant cherchera à produire volontairement des erreurs dans le cryptosystème. L'effet souhaité par cette injection est le comportement inhabituel des opérations cryptographiques. Si l'attaque est correctement réalisé l'attaquant pourra extraire des informations secrètes telle que la clé utilisée. On retrouve parfois certains procédé en cryptanalyse mêlant une attaque par faute à d'autres attaques telle qu'une timing attack. Ces attaques peuvent porter sur différents composants matériels tels que le crypto-processeur, les logiciels, etc.

Nous aborderons dans le présent document, les détails d'une attaque par injection de fautes sur l'algorithme Data Encryption Standard (DES). Pour ce faire, nous réaliserons une attaque interprétable par un exécutable fourni avec ce même document ainsi que les sources nécessaires à la compilation. Nous supposerons avoir en notre possession un couple formé d'un clair et d'un chiffré correct, ainsi que de trente-deux autres couples formés à chaque fois de ce même message clair et d'un chiffré erroné produit à l'aide de diverses injections de fautes. Notez que ce document est consultable en ligne [ici](#)..

Sommaire

1	Attaque par faute sur le DES	5
2	Application concrète	7
2.1	Description précise de l'attaque	7
2.1.1	Description théorique détaillé	7
2.1.2	Attaque pratique	11
2.1.3	Algorithme	13
2.2	les 48 bits de clé obtenu	14
3	Retrouver la clé complète du DES	16
3.1	Méthode pour trouver la clé K_{64}	16
3.1.1	Trouver les bits manquants	16
3.1.2	Calculs des bits de parité :	18
3.1.3	Algorithme	18
3.2	Les 64 bits de clé obtenu	19
4	Fautes sur les tours précédents	21
4.1	Faute provoquée sur le 14 ^{eme} tour	21
4.2	Faute provoquée sur les autres tours	22

5	Contre mesures	23
6	Compilation et exécutions	23

1 Attaque par faute sur le DES

Data Encryption Standard est un algorithme de chiffrement à clé secrète succédant à l'algorithme Lucifer et développé par IBM 15 Janvier 1977, DES est un chiffrement par bloc de 64 bits basé sur un schéma feistel à 16 tours. La clé initiale fait 64 bits, seulement 56 d'entre eux sont effectivement utilisés par l'algorithme; en effet huit bits sont utilisés uniquement pour le contrôle de la parité. Ces 56 bits sont par la suite transformés en 16 sous clés à travers le Key Schedule. Ces dernières serviront à chiffrer les 2 parties du messages à travers les 16 tours.¹

Qu'est ce qu'une attaque par Faute sur DES ?

Une attaque par fautes sur le Data Encryption Standard consiste à perturber plusieurs fois le processus de chiffrement en injectant des fautes dans le but d'obtenir la clé de chiffrement. Plus concrètement l'attaquant intervertit certains bits de la partie droite R_i du chiffré à la sortie d'un tour choisi parmi ceux qui permettent un procédé calculatoirement faisable², tel que le 15^{eme} tour du Feistel, et ce, un certains nombre de fois, afin d'obtenir plusieurs chiffrés faux à l'aide du même message clair et de la même clé de chiffrement.

L'attaquant peut choisir la position de la faute ainsi que le nombre de bit concernés. Il y'a par définition deux modèles d'erreurs :

- Le single bit flip : un et un seul bit de R_{15} est inversé parmi les 32.
- Le single byte flip : un octet de R_{15} est inversé parmi les 4 ($32 = 4 \times 8$).

Une fois l'injection de fautes effectuée, l'attaquant dispose du message clair, de son chiffré correct et d'un ensemble de messages chiffrés³. Tous les chiffrés sont obtenus en utilisant la même clé de chiffrement de 64 bits que l'attaquant à pour objectif de déterminer. On considérera également que l'attaquant maîtrise suffisamment les différents processus lié au DES et qu'il est capable de chiffrer et de déchiffrer librement des messages avec le DES⁴.

¹Pour plus d'informations, un document est joint détaillant l'algorithme du DES.

²Nous désignerons dans la partie 4 les tours possibles.

³32 dans le cas du one bit flip.

⁴Dan notre cas nous avons à disposition un programme implémentant le DES.

L'attaquant ne disposant pas de la clé, la seule manière de pénétrer le feistel est d'utiliser les sorties⁵ du DES que l'attaquant possède. Il connaît la construction interne du DES et de la fonction f , il se servira donc des messages qu'il a en sa possession pour inverser les opérations liées au chiffrement entre la sortie et le 15^{ème} tour afin de remonter la 16^{ème} sous clé K_{16} . Chaque chiffré faux apporte de l'information sur un certain nombre de bits de la clé. En effet nous verrons par la suite que chaque boîte S est concernée par un nombre précis de message, cela est dû au simple bit flip sur R_{15} qui par construction infecte l'entrée d'au plus 2 S-Box.

