Rapport de Projet
Attaque par faute sur DES

BEN-CHARKE Maryem
22406003

Encadrant : Prof. Louis Goubin

Mai 2025

Introduction

Ce rapport présente une attaque par fautes appliquée à l'algorithme de chiffrement symétrique DES (Data Encryption Standard). Cette attaque permet de retrouver la clé secrète utilisée lors du chiffrement, en comparant la sortie correcte avec des sorties erronées provoquées volontairement.

Question 1 : Principe de l'attaque par fautes sur le DES

Le principe d'une attaque par fautes appliquée au DES consiste à injecter une faute dans l'algorithme de chiffrement, de manière contrôlée, afin d'obtenir un message chiffré erroné que l'on compare à un chiffrement correct. Dans notre cas, l'attaque cible la sortie R_{15} du 15$^{\text{e}}$ tour.

L'algorithme DES est constitué de 16 tours organisés selon la structure de Feistel. Chaque tour transforme les moitiés gauche et droite (L_i, R_i) du message intermédiaire selon les équations suivantes :

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

où K_i est la sous-clé du i^{e} tour, dérivée de la clé principale K, et f est définie par :

$$f(R, K_i) = P\left(S\left(E(R) \oplus K_i\right)\right)$$

avec :

— E : fonction d'expansion de 32 à 48 bits,

— S : fonction de substitution composée de 8 S-boxes (6 bits $\rightarrow$ 4 bits),

— P : permutation finale.

L'attaque par faute repose sur la comparaison entre deux sorties :

— C : résultat du chiffrement normal, soit R_{16} à la sortie du dernier tour.

— C' : résultat du chiffrement après injection d'une faute sur R_{15}.

À partir de ces deux sorties, on exploite la différence :

$$R_{16} \oplus R'_{16} = f(R_{15}, K_{16}) \oplus f(R'_{15}, K_{16})$$

En développant f :

$$P^{-1}(R_{16} \oplus R'_{16}) = S(E(R_{15}) \oplus K_{16}) \oplus S(E(R'_{15}) \oplus K_{16})$$

On obtient alors une équation sur les sorties des S-boxes, permettant d'attaquer K_{16}, la sous-clé du dernier tour. Cette relation permet de remonter à certains bits de K_{16} en testant toutes les combinaisons de 6 bits possibles pour chaque S-box affectée.

La position de la faute est déduite à partir de la différence $R_{15} \oplus R'_{15}$. Cela indique quelle(s) S-box(es) ont été touchées. En analysant les bits de $P^{-1}(R_{16} \oplus R'_{16})$ correspondant à ces S-boxes, on déduit les fragments de clé K_{16} responsables de la différence.

Une recherche exhaustive (au plus $2^6 = 64$ possibilités par S-box) permet de déterminer, pour chaque S-box touchée, le fragment de 6 bits de K_{16} vérifiant l'équation. En répétant cela sur plusieurs fautes différentes, on croise les résultats et on garde le seul fragment de clé qui est compatible avec toutes les fautes : cela donne 6 bits fiables par S-box.

Avec suffisamment de fautes (ici 32 fautes ont été utilisées), il est possible de reconstruire entièrement K_{16}, la sous-clé de 48 bits utilisée au dernier tour de DES. Ensuite, il sera possible de remonter à la clé complète K.

Question 2 : Application concrète de l'attaque

Nous appliquons ici la méthode décrite à la question précédente sur les données fournies :

— **Message clair :** `AE2AD5A74FF82634`

— **Message chiffré juste :** `52E4186D3632B556`

— **Messages chiffrés fautés :** 32 messages obtenus par injection de fautes sur R_{15}.

L'analyse porte sur chaque paire (C, C') formée du message chiffré juste C et d'un message fauté C'. Pour chacune, on applique l'équation :

$$P^{-1}(R_{16} \oplus R'_{16}) = S(E(R_{15}) \oplus K_{16}) \oplus S(E(R'_{15}) \oplus K_{16})$$

Chaque différence $R_{15} \oplus R'_{15}$ indique quelle S-box a été impactée. En analysant la différence $R_{16} \oplus R'_{16}$ (après inversion de la permutation P), on obtient la sortie XOR des deux S-box correspondantes. On effectue alors une recherche exhaustive sur les 64 clés de 6 bits possibles pour la S-box concernée.

