Université de Versailles Saint Quentin en Yvelines

l'UFR des Sciences

Rapport Contrôle Continu

Calcul Sécurisé

Spécialité : M1 INFO Secrets

Attaque par faute sur le DES

---

https://github.com/Abdelalim03/DES-fault-attack

Realisé par :

Abdelalim Atoui

22405637

Abdelalim.atoui@ens.uvsq.fr

Encadré par :

Dr. Louis Goubin

2024/2025

Table des matières

Table des figures

List of Algorithms

1 Attaque par faute sur le DES

Le **Data Encryption Standard (DES)** est un algorithme de chiffrement symétrique normalisé en 1977 par le NIST (National Institute of Standards and Technology), largement utilisé dans les années 80 et 90. Il repose sur un chiffrement en bloc de 64 bits avec une clé effective de 56 bits, et suit une structure de type Feistel comportant 16 tours. Le fonctionnement de chaque tour repose sur des permutations, substitutions (via des S-boxes), et sur l'application d'une sous-clé dérivée de la clé principale.

Malgré sa robustesse apparente à l'époque de sa standardisation, le DES souffre aujourd'hui de plusieurs vulnérabilités cryptographiques. Sa faiblesse principale réside dans la longueur de sa clé, rendant possible une attaque par *recherche exhaustive* (brute force) nécessitant au plus 2^{56} essais, ce qui est réalisable avec les moyens informatiques modernes.

Une autre classe d'attaques plus efficaces contre le DES repose sur l'**injection de fautes** pendant l'exécution du chiffrement, et c'est précisément ce type d'attaque que nous allons étudier dans ce rapport.

L'attaque par faute sur le DES repose sur l'idée d'introduire une erreur contrôlée durant l'exécution de l'algorithme, afin d'observer la différence entre le résultat correct et les résultats erronés. Ces différences peuvent révéler des informations sur la clé secrète utilisée.

Dans ce cas précis, on suppose que l'attaquant est capable d'injecter une faute dans la valeur de sortie R_{15} du 15$^{\text{e}}$ tour. Rappelons que dans le DES, chaque tour i (pour i allant de 1 à 16) transforme une paire (L_i, R_i) en (L_{i+1}, R_{i+1}) selon les formules suivantes :

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

où f est une fonction non-linéaire, et K_i est la sous-clé du tour i.

Principe de l'attaque

1. L'attaquant chiffre un message clair M et obtient un message chiffré correct C.
2. Ensuite, il provoque une ou plusieurs fautes au niveau de R_{15} (sortie du 15$^{\text{e}}$ tour), ce qui modifie la valeur finale chiffrée C'.
3. En comparant C et C' (donc sans connaître la clé), l'attaquant peut remonter à l'effet de la faute sur R_{16}, et donc à $f(R_{15}, K_{16})$.
4. Comme $R_{16} = L_{15} \oplus f(R_{15}, K_{16})$, une différence dans R_{15} implique une différence dans $f(R_{15}, K_{16})$.
5. Grâce aux sorties fautées, on peut remonter au niveau de l'entrée de la permutation finale, et en inversant la permutation, obtenir des informations différentielles utiles sur les entrées/sorties des S-boxes dans f.

6. En testant toutes les valeurs possibles de sous-clé K_{16} pour chaque S-box (6 bits), et en comparant avec les différences obtenues dans les résultats fautés, l'attaquant peut retrouver les 6 bits de K_{16} responsables du comportement fauté dans chaque S-box.

Résultat de l'attaque

Une fois l'attaque répétée sur suffisamment de fautes, l'attaquant peut récupérer les 48 bits de la sous-clé K_{16}. Puis, comme les sous-clés sont dérivées de la clé principale de 56 bits par une permutation et rotation, on peut remonter partiellement à cette dernière.

Ainsi, cette attaque permet de compromettre efficacement une partie significative de la clé DES, avec des moyens limités (seulement besoin de provoquer des fautes sur un tour précis).

Comparaison avec la recherche exhaustive : contrairement à l'attaque par force brute qui demande 2^{56} essais dans le pire des cas, l'attaque par faute présentée ici permet de récupérer une portion substantielle de la clé avec bien moins de calculs, à condition de pouvoir injecter des fautes contrôlées. Cela en fait une attaque plus puissante dans le cadre de l'analyse différentielle par faute.

Dans la partie suivante, nous allons démontrer concrètement cette attaque en utilisant des données fournies, et implémenter les étapes critiques en **Go (Golang)** pour en évaluer la complexité et l'efficacité réelle.