Une fois K_{16} obtenue, une analyse du *key schedule* et des positions des 8 bits perdus lors du passage de 56 bits de clés à 48 bits⁶, permettra à l'attaquant de remonter ces derniers. L'attaquant aura en sa possession à la fin de cette étape une clé de 64 bits dont le 8^{ème} bit⁷ de chaque octet est potentiellement erroné.

Finlament, l'attaquant n'aura plus qu'à calculer les 8 bits de parités en mettant le bit à 0 si l'octet a un nombre impair de 1 et inversement. À la fin de cette étape l'attaquant obtient une clé de 64 bits et procède à un test de cette dernière en chiffrant avec le message clair.

Si la sortie du chiffrement correspond au chiffré correct, la clé est donc bel et bien K_{64} et l'attaque est terminée.

Cette attaque est beaucoup plus puissante et beaucoup moins coûteuse que des attaques classique tel qu'une attaque par force brut de complexité 2^{56} opérations ou encore l'attaque par recherche exhaustive avec complément bit à bit de complexité 2^{54} opérations. la complexité de l'attaque par fautes sera calculée dans la prochaine partie.

Ce qui précède est une ébauche théorique de l'attaque, nous détaillerons dans les prochaines sections la méthode pratique à utiliser pour retrouver la clé de chiffrement. Pour ce faire nous avons écrit un programme en C++, réalisant l'attaque et montrant son déroulement à travers une animation⁸.

⁵Les différents chiffrés obtenus précédemment

⁶On analysera les différentes tables de permutations du DES.

⁷Bit de parité

⁸Les détails pour la compilation et l'exécution sont décrits dans le README joint.

2 Application concrète

2.1 Description précise de l'attaque

2.1.1 Description théorique détaillé

Comme décrit dans la section précédente nous devons commencer par déterminer K_{16} afin de remonter la clé de chiffrement initiale.

Pour ce faire nous allons suivre les étapes suivantes :

1. Analyser la structure Feistel du DES :

Nous avons à chaque tour les équations suivantes :

$$\begin{aligned} R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \\ L_i &= R_{i-1} \end{aligned}$$

L'injection de faute se fait sur le 15^{ème} tour nous pouvons donc écrire :

$$\begin{aligned} R_{16}^c &= L_{15} \oplus f(R_{15}^c, K_{16}) \\ L_{16}^c &= R_{15}^c \end{aligned}$$

De la même manière : $\forall R_{16}^f \in L$

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

Nous remarquons que K_{16} et L_{15} sont communs à R_{16}^c et R_{16}^f nous éliminerons donc les L_{15} de la manière suivante : $\forall R_{16}^f \in L$

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

$$\begin{aligned} R_{16}^c \oplus R_{16}^f &= (L_{15} \oplus L_{15}) \oplus (f(R_{15}^c, K_{16}) \oplus f(R_{15}^f, K_{16})) \\ R_{16}^c \oplus R_{16}^f &= f(R_{15}^c, K_{16}) \oplus f(R_{15}^f, K_{16}) \end{aligned}$$

2. Analyser la fonction f :

La fonction f est la fonction interne du DES, elle prend en entrée une sous-clé K_i et une partie droite R_{i-1} . Elle calcule d'abord l'expansion de R_{i-1} qui passe de 32 à 48 bits et calcule ensuite $E(R_{i-1}) \oplus K_i$ qui sera utilisé comme entrée des 8 S-box.

L'entrée des S-boxs fait donc 48 bits⁹ et donne en sortie 32 bits¹⁰.

Nous pouvons donc écrire :

$$\begin{aligned} f(R_{i-1}, K_i) &= P(S(E(R_{i-1}) \oplus K_i)) \\ f(R_{i-1}, K_i) &= P(S_1(E(R_{i-1} \oplus K_i)_{b_0 \rightarrow b_5}) || \dots || S_8(E(R_{i-1} \oplus K_i)_{b_{43} \rightarrow b_{48}})) \end{aligned}$$

Dans notre cas nous nous aurons : $\forall R_{16}^f \in L$

$$\begin{aligned} f(R_{15}^c, K_{16}) &= P(S(E(R_{15}^c) \oplus K_{16})) \\ f(R_{15}^f, K_{16}) &= P(S(E(R_{15}^f) \oplus K_{16})) \end{aligned}$$