Cette recherche donne, pour chaque faute et pour chaque S-box atteinte, un ou plusieurs candidats possibles. En recoupant les résultats sur les 32 fautes, on isole pour chaque S-box le seul fragment de 6 bits qui est compatible avec toutes les fautes où elle est impliquée. On reconstitue ainsi la sous-clé K_{16} en concaténant ces fragments.

Dans notre cas, les fragments trouvés pour chaque S-box sont les suivants :

```
S-box 0: fragment_val=63 (max_count=9)
S-box 1: fragment_val=51 (max_count=9)
S-box 2: fragment_val=13 (max_count=8)
S-box 3: fragment_val=32 (max_count=6)
S-box 4: fragment_val=30 (max_count=5)
S-box 5: fragment_val=58 (max_count=3)
S-box 6: fragment_val=58 (max_count=2)
S-box 7: fragment_val=37 (max_count=2)
```

On reconstruit alors la sous-clé K_{16} en concaténant les 8 fragments (6 bits chacun) :

K16 (binaire) : `111111100110011011000000111101110101110101001 01`

En hexadécimal, cela correspond à :

K16 (hex) : `FF33607BAEA5`

Ainsi, à partir des 32 messages fautés, nous avons pu retrouver avec succès les 48 bits de la sous-clé K_{16}. Cette étape constitue la base de l'étape suivante, qui consiste à retrouver la clé complète de 56 bits ayant généré K_{16}.

Question 3 : Reconstruction de la clé complète

À la fin de la question 2, on connaît la sous-clé du 16$^{\text{e}}$ tour :

$$K_{16} = \texttt{FF33 607B AEA5}_{16} = 11111111\,00110011\,01100000\,01111011\,10101110\,10100101_2.$$

1. Remonter K_{16} vers une clé 56 bits

La permutation PC_2 écarte huit positions (14, 15, 19, 20, 51, 54, 58 et 60). En appliquant son inverse on obtient :

$$PC_2^{-1}(K_{16}) = \underbrace{11101110}_{1\text{–}8}\ 0000\_\_\_\_\ 0111\_\_\_\_\ 01101011\,01100111\,001001\_\ \_\ 100$$

Les huit traits « _ » signalent exactement les bits manquants ; on dispose donc de 48 bits connus + 8 inconnus.

2. Recherche exhaustive sur les 8 bits

1) On boucle sur les $2^8 = 256$ valeurs possibles des bits absents.
2) Pour chaque essai, on insère ces huit bits et on obtient ainsi une clé effective K_{56} de 56 bits.
3) On régénère ensuite les 16 sous-clés à partir de K_{56} (permutations PC_1, décalages, puis PC_2).
4) Enfin, on chiffre le message clair $\texttt{AE2AD5A74FF82634}_{16}$ et l'on garde la clé qui produit le chiffre attendu $\texttt{52E4186D3632B556}_{16}$.

Un seul candidat passe ; on trouve :

$$K_{56} = \texttt{EE0E72AAD6CE4AE4}_{16}.$$

3. Ajout des bits de parité

On découpe K_{56} en huit blocs de 7 bits et on ajoute un bit de parité impair à droite de chacun ; on obtient :

$$K_{64} = \texttt{EF0E73ABD6CE4AE5}_{16}.$$

4. Vérification

$$K_{64} = \texttt{EF0E73ABD6CE4AE5}_{16}$$

$$E_{K_{64}}\big(\texttt{AE2AD5A74FF82634}\big) = \texttt{52E4186D3632B556}_{16} \quad \Longrightarrow \quad \text{clé correcte.}$$

Complexité : seulement $2^8 = 256$ DES, soit quelques millisecondes.