2 Application concrète

Dans cette partie, nous appliquons concrètement l'attaque par fautes sur DES à partir des données fournies dans le fichier `enonce-cc.txt`. Celui-ci contient :

— Le message clair utilisé pour toutes les exécutions du chiffrement ;
— Le message chiffré sans faute (**chiffré juste**) ;
— 32 messages chiffrés obtenus après l'injection de fautes sur la sortie du 15^{e} tour (**chiffrés fautés**) ;

Les 33 exécutions utilisent le même message clair et la même clé secrète.

2.1 Méthodologie pour retrouver la clé K_{16}

Le but de cette attaque est de retrouver les 48 bits de la sous-clé K_{16} utilisée au 16^{e} tour de l'algorithme DES. Pour cela, nous avons exploité les différences entre un texte chiffré correct C et plusieurs textes chiffrés fautés C'_i, obtenus en injectant des fautes dans l'état intermédiaire du DES (plus précisément dans R_{15}). Voici les étapes de notre démarche :

1. Conversion et préparation : Tous les messages sont fournis en hexadécimal. Ils sont convertis en tableaux d'octets, puis transformés en chaînes binaires pour pouvoir appliquer les différentes permutations et opérations bit à bit.

2. Application de la permutation initiale (IP) : On applique l'inverse de la permutation finale du DES (qui est en fait la permutation initiale IP) sur chaque texte chiffré (correct ou fauté). Cela permet d'obtenir les deux moitiés de l'état final :

$$\mathrm{IP}(C) = L_{16} \,||\, R_{16}, \quad \mathrm{IP}(C'_i) = L_{16}^{(i)} \,||\, R_{16}^{(i)}$$

En exploitant la structure du Feistel, on sait que :

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

Et de même pour le chiffré fauté :

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

3. Élimination de L_{15} et déduction de la différence sur f : On soustrait les deux équations précédentes :

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

Cela nous donne l'équation fondamentale de notre attaque :

$$\Delta f = \Delta R_{16}$$

Puisque la fonction f comprend l'expansion E, l'opération XOR avec K_{16}, les S-boxes S_j, et la permutation P, on peut exprimer :

$$f(R_{15}, K_{16}) = P\left(S\left(E(R_{15}) \oplus K_{16}\right)\right)$$

Avec :

$$S = S_1||S_2||\ldots||S_8$$

4. Inversion de la permutation P : On applique P^{-1} à la différence ΔR_{16} afin d'obtenir la différence en sortie des S-boxes :

$$\Delta_{\text{S-boxes}} = P^{-1}(\Delta R_{16})$$

5. Génération d'équations différentielles pour chaque S-box : La sortie de la permutation inversée nous fournit 8 blocs de 4 bits chacun, soit une équation par S-box :

$$\begin{cases} \Delta_1 = S_1\left(E(R_{15})_1 \oplus K_1\right) \oplus S_1\left(E(R_{15}^{(i)})_1 \oplus K_1\right) \\ \Delta_2 = S_2\left(E(R_{15})_2 \oplus K_2\right) \oplus S_2\left(E(R_{15}^{(i)})_2 \oplus K_2\right) \\ \quad \vdots \\ \Delta_8 = S_8\left(E(R_{15})_8 \oplus K_8\right) \oplus S_8\left(E(R_{15}^{(i)})_8 \oplus K_8\right) \end{cases}$$

Chaque Δ_j (4 bits) est déduit de $P^{-1}(\Delta R_{16})$.

6. Recherche exhaustive par S-box : Pour chaque S-box S_j affectée (i.e., $\Delta_j \neq 0000$), on teste les 64 clés possibles (6 bits) :

$$\text{Si } S_j(x \oplus k) \oplus S_j(x' \oplus k) = \Delta_j \quad \text{alors } k \text{ est une candidate valide}$$

On répète cela pour chaque faute, et on conserve l'intersection des clés valides.

7. Construction finale de la sous-clé K_{16} : Une fois les sous-clés partielles des 8 S-boxes identifiées (6 bits chacune), on les concatène :

$$K_{16} = K_1||K_2||\ldots||K_8 \quad \text{avec } K_j \in \{0,1\}^6$$

L'ensemble de cette méthode a été implémenté en langage Go, en exploitant les tables du DES (E, P, IP, S-boxes) et en automatisant l'analyse de toutes les fautes via un script.