3. Parcours inverse de la fonction f :

Le but est de remonter jusqu'à K_{16} qui se trouve en entrée de la S-Box dans le calcul $E(R_{15}) \oplus K_{16}$

D'après les formules précédentes : $\forall R_{16}^f \in L$

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

Par conséquent : $\forall R_{16}^f \in L$

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

$$P^{-1}(R_{16}^c \oplus R_{16}^f) = P^{-1}(P(S(E(R_{15}^c) \oplus K_{16}))) \oplus P^{-1}(P(S(E(R_{15}^f) \oplus K_{16})))$$

$$P^{-1}(R_{16}^c \oplus R_{16}^f) = S(E(R_{15}^c) \oplus K_{16}) \oplus S(E(R_{15}^f) \oplus K_{16})$$

⁹6 bits par S-Box

¹⁰4 bits par S-Box

4. Lier les messages chiffrés faux à la S-Box qu'ils infectent :

L'injection de faute est de type one bit flip¹¹ il y'a donc un seul bit erroné dans chaque R_{15}^f . Il est donc important de remarquer qu'un chiffré erroné ne fausse le résultat que de certaines S-Boxes.

En analysant la permutation d'expansion, le bit erroné peut être ajouté ou pas une seconde fois dans les 16 bits d'expansions; on en conclue donc qu'un chiffré ne peut infecter que deux S-box au maximum. En conséquence de quoi, nous déterminerons les messages infectants chaque S-box, et décomposerons donc nos messages en 8 listes. Chaque liste est donc lié à une S-box en particulier ce qui convient à dire que chaque liste donne de l'information sur les 4 bits de sortie de la S-box en question.

À la fin de cette étape nous chercherons à obtenir les 6 bits en entrée de chaque S-Box séparément.

Rappelons que : $\forall R_{16}^f \in L$

$P^{-1}(R_{16}^c \oplus R_{16}^f)$ désigne l'entrée des S-Box et qu'il est facilement calculable.

Plus concrètement : $\forall R_{16}^f \in L$, pour $i \in \{1, \dots, 29\}$ et $j \in \{1, \dots, 8\}$

$$P^{-1}(R_{16}^c \oplus R_{16}^f)_{b_i \rightarrow b_{i+3}} = S_j(E(R_{15}^c) \oplus K_{16})_{b_i \rightarrow b_{i+3}} \oplus S_j(E(R_{15}^f) \oplus K_{16})_{b_i \rightarrow b_{i+3}}$$

5. Attaque exhaustive sur K16 :

Après avoir généré les 2^{64} ¹² mots de 6 bits, nous ferons une recherche exhaustive sur la clé K16, en remplaçant donc à chaque fois K16 par l'un des 64 mots de 6 bits générés, et ce en calculant donc pour chacune des S-box et pour chaque chiffré erroné lié à cette dernière :

$$S_j(E(R_{15}^c) \oplus K_{16})_{b_i \rightarrow b_{i+3}} \oplus S_j(E(R_{15}^f) \oplus K_{16})_{b_i \rightarrow b_{i+3}} \text{ pour } i \in \{1, \dots, 29\} \text{ et } j \in \{1, \dots, 8\}$$

¹¹Nous procéderons évidemment à la vérification du type d'injection avant de commencer l'attaque.

¹²Toutes les combinaisons possibles de 6 bits

Si ce calcul est équivalent à $P^{-1}(R_{16}^c \oplus R_{16}^f)_{b_i \rightarrow b_{i+3}}$ pour $i \in \{1, \dots, 29\}$ et $j \in \{1, \dots, 8\}$ alors les 6 bits utilisé pour remplacer K_{16} sont une partie¹³ candidate pour K_{16} , lié au chiffré faux sur lequel le calcul est effectué.

Naturellement chaque chiffré faux de chaque S-Box admet plusieurs solutions candidates. À la fin de cette étape nous avons donc plusieurs solutions de 6 bits lié à chaque chiffré faux de chaque S-box.

6. Déterminer K_{16} :

Il nous suffira finalement de trouver dans un premier temps la solution commune aux différentes solutions des chiffrés de chaque S-box afin de déterminer les 8 parties de 6 bits de la sous-clé réelle K_{16} .

En concaténant dans l'ordre les 8 parties on aura un mot sur 48 bits qui n'est autre que la sous-clé K_{16}

¹³6 bits

2.1.2 Attaque pratique

Nous avons implémenté un programme en c++ permettant d'effectuer l'attaque.