Trace d'exécution. Voici le résultat obtenu lors de l'exécution du code Java ; les deux captures ci-dessous reprennent la sortie console complète.

```
--- Test DES Standard ---
Test 1:
Clair : 0123456789abcdef
Clé   : 133457799bbcdff1
Chiffré Attendu: 85e813540f0ab405
Chiffré Calculé: 85e813540f0ab405
Test 1 DES: SUCCÈS
--- Fin Test DES ---

Fragments K16 (valeur 0-63) et leur max_count:
S-box 0: fragment_val=63 (max_count=9)
S-box 1: fragment_val=51 (max_count=9)
S-box 2: fragment_val=13 (max_count=8)
S-box 3: fragment_val=32 (max_count=6)
S-box 4: fragment_val=30 (max_count=5)
S-box 5: fragment_val=58 (max_count=3)
S-box 6: fragment_val=58 (max_count=2)
S-box 7: fragment_val=37 (max_count=2)
K16 binaire construite: 111111110011001101100000011110111010111010100101
K16 trouvée (hex) : ff33607baea5
Clé 64 bits après PC1_MOIN_1 (avant bruteforce des 8 bits):
```

```
Clé 64 bits après PC1_MOIN_1 (avant bruteforce des 8 bits):
11101110 00001000 01000010 10101010 11010110 11001110 01001010 10100100
Clé 56-bit (effective) trouvée (avant ajustement parité)! Hex: ee0e72aad6ce4ae4
Clé avant ajustement parité (binaire):
11101110 00001110 01110010 10101010 11010110 11001110 01001010 11100100
Clé après ajustement parité (binaire):
11101111 00001110 01110011 10101011 11010110 11001110 01001010 11100101
Clé finale 64 bits (avec parité) : ef0e73abd6ce4ae5
Test avec clé trouvée: Chiffrement de ae2ad5a74ff82634 avec ef0e73abd6ce4ae5 -> 52e4186d3632b556
VÉRIFICATION FINALE: SUCCÈS! La clé trouvée chiffre correctement le message.
```

FIGURE 1 – Sortie du programme : récupération de K_{16} puis de la clé finale `EF0E73ABD6CE4AE5`.

Conclusion : à partir de la sous-clé K_{16} obtenue par l'analyse différentielle des fautes, on retrouve la clé maître du DES sans coût exponentiel significatif.

Question 4 : Fautes sur les tours précédents (R14, R13...)

Dans les questions précédentes, nous avons supposé que la faute est injectée sur la sortie de R_{15}, c'est-à-dire juste avant le dernier tour du DES. Cette hypothèse simplifie fortement l'attaque, car seule la sous-clé K_{16} intervient dans la différence observable à la sortie du chiffrement.

Nous examinons ici ce qu'il se passerait si la faute était injectée plus tôt, par exemple sur la sortie de R_{14} (14^{e} tour), ou encore plus en amont.

Propagation de la faute dans un réseau de Feistel

Lorsque la faute est introduite en R_{14}, elle affecte non seulement R_{15}, mais aussi tous les tours suivants, car :

$$L_{i+1} = R_i \quad ; \quad R_{i+1} = L_i \oplus f(R_i, K_i)$$

La faute se propage donc dans les deux moitiés du message, et à travers les fonctions non-linéaires de chiffrement. Ainsi, à chaque tour, la diffusion rend plus difficile l'identification de l'effet direct de la faute.

Pour illustrer cela, supposons que l'on note e la faute binaire injectée :

— En R_{15} : on peut isoler directement son effet sur R_{16} et attaquer K_{16}.

— En R_{14} : la faute se propage dans R_{15}, puis R_{16}, rendant l'équation de l'attaque plus complexe. Il faut connaître à la fois K_{15} et K_{16} pour reconstituer l'effet de la faute.

— En R_{13} ou avant : le nombre de sous-clés impliquées augmente (au moins K_{14}, K_{15}, K_{16}), ce qui rend l'analyse combinatoire exponentiellement plus difficile.

Complexité de l'attaque en fonction du tour

— **Tour 15** : Complexité $\sim 2^{12}$ (analyse S-box indépendante).