Algorithme simplifié de récupération de K_{16}

Algorithm 1 Récupération de K_{16} à partir des chiffrés fautés

Require : Texte chiffré correct C, liste de chiffrés fautés $\{C'_i\}$, S-boxes

Ensure : K_{16} en binaire

$R_{16}, L_{16} \leftarrow \mathrm{IP}(C)$
$R_{15} \leftarrow L_{16}$
$E_{R_{15}} \leftarrow E(R_{15})$
for all C'_i dans la liste **do**
    $R_{16}^{(i)}, L_{16}^{(i)} \leftarrow \mathrm{IP}(C'_i)$
    $R_{15}^{(i)} \leftarrow L_{16}^{(i)}$
    $E_{R_{15}^{(i)}} \leftarrow E(R_{15}^{(i)})$
    $\Delta R_{16}^{(i)} \leftarrow R_{16} \oplus R_{16}^{(i)}$
    $\Delta_{\text{S-boxes}} \leftarrow P^{-1}(\Delta R_{16}^{(i)})$
    for $j \leftarrow 1$ to 8 **do**
        if $\Delta_j \neq 0000$ **then**
            for all clés k de 0 à 63 **do**
                if $S_j(E^j_{R_{15}} \oplus k) \oplus S_j(E^j_{R_{15}^{(i)}} \oplus k) = \Delta_j$ **then**
                    Ajouter k à la liste des clés possibles pour S_j
                end if
            end for
        end if
    end for
end for
for $j \leftarrow 1$ to 8 **do**
    $K_j \leftarrow$ intersection des clés valides sur toutes les fautes
end for
return $K_{16} = K_1 || K_2 || \ldots || K_8$

2.2 Résultat : 48 bits de sous-clé K_{16} retrouvés

Les bits de la sous-clé K_{16} récupérés sont les suivants :

<111101011111101101110010101101100011110001010011>

Et en hexadécimal :

<F5FB72B63C53>

Ce résultat constitue une estimation fiable de la sous-clé K_{16}, issue de l'analyse différentielle menée sur les S-boxes affectées.

Nous allons maintenant utiliser cette sous-clé pour tenter de retrouver les 56 bits de la clé principale dans la section suivante.

3 Retrouver la clé complète du DES

3.1 Complément des 8 bits manquants à partir de K_{16}

À partir de l'attaque précédente, nous avons pu retrouver les 48 bits de la sous-clé K_{16}. Cette sous-clé est issue de la clé principale de 56 bits à l'aide de la permutation $PC2$. Cependant, la permutation $PC2$ ne sélectionne que 48 bits parmi les 56 bits de la clé initiale. Il nous manque donc 8 bits pour reconstituer la clé complète.

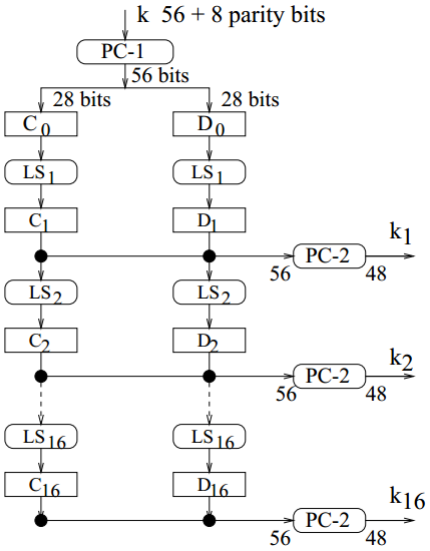

FIGURE 1 – Génération des sous-clés du DES à partir de la clé principale.

Comme le montre la figure ci-dessus, à chaque tour du DES, les registres C_i et D_i sont modifiés par des décalages à gauche. Mais la somme des décalages cumulés au bout de 16 tours est égale à 28, ce qui revient à un tour complet (rotation sur 28 bits). Ainsi, on a :

$$C_{16} = C_0 \quad \text{et} \quad D_{16} = D_0$$

Par conséquent, la sous-clé K_{16} est obtenue en appliquant $PC2$ à $C_0||D_0$:

$$K_{16} = PC2(C_0||D_0)$$

On applique l'inverse de $PC2$ (notée $PC2^{-1}$) sur K_{16} pour retrouver les 56 bits partiels. Cependant, la permutation $PC2$ ne sélectionne que 48 bits sur les 56 bits de la clé initiale. Il nous manque donc 8 bits pour reconstituer la clé complète.