Nous avons choisi de travailler en C++ en évitant les types complexes tel que les bitset, les long etc, afin d'ajouter ce programme à un programme plus général traitant des attaques par fautes. Nous utiliserons des strings tout au long de l'implémentation.

Les fichiers utilisés :

- `Concerter.cpp` : contient un convertisseur pour gérer les différentes conversions (binaire - hexadécimal , binaire - décimal etc...)
- `Utils.cpp` : contient des méthodes gérant les opérations usuelles tels que le xor.
- `Des.cpp` : contient une implémentation du DES.
- `Analyzer.cc` : contient les méthodes liées à l'analyse de l'injection de faute.
- `Reverser.cpp` : contient les méthodes liées à la pénétration inverse du DES.
- `Attack.cpp` : contient les méthodes responsables des différentes attaques de ce projet.
- `Printer.cpp` : contient les méthodes liées à l'animation de l'attaque.

Le fichier `Attack.cpp` contient la méthode `attack48()` qui va implémenter cette attaque comme suit :

1. Annuler la permutation IP^{-1} en effectuant simplement une permutation IP sur le chiffré correct.
2. Découper le message obtenu en 2 parties : L_{16}^c et R_{16}^c
3. Échanger L_{16}^c et R_{16}^c et stocker L_{16}^c dans R_{15}^c
4. Calculer l'expansion de R_{15}
5. Mettre en relation les sbox et les chiffrés faux concernés

6. Générer les 64 mots de 6 bits possibles.
7. Pour chaque sbox
 - 7.1. Pour chaque chiffré faux lié
 - i. Permutation IP sur le chiffré faux
 - ii. Découper le message obtenu en 2 parties : L_{16}^f et R_{16}^f
 - iii. Échanger L_{16}^f et R_{16}^f et stocker L_{16}^f dans R_{15}^f
 - iv. Calculer l'expansion de R_{15}^f
 - v. calculer $R_{16}^c \oplus R_{16}^f$
 - vi. calculer la valeur attendue des 4bits à la sortie de la S-box avec la permutation $P^{-1}(R_{16}^c \oplus R_{16}^f)$
 - vii. Pour chaque mot de 6 bits parmi les 64
 - A. Calcul de l'entrée de la sbox lié au chiffré correct
 - B. Calcul de l'entrée de la sbox lié au chiffré incorrect
 - C. Calcul de la sortie de la sbox lié au chiffré correct
 - D. Calcul de la sortie de la sbox lié au chiffré incorrect
 - E. Calcul du xor entre les deux sortie
 - F. Comparer le résultat et la valeur de vérification sur 4 bits
 - G. S'ils correspondent, stocker les 6 bits comme solutions valide pour le chiffré en question.
 - viii. Ajouter la liste de solutions de ce chiffré au listes des autres chiffrés infectants la S-Box actuelle.
8. Trouver les 8 solutions communes aux solutions des chiffrés de chaque S-Box
9. Concaténer les 8 solutions pour former un mot de 48 bits désignant K_{16}

Le code est entièrement commenté et permet donc de comprendre très facilement l'implémentation.

2.1.3 Algorithme

Algorithm 1 Find Subkey K_{16}

Input : plaintext, ciphertext, ciphers[]
Output : K_{16}

```

1: procedure FIND  $K_{16}$ 
2:    $correct \leftarrow IP(ciphertext)$ 
3:    $R_{16}^c \leftarrow SplitTwoParts(correct[0])$ 
4:    $L_{16}^c \leftarrow SplitTwoParts(correct[1])$ 
5:    $Swap(R_{16}^c, L_{16}^c)$ 
6:    $R_{15}^c \leftarrow L_{16}^c$ 
7:    $Expanded^c \leftarrow Expansion(R_{15}^c)$ 
8:    $links[ ] \leftarrow Link(ciphers)$ 
9:    $binaries[ ] \leftarrow GenerateAttackKeys(6)$ 
10:   $\alpha \leftarrow 0$   $\beta \leftarrow 0$   $l \leftarrow 0$ 
11:  for  $i \in \{1, \dots, 8\}$  do
12:    for  $j \in \{1, \dots, 6\}$  do
13:       $incorrect \leftarrow IP(ciphers[links[i][j]])$ 
14:       $R_{16}^f \leftarrow SplitTwoParts(correct[0])$ 
15:       $L_{16}^f \leftarrow SplitTwoParts(correct[1])$ 
16:       $Swap(R_{16}^f, L_{16}^f)$ 
17:       $R_{15}^f \leftarrow L_{16}^f$ 
18:       $Expanded^f \leftarrow Expansion(R_{15}^f)$ 
19:       $xored \leftarrow Xor(R_{16}^c, R_{16}^f)$ 
20:       $expected \leftarrow UndoFFunctionPermutation(xored)$ 
21:      for  $k \in \{1, \dots, 64\}$  do
22:         $input^c \leftarrow Xor(Expanded_{\alpha \rightarrow \alpha+6}^c, binaries[k])$ 
23:         $input^f \leftarrow Xor(Expanded_{l \rightarrow l+6}^f, binaries[k])$ 
24:         $output^c \leftarrow GetSbox(input^c, i)$ 
25:         $output^f \leftarrow GetSbox(input^f, i)$ 
26:         $result \leftarrow Xor(output^c, output^f)$ 
27:        if  $result = expected_{\beta \rightarrow \beta+4}$  then
28:           $solutions[i][j][l] \leftarrow binaries[k]$ 
29:         $l \leftarrow 0$ 
30:       $\alpha \leftarrow \alpha + 6$ 
31:     $\beta \leftarrow \beta + 4$ 
32:   $parts[ ] \leftarrow FindCommonSultion(solutions)$ 
33:   $K_{16} \leftarrow Concatenate(parts)$ 
34:  return  $K_{16}$ 