— **Tour 14** : Il faut deviner K_{15} pour retrouver R_{15}, puis attaquer K_{16}. Complexité $\sim 2^{48}$ (si on ne peut pas isoler K_{15} facilement).

— **Tour 13** : Propagation sur 3 tours, plus de 96 bits de sous-clés impliqués. Complexité $\gg 2^{56}$.

À partir du tour 13, l'attaque n'est plus réaliste dans un cadre raisonnable de puissance de calcul. Cela est dû à l'effet cumulatif de la propagation de la faute dans les transformations non linéaires successives du DES.

Conclusion

Une attaque par faute reste réaliste jusqu'au 14^{e} tour, mais au-delà, l'effet de diffusion rend l'identification des sous-clés responsables quasi impossible sans information supplémentaire. Le tour 15 est donc le point optimal et stratégique pour une attaque par faute efficace sur le DES.

Question 5 : Contre-mesures contre l'attaque par fautes

Pour répondre à la question, je présente deux contre-mesures faciles à mettre en place, puis j'en estime le surcoût :

1. **Chiffrement *puis* déchiffrement** : on refait le calcul à l'envers pour vérifier qu'aucune faute ne s'est glissée ;
2. **Masquage interne aléatoire** : on cache les états intermédiaires derrière des masques XOR tirés au hasard.

1. Chiffrer puis déchiffrer

Principe. On chiffre le bloc clair M :

$$C = E_K(M),$$

puis on le déchiffre dans la foulée :

$$M^\star = D_K(C).$$

Si $M^\star \neq M$, on lève une alarme : la faute est détectée.

Coût. On exécute deux DES complets + une comparaison de 64 bits (négligeable) :

$$T_{\text{protégé}} \approx 2\,T_{\text{DES}} \quad \Longrightarrow \quad \gamma \;=\; \frac{T_{\text{protégé}}}{T_{\text{DES}}} \;\approx\; 2.$$

Mémoire. Juste un registre temporaire de 64 bits.

2. Masquage booléen (1er ordre)

Principe. Avant le tour 0, on tire deux masques aléatoires de 32 bits :

$$M_L,\, M_R \;\leftarrow\; \{0,1\}^{32}.$$

On travaille ensuite sur les paires masquées :

$$\widetilde{L}_0 = L_0 \oplus M_L, \quad \widetilde{R}_0 = R_0 \oplus M_R,$$

puis, pour chaque tour i ($1 \le i \le 16$) :

$$\begin{aligned} \widetilde{L}_i &= \widetilde{R}_{i-1}, \\ \widetilde{R}_i &= \widetilde{L}_{i-1} \;\oplus\; f\big(\widetilde{R}_{i-1} \oplus M_R,\; K_i\big) \;\oplus\; M_L. \end{aligned}$$

Ainsi, en enlevant le masque on retrouve bien (L_i, R_i) à chaque tour.

Coût. Par tour, on ajoute deux XOR 32 bits et on remplace les 8 S-boxes par des versions « pré-masquées ». Dans la pratique (implémentation table-lookup), on compte grosso-modo le même temps qu'une S-box originale ; d'où :

$$T_{\text{protégé}} \;\approx\; 1.5 \text{ à } 2\,T_{\text{DES}} \qquad \Longrightarrow \qquad \gamma \approx 1.5\text{–}2.$$

Mémoire. Deux registres de 32 bits (les masques) + tables S agrandies (dépend de l'optimisation).

Conclusion La redondance est simple à coder mais coûte « $\times 2$ » à coup sûr. Le masquage peut être un peu plus léger (surtout si on optimise les tables), mais demande un bon générateur aléatoire et du soin dans le code bas-niveau pour ne pas réintroduire de biais.

Conclusion

Cette attaque permet de retrouver la clé secrète du DES avec seulement 32 messages fautés, en exploitant une simple perturbation matérielle. Elle démontre la fragilité de DES face à des attaques physiques, même si sa structure reste un excellent support pédagogique pour l'étude de la cryptanalyse. Le code source complet, entièrement implémenté en **Java**, est disponible dans le dépôt GitHub suivant : github.com/Maryem-bencharke/Des-Fault-Attack