Méthode : Pour retrouver la clé de 56 bits :

1. Identifier les 8 positions de bits (sur les 56) qui ne sont pas utilisés par $PC2$;
2. Générer toutes les combinaisons possibles des 8 bits manquants (soit $2^8 = 256$ possibilités) ;
3. Pour chaque combinaison, injecter les bits manquants dans la clé partielle, et appliquer la permutation inverse $PC1^{-1}$ afin d'obtenir les 64 bits avec bits de parité ;
4. Ajouter les bits de parité : chaque octet de la clé DES finale doit contenir un nombre impair de bits à 1 ;
5. Tester chaque clé complète sur le texte clair initial : seule la bonne clé donnera le bon texte chiffré.

Parité DES : Conformément au standard `FIPS 46-3`, la clé DES fait 64 bits, dont 56 bits utilisés par l'algorithme. Les 8 bits restants sont des bits de parité, placés en fin de chaque octet pour garantir une **parité impaire** (nombre impair de bits à 1 dans chaque octet).

Voici un pseudocode résumant les étapes principales de la fonction `RecoverMainKey` :

Algorithm 2 Récupération de la clé complète DES à partir de K_{16}

Require : K_{16} : 48 bits de la sous-clé du dernier tour
Ensure : Clé DES complète (64 bits) avec bits de parité

```
Identifier les 8 bits manquants non sélectionnés par la permutation PC2
for all combinaisons possibles des bits manquants (256 au total) do
    Appliquer PC2^-1 pour reconstruire la clé partielle de 56 bits
    Insérer les bits manquants à leur position respective
    Appliquer PC1^-1 pour passer de 56 à 64 bits
    Ajouter les bits de parité (1 bit par octet) pour assurer la parité impaire
    Tester la clé obtenue : chiffrer le texte clair initial
    if le résultat correspond au chiffré correct then
        return clé valide (64 bits)
    end if
end for
return aucune clé trouvée si toutes les combinaisons échouent
```

3.2 Récupération automatisée avec validation cryptographique

Nous avons automatisé cette procédure à l'aide d'un script Go. La fonction `RecoverMainKey(K16)` parcourt toutes les combinaisons possibles des bits manquants, et pour chaque clé candidate, effectue les étapes suivantes :

1. Reconstruction des 56 bits de la clé initiale par `ReversePC2` ;
2. Application de `ReversePC1` pour obtenir les 64 bits ;
3. Ajout des bits de parité dans chaque octet (`addParityBits`) ;
4. Test de chiffrement : la clé est retenue si elle chiffre le texte clair initial vers le bon chiffré.

Clé finale retrouvée : Après exécution du code, nous avons retrouvé la **clé DES complète**, représentée en hexadécimal sur 8 octets :

`E5 F7 D5 FE DA 43 67 26`

Chaque octet vérifie bien la parité impaire, conformément à la spécification du DES.

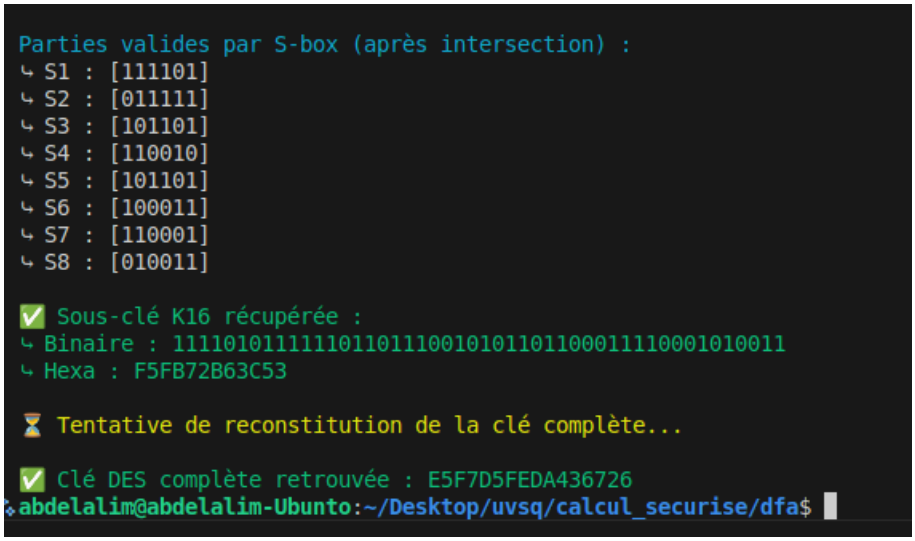

FIGURE 2 – Resultat donné par notre script en GO

Ce processus reste très efficace, car l'espace de recherche est limité à seulement 256 clés.