```


Complexité : La complexité cryptographique de cet algorithme dépend de la recherche exhaustive, les autres opérations se font en temps constant et sont donc négligeables. Nous avons un total de 8 S-box et chaque est lié à 6 messages différents, chaque message opère une boucle de 2^6 tours, nous avons donc finalement une complexité de $8 \times 6 \times 2^6 = 2^{10} \times 3 \approx 2^{12}$

2.2 les 48 bits de clé obtenu


La clé K_{16} obtenu est [F180096CF02](#).

Quelques passage de l'animation :

Traîtement des S-box 1 et 2 :

```
TRETEMENT OF  SBOX-1 ...


CIPHER 28 [ 000100 000101 010100 010101 011010 011011 100010 100011 111100 111101 ]
CIPHER 29 [ 100000 100010 100100 100110 101001 101011 110100 110101 110110 110111 111100 111110 ]
CIPHER 30 [ 010001 010101 011000 011001 011100 011101 111000 111011 111100 111111 ]
CIPHER 31 [ 000011 001011 010010 011010 110010 110100 110110 110111 111010 111100 111110 111111 ]
CIPHER 32 [ 000010 001100 010010 011100 101100 111100 ]
CIPHER 1 [ 000101 001000 001100 011000 011001 011100 100101 101000 101100 111000 111001 111100 ]

TRETEMENT OF  SBOX-2 ...


CIPHER 24 [ 001000 001001 010100 010101 011000 011001 011010 011011 101010 101011 111010 111011 111110 111111 ]
CIPHER 25 [ 011000 011001 011010 011011 100000 100001 100010 100011 101101 101111 ]
CIPHER 26 [ 001000 001100 011000 011100 100010 100110 101000 101001 101100 101101 ]
CIPHER 27 [ 000100 001100 010000 011000 ]
CIPHER 28 [ 001000 001001 001100 011000 011001 011100 100111 101100 110111 111100 ]
CIPHER 29 [ 011000 111000 ]
```

Détection de la solution en commun pour les S-box 1 et 2 :

```
DETRERMINATION OF THE SOLUTIONS COMMON TO EACH S-BOX ...

COMMON SOLUTION OF  SBOX-1 ...

CIPHER 28 [ 000100 000101 010100 010101 011010 011011 100010 100011 111100 111100 111101 ]
CIPHER 29 [ 100000 100010 100100 100110 101001 101011 110100 110101 110110 110111 111100 111110 ]
CIPHER 30 [ 010001 010101 011000 011001 011100 011101 111000 111011 111100 111100 111111 ]
CIPHER 31 [ 000011 001011 010010 011010 110010 110100 110110 110111 111010 111100 111110 111111 ]
CIPHER 32 [ 000010 001100 010010 011100 101100 111100 ]
CIPHER 1 [ 000101 001000 001100 011000 011001 011100 100101 101000 101100 111000 111001 111100 ]

COMMON SOLUTION OF  SBOX-2 ...

CIPHER 24 [ 001000 001001 010100 010101 011000 011001 011010 011011 101010 101011 111010 111011 111110 111111 ]
CIPHER 25 [ 011000 011001 011010 011011 100000 100001 100010 100011 101101 101111 ]
CIPHER 26 [ 001000 001100 011000 011100 100010 100110 101000 101001 101100 101101 ]
CIPHER 27 [ 000100 001100 010000 011000 011000 ]
CIPHER 28 [ 001000 001001 001100 011000 011001 011100 100111 101100 110111 111100 ]
CIPHER 29 [ 011000 011000 111000 ]
```

Formation de K_{16} :

!! EXAUSTIVE RESEARCH THEREFORE GIVES THE FOLLOWING COMMON SOLUTIONS ...

S-BOX 1	[111100		3C]
S-BOX 2	[011000		18]
S-BOX 3	[000000		00]
S-BOX 4	[001001		09]
S-BOX 5	[011011		1B]
S-BOX 6	[001111		0F]
S-BOX 7	[110000		30]
S-BOX 8	[000010		02]

🔗 CONCATENATE THE 8 WORDS OF 6 BITS TO FORM THE FOLLOWING 48 BITS WORD...

[111100 011000 000000 001001 011011 001111 110000 000010]

🔑 111100011000000000000100101101100111110000000010

K_{16} en hexadécimal :

🛡️ DIFFERENTIAL FAULT ANALYSIS ON DATA ENCRYPTION STANDARD 🛡️

GIVEN DATA 📄

- Plaintext : 89CDE529B03B2A29
- Ciphertext : CE3B17D676D5F63B

K16 : F180096CFC02 (The last key of key schedule algorithm)

® CHAHI RABIE ALA EDDINE 2020

3 Retrouver la clé complète du DES

3.1 Méthode pour trouver la clé K_{64}

3.1.1 Trouver les bits manquants

Dans ce qui précède nous avons réussi à retrouver la clé K_{16} de 48 bits, il nous faut retrouver les 8 bits manquants pour avoir les 56 bits de clé ainsi que les 8 bits de parités restants pour compléter la clé initiale de celle 64 bits.

Nous allons faire ceci en plusieurs étapes :

1. Analyse du la key schedule :

La clé initiale K_{64} subit une première permutation et perd ses bits de parité, la clé obtenu fait donc 56 bits, cette clé va être découpé en deux parties qui subiront des shifts circulaires¹⁴ avant d'être concaténer pour former la sous clé du tour.

2. Conclusion du key schedule :

La première sous clé est correspondante à la dernière car la somme des shifts circulaires donne 28 comme ceci :

$$\sum_{n=1}^{n=16} shift_i = 28$$

Ce qui correspond à la taille des deux blocs de chaque sous clé.

Il suffit donc d'effectuer les permutations inverses de $PC1$ et $PC2$ sur K_{16} pour avoir :

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

¹⁴décalage d'une 1 ou 2 positions selon le numéro du tour

Seulement 8 bits sont perdus entre K_{56} et K_{16} il faudra donc déterminer la position ces 8 bits à l'aide d'une recherche exhaustive. En effet, le nombre de tous les mots de 8 bits est 2^8 ce qui est largement calculable avec une machine en 2020.

3. Déterminer les positions des bits manquants :

En analysant la permutation $PC2$. nous avons pu déduire les positions des bits manquants :

Positions des bits manquants¹⁵ = {9, 18, 22, 25, 35, 38, 43, 54}

Nous appliquerons ensuite la permutation $PC1$ c'est pourquoi nous devons l'analyser également afin de déterminer la nouvelle position de ces bits.

Positions des bits manquants¹⁶ = {14, 15, 19, 20, 51, 54, 58, 60}

4. Attaque par recherche exhaustive sur les 8 bits manquants :

Nous calculons donc dans un premier temps $PC2^{-1}(K_{16})$, nous obtenons une clé de 56 bits dont les bits manquants sont mis à zéro. Nous calculerons par la suite $PC1^{-1}(PC2^{-1}(K_{16}))$ afin d'obtenir une clé sur 64 bits dont les bits manquants ainsi que les bits de parité sont mis à 0.

Nous ferons par la suite une recherche exhaustive sur toutes les combinaisons possible de 8 bits générés préalablement, en testant à chaque fois les 8 bits candidats en faisant un chiffrement DES sur le text clair donné en le comparant au chiffré correct, s'ils correspondent on arrête et on a trouvé les 56 bits de clé.

Notons que ceci est possible car les 8 bits de parité n'interviennent pas ni dans le key schedule ni dans le chiffrement contrairement au 8 bits perdus.

¹⁵Après $PC2$

¹⁶Après $PC1$

3.1.2 Calculs des bits de parité :

L'étape précédente nous a permis d'avoir une clé sur 64 bits dont le 8^{eme} bit de chaque octet désignant le bit de parité est potentiellement erronné. Retrouver le bit de parité d'origine est très simple, il suffit de mettre chaque 8ème bit de chaque octet à 0 si ce dernier possède a un nombre impair de 1 et à 0 dans le cas contraire.