3.3 Analyse de complexité de l'attaque complète

L'attaque différentielle par injection de fautes exploitée ici permet de retrouver efficacement la clé complète du DES en deux phases. Sa complexité peut être analysée comme suit :

Phase 1 — Récupération de la sous-clé K_{16} (48 bits) Pour chaque S-box S_j, on effectue une recherche exhaustive sur les 64 valeurs possibles (6 bits) de clé locale. Seules les S-boxes pour lesquelles une différence de sortie est détectée (i.e., $\Delta_j \neq 0000$)

sont analysées. En moyenne, une faute affecte environ 3 à 5 S-boxes. Si l'on dispose de f chiffrés fautés, la complexité est donc :

$$\mathcal{O}(f \cdot s \cdot 64) \quad \text{avec } s \leq 8$$

Dans notre cas, on a trente-deux fautes qui suffisent pour obtenir une intersection non vide pour chaque S-box, ce qui permet de reconstruire K_{16} en quelques secondes, donc au maximum :

$$\mathcal{O}(2^{14})$$

Phase 2 — Reconstruction de la clé complète (56 puis 64 bits) La permutation $PC2$ n'utilise que 48 des 56 bits initiaux, laissant 8 bits indéterminés. On effectue alors une recherche exhaustive sur ces 8 bits, soit :

$$2^8 = 256 \text{ clés candidates}$$

Pour chaque candidate, on applique $PC1^{-1}$, on ajoute les bits de parité pour obtenir une clé DES de 64 bits, puis on chiffre le texte clair initial pour comparer avec le chiffré correct. Cette étape a une complexité constante :

$$\mathcal{O}(256)$$

Conclusion L'attaque complète présente une complexité très faible par rapport à une attaque exhaustive (2^{56}) sur l'espace total de clés. Grâce à la structure de Feistel et aux propriétés des permutations P, $PC2$ et $PC1$, la clé DES peut être retrouvée avec environ 32 fautes et 256 essais finaux, ce qui rend cette attaque réalisable en quelques secondes sur une machine standard.

4 Attaques avec fautes sur les tours précédents

4.1 Faute sur la sortie R_{14}

Dans les questions précédentes, nous avons supposé que la faute était injectée sur la sortie R_{15}. Explorons maintenant le cas où la faute est injectée plus tôt, en particulier à la sortie du 14$^{\text{e}}$ tour.

Rappels sur le fonctionnement du DES :

$$\begin{cases} R_{15} = L_{14} \oplus f(K_{15}, R_{14}) \\ L_{15} = R_{14} \end{cases} \qquad \begin{cases} R'_{15} = L_{14} \oplus f(K_{15}, R'_{14}) \\ L'_{15} = R'_{14} \end{cases}$$

On en déduit :

$$R_{15} \oplus R'_{15} = f(K_{15}, R_{14}) \oplus f(K_{15}, R'_{14})$$

Lien avec les sorties du 16$^{\text{e}}$ tour : Or, on peut exprimer R_{14} et R'_{14} en fonction des sorties du 16$^{\text{e}}$ tour :

$$R_{14} = L_{15} = R_{16} \oplus f(K_{16}, L_{16}) \quad \Rightarrow \quad P^{-1}(L_{15} \oplus R_{16}) = S_i(E(L_{16}) \oplus K_{16})$$

$$R'_{14} = L'_{15} = R'_{16} \oplus f(K_{16}, L'_{16}) \quad \Rightarrow \quad P^{-1}(L'_{15} \oplus R'_{16}) = S_i(E(L'_{16}) \oplus K_{16})$$

Ainsi, on peut mener une attaque différentielle standard pour retrouver K_{16}, exactement comme on l'a fait pour K_{16} dans la question précédente, avec une complexité de :

$$\boxed{\mathcal{O}(2^{14})}$$

Approche de l'attaque :

1. On applique une attaque différentielle sur K_{16} via les sorties $R_{16}, L_{16}, R'_{16}, L'_{16}$;
2. Pour chaque K_{16} candidat, on déduit :

$$R_{14} = R_{16} \oplus f(K_{16}, L_{16}), \quad R'_{14} = R'_{16} \oplus f(K_{16}, L'_{16})$$

3. On applique ensuite une attaque différentielle sur K_{15}, en utilisant les équations :

$$R_{15} \oplus R'_{15} = f(K_{15}, R_{14}) \oplus f(K_{15}, R'_{14})$$

4. Chaque attaque sur une sous-clé (par S-box) nécessite environ 2^{14} essais.

Complexité totale : L'attaque nécessite donc deux phases imbriquées :

— On effectue une attaque différentielle pour retrouver les bits de K_{16} avec une complexité de $\mathcal{O}(2^{14})$;

— Pour chaque candidat possible de K_{16} c'est à dire L_{15} et L_{15}, on applique une attaque sur K_{15} avec une complexité de $\mathcal{O}(2^{14})$ également.