Nous devons comme étape ultime vérifié le résultat de l'attaque en chiffrant le message clair à l'aide de K_{64} qu'on vient d'obtenir et de comparer le résultat au chiffré correct. S'ils sont équivalents, la clé trouvée est bel bien la clé complète du DES.

Cette attaque est implémenté dans les deux méthodes `attack56()` et `attack64()` dans le fichier `attack.cpp`.

3.1.3 Algorithme

Algorithm 2 Find key K_{64}

Input : plaintext, ciphertext, K_{16}

Output : K_{64}

```
1: procedure FIND  $K_{64}$ 
2:    $binaries[ ] \leftarrow \text{GenerateAttackKeys}(8)$ 
3:    $K_{56} \leftarrow PC2^{-1}(K_{16})$ 
4:    $K_{64} \leftarrow PC1^{-1}(K_{56})$ 
5:   for  $i \in \{1, \dots, 256\}$  do
6:      $K_{64} \leftarrow \text{GetCandidate}(K_{64}, binaries[u])$ 
7:      $temp \leftarrow DES_{K_{64}}(plaintext)$ 
8:     if  $temp = ciphertext$  then
9:        $K_{64} \leftarrow temp$ 
10:      break
11:  for ( $i = 0, i \leq 63, i+ = 8$ ) do
12:     $K_{64} \leftarrow \text{ComputeParityBits}(K_{64}, i)$ 
13:  if  $DES_{K_{64}}(plaintext) \neq ciphertext$  then
14:    Error( $K_{64}$ )
15:  return  $K_{64}$ 
```

3.2 Les 64 bits de clé obtenu

La clé K_{64} obtenu est [2C58400B195B802A](#).

Quelques étapes de l'animation :

Calculer les inverses de $PC1$ et $PC2$:

STEP 3 [DETERMINE THE 56 BITS OF THE KEY : FIND THE 8 LOST BITS]

UNDO $PC2(K_{16})$...

$PC^{-2}(F180096CFC02F180096CFC02) = 4026813A801BB2$

UNDO $PC1(PC^{-2}(K_{16}))$...

$PC^{-1}(4026813A801BB2) = 2C58400A185A802A$

!! NOTE THAT WE DON'T KNOW YET THE 8 MISSING BITS THEN THEY ARE SET TO 0

Trouver les 56 bits de clé :

⚠ EXHAUSTIVE SEARCH ON THE 8 LOST BITS USING DES ...

THE 8 BITS THAT CORRECTLY ENCRYPT THE PLAINTEXT GIVE THE FOLLOWING 64-BIT KEY...

🔑 $K_{64}[2C58400A185A802A]$

!! NOTE THAT THE PARITY BITS ARE WRONG

Calculer les bits de parité :

STEP 4 [DETERMINE THE 64 BITS OF THE KEY : FIND THE 8 PARITY BITS]

COMPUTATION OF PARITY BITS ...

🔑 K64[2C58400B195B802A]

Vérifier la clé obtenu :

STEP 5 [CHECK THE OBTAINED KEY]

ENCRYPT THE PLAINTEXT WITH THE OBTAINED KEY 🔑 AND COMPARE IT TO THE GIVEN CIPHERTEXT ...

DES[2C58400B195B802A](89CDE529B03B2A29) = {CE3B17D676D5F63B}

CIPHERTEXT = [CE3B17D676D5F63B] ✓

🔑 [2C58400B195B802A] IS THEREFORE THE RIGHT KEY USED, THE ATTACK WORKED 🌟🌟🌟🌟🔄

4 Fautes sur les tours précédents

Rappelons que la complexité de notre algorithme pour trouver K_{16} à l'aide d'une injection par faute sur le 15^{eme} tour est d'environ 2^{12} .

4.1 Faute provoquée sur le 14^{eme} tour

On raisonnait de la même manière que précédemment on obtient 2 équations :

$$R_{15}^c = L_{14}^c \oplus f(R_{14}^c, K_{15})$$

$$L_{15}^c = R_{14}^c$$

Mais aussi :

$$R_{15}^f = L_{14}^f \oplus f(R_{14}^f, K_{15})$$

$$L_{15}^f = R_{14}^f$$

Nous avons par conséquent :

$$R_{15}^c \oplus R_{15}^f = L_{14}^c \oplus f(R_{14}^c, K_{15}) \oplus L_{14}^f \oplus f(R_{14}^f, K_{15})$$

Ce qui donne :

$$P^{-1}(R_{15}^c \oplus R_{15}^f) = S_k(E(R_{14}^c) \oplus K_{15}) \oplus S_k(E(R_{14}^f) \oplus K_{15})$$

Par ailleurs :

$$R_{14}^c = L_{15}^c = R_{16}^c \oplus f(K_{16}, L_{16}) \Leftrightarrow P^{-1}(L_{15}^c \oplus R_{16}^c)_{i \rightarrow j} = S_k(E(L_{16}^c) \oplus K_{16})_{i \rightarrow j}$$

$$R_{14}^f = L_{15}^f = R_{16}^f \oplus f(K_{16}, L_{16}^f) \Leftrightarrow P^{-1}(L_{15}^f \oplus R_{16}^f)_{i \rightarrow j} = S_k(E(L_{16}^f) \oplus K_{16})_{i \rightarrow j}$$

Le but est de trouver K_{15} , nous procéderons donc à une recherche exhaustive pour déduire les possible L_{15}^c et L_{15}^f en utilisant :

$$R_{14}^c = L_{15}^c = R_{16}^c \oplus f(K_{16}, L_{16}) \Leftrightarrow P^{-1}(L_{15}^c \oplus R_{16}^c)_{i \rightarrow j} = S_k(E(L_{16}^c) \oplus K_{16})_{i \rightarrow j}$$

$$R_{14}^f = L_{15}^f = R_{16}^f \oplus f(K_{16}, L_{16}^f) \Leftrightarrow P^{-1}(L_{15}^f \oplus R_{16}^f)_{i \rightarrow j} = S_k(E(L_{16}^f) \oplus K_{16})_{i \rightarrow j}$$

pour les utiliser dans une deuxième recherche exhaustive pour trouver K_{15} en utilisant :

$$P^{-1}(R_{15}^c \oplus R_{15}^f) = S_k(E(R_{14}^c) \oplus K_{15}) \oplus S_k(E(R_{14}^f) \oplus K_{15}).$$

Le fait de remonter les équations de cette manière permet théoriquement de remonter le DES quelque soit le tour dans lequel les fautes sont injecté, quoique la réalité est toute autre, nous verrons dans la suite les limites de cette attaque.

Ces deux opérations sont de même complexité que la recherche exhaustive sur K_{16} donc la complexité d'attaque sur chaque tour implique de multiplier la complexité de l'attaque sur K_{16} par la complexité du tour qui précède, la complexité est donc élevée au carré pour le 14^{eme} tour et donne $(2^{12})^2 = 2^{24}$.

Rappelons que la loi de Moore indique qu'il est possible d'aller jusqu'à 2^{80} dans un contexte publique. Cette attaque est donc largement faisable.

4.2 Faute provoquée sur les autres tours

Comme dit plus haut cette attaque permet de remonter dans le DES quelque soit le tour, seulement les capacités calculatoire étant limité à 2^{80} et sachant que notre attaque est élevée au carré à chaque tour nous avons donc les complexité suivante

- 15^{eme} tour : 2^{12}
- 14^{eme} tour : 2^{24}
- 13^{eme} tour : 2^{36}
- 12^{eme} tour : 2^{48}
- 11^{eme} tour : 2^{60}

- 10^{eme} tour : 2^{72}
- 9^{eme} tour : 2^{84}

L'attaque est donc calculatoirement possible jusqu'au 10 ème tour !

5 Contre mesures

Voici une liste de contre mesure empêchant l'attaque par faute sur le Des :

- Déchiffrer le message obtenu à l'aide de la clé avant de retourner le résultat pour vérifier s'il s'agit du bon chiffré. Dans le cas contraire aucun résultat ne devrai être retourné après un certains nombre de calcul. Cette méthode se décline sous plusieurs format, en effet certaine amélioration permettent de ne refaire qu'un certains nombre de calcul pour optimiser le temps d'exécutions. Néanmoins, dans tous les cas cette contre mesure nécessite un temps d'exécution supérieur car la vérification est supposé être automatique.
- Empêcher physiquement l'injection de faute ,en protégeant le matériel physique qui exécute le DES. Cette approche est très peu utilisé car le matériel de protection coûte souvent très cher.

6 Compilation et exécutions

Le programme fournit contient un script de compilation pour compiler le programme il suffit de taper : make pour l'exécution deux programmes sont disponible :

1. ./FaultAttackAnalysis animation : propose une animation totale de toutes les étapes décrites dans ce document.
2. ./FaultAttackAnalysis result : donne les résultats de l'attaque.