Ce qui donne une complexité globale de type produit :

$$\boxed{\mathcal{O}(2^{14}) \times \mathcal{O}(2^{14}) = \mathcal{O}(2^{28})}$$

Cela reste raisonnable dans un cadre académique ou sur des machines modernes.

4.2 Faute sur la sortie R_i pour $i < 14$

Si la faute est injectée sur un tour plus ancien, il faut reconstituer encore plus d'états intermédiaires. À chaque niveau, une attaque différentielle est nécessaire pour retrouver une sous-clé supplémentaire, avec un coût de $\mathcal{O}(2^{14})$ à chaque fois.

Par exemple, pour une faute sur R_{13} :

— On attaque d'abord K_{16}, puis K_{15}, puis K_{14}

— Chaque étape utilise les résultats de la précédente

Complexité cumulée par tour :

— Faute sur R_{15} : $\mathcal{O}(2^{14})$

— Faute sur R_{14} : $\mathcal{O}(2^{28})$

— Faute sur R_{13} : $\mathcal{O}(2^{42})$

— Faute sur R_{12} : $\mathcal{O}(2^{56})$

— Faute sur R_{11} : $\mathcal{O}(2^{70})$

— Faute sur R_{10} : $\mathcal{O}(2^{84})$

4.3 Jusqu'à quel tour l'attaque est-elle réaliste ?

On considère qu'une attaque est réaliste si sa complexité reste inférieure à 2^{80}. Dans ce cadre :

— Les attaques sur R_{15}, R_{14}, R_{13} sont **réalistes** ;

— L'attaque sur R_{12} est encore envisageable dans des environnements puissants ou distribués ;

— À partir de R_{11}, la complexité dépasse 2^{80}, rendant l'attaque **non praticable**.

5 Contre-mesures contre les attaques par fautes

Pour se protéger contre les attaques par injection de fautes sur DES, il existe plusieurs approches possibles. On peut soit agir au niveau logiciel pour détecter les erreurs, soit intervenir au niveau matériel pour empêcher qu'elles aient lieu. Voici les principales stratégies envisageables, avec une réflexion sur leur impact.

5.1 Contre-mesures logicielles (ou algorithmiques)

La méthode la plus simple consiste à demander au système de vérifier lui-même que le résultat du chiffrement est cohérent. Une technique classique est de chiffrer deux fois le même bloc de données, avec la même clé, puis de comparer les deux résultats. Si une erreur est survenue lors de l'un des deux traitements, les deux chiffrés ne correspondront pas, et la carte pourra refuser de répondre.

Ce type de protection est particulièrement utile contre les fautes aléatoires, car il est très improbable qu'une même erreur affecte les deux exécutions de manière identique. L'inconvénient est qu'on double le temps de chiffrement, mais cela reste acceptable dans la plupart des cas, surtout qu'on utilise souvent le triple DES dans les applications réelles.

D'autres variantes existent, comme le fait de recalculer seulement une partie des étapes pour économiser du temps, ou encore de vérifier que le message chiffré redonne bien le texte clair après déchiffrement. Dans tous les cas, cela implique un coût supplémentaire, mais raisonnable.

5.2 Contre-mesures matérielles

Une autre approche consiste à empêcher physiquement l'injection de fautes. Cela peut se faire en ajoutant des capteurs pour surveiller l'environnement matériel : si la température, la tension ou la fréquence devient anormale, le système peut automatiquement se mettre en pause ou refuser de continuer.

Ce genre de protection a l'avantage de ne pas ralentir le chiffrement, mais elle demande une conception matérielle plus complexe et plus coûteuse. Elle est donc moins répandue dans les systèmes classiques, mais reste pertinente dans les environnements sensibles.

Conclusion

En résumé, les contre-mesures algorithmiques sont simples à mettre en place et déjà efficaces dans la majorité des cas, au prix d'une légère perte de performance. Les solutions physiques, quant à elles, sont plus robustes mais aussi plus coûteuses. Une combinaison des deux types de protections offre généralement le meilleur compromis entre sécurité et efficacité.