Calcul sécurisé - Attaque par faute sur DES

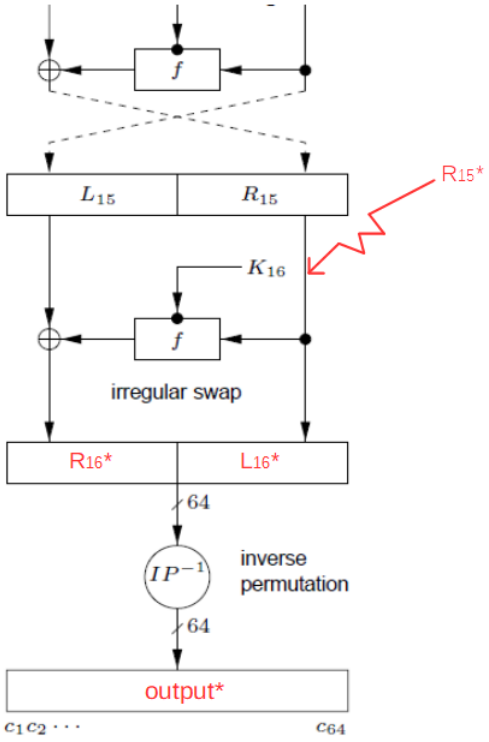
CAUMES Clément 21501810

Master 1 Informatique SeCReTs

Table des matières

1 Partie 1 : Attaque par faute sur le DES

Une attaque par faute consiste à changer le résultat d'un sous calcul afin d'obtenir une information secrète. Ce changement va donc produire volontairement une erreur. Cette attaque est physique car, pour modifier la valeur de certains bits, il est nécessaire d'agir physiquement sur les composants électroniques. Dans le cas du DES avec une attaque par faute sur la valeur de sortie R_{15} du 15^e tour, cela signifie que la valeur R_{15} va être changer par l'attaquant.



A partir de cette attaque, il est possible de retrouver la clé secrète utilisée par la victime à partir de la sous clé K_{15}. On suppose ici que nous sommes l'attaquant et que nous avons réussi à obtenir de la victime le message clair associé à son message chiffré (avec une clé inconnue pour le moment, qui est à trouver). De plus, nous avons eu de la victime 32 chiffrés (toujours avec la même clé) et dont on a réussi à faire une attaque par faute. Ainsi, pour mener correctement à bien cette attaque, il faut trouver K_{16} puis en déduire K. Le détail de cette attaque sera montré par la suite.

2 Partie 2 : Application concrète

2.1 Question 1

Cette attaque par faute sur le dernier tour du DES comporte plusieurs étapes :

— étape 1 : trouver R_{15} à partir du chiffré juste et les $R_{15}*$ à partir des chiffrés faux.
Pour cela, on fait une permutation initiale (qui annule la permutation finale IP^{-1}) pour trouver L_{16} et R_{16}. On fera de même pour $R_{15}*$ à partir des chiffrés faux. On peut désormais écrire les formules suivantes :

$R_{16} = L_{15} \oplus f(K_{16}, R_{15})$ et $L_{16} = R_{15}$ pour le chiffré juste.
$R_{16}* = L_{15} \oplus f(K_{16}, R_{15}*)$ et $L_{16}* = R_{15}*$ pour les chiffrés faux.

Le but ici est d'obtenir K_{16} : pour cela, on fait le XOR entre R_{16} et un $R_{16}*$. Ce qui nous donne l'équation suivante :

$R_{16} \oplus R_{16}* = L_{15} \oplus f(K_{16}, R_{15}) \oplus L_{15} \oplus f(K_{16}, R_{15}*)$
Les L_{15} s'annulent et on obtient : $R_{16} \oplus R_{16}* = f(K_{16}, R_{15}) \oplus f(K_{16}, R_{15}*)$

Or, $f(K_{i+1}, R_i) = P(S(E(R_i) \oplus K_{i+1})) = P(S_1(E(R_i) \oplus K_{i+1})_{b_1 \to b_6} || ... || S_8(E(R_i) \oplus K_{i+1})_{b_{43} \to b_{48}})$

— étape 2 : pour chaque chiffré faux (associé à son $R_{15}*$), établir 8 équations pour chaque boîte-S.

$P^{-1}(R_{16} \oplus R_{16}*)_{b_1 \to b_4} = S_1(E(R_{15}) \oplus K_{16})_{b_1 \to b_4} \oplus S_1(E(R_{15}*) \oplus K_{16})_{b_1 \to b_4}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_5 \to b_8} = S_2(E(R_{15}) \oplus K_{16})_{b_5 \to b_8} \oplus S_2(E(R_{15}*) \oplus K_{16})_{b_5 \to b_8}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_9 \to b_{12}} = S_3(E(R_{15}) \oplus K_{16})_{b_9 \to b_{12}} \oplus S_3(E(R_{15}*) \oplus K_{16})_{b_9 \to b_{12}}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_{13} \to b_{16}} = S_4(E(R_{15}) \oplus K_{16})_{b_{13} \to b_{16}} \oplus S_4(E(R_{15}*) \oplus K_{16})_{b_{13} \to b_{16}}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_{17} \to b_{20}} = S_5(E(R_{15}) \oplus K_{16})_{b_{17} \to b_{20}} \oplus S_5(E(R_{15}*) \oplus K_{16})_{b_{17} \to b_{20}}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_{21} \to b_{24}} = S_6(E(R_{15}) \oplus K_{16})_{b_{21} \to b_{24}} \oplus S_6(E(R_{15}*) \oplus K_{16})_{b_{21} \to b_{24}}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_{25} \to b_{28}} = S_7(E(R_{15}) \oplus K_{16})_{b_{25} \to b_{28}} \oplus S_7(E(R_{15}*) \oplus K_{16})_{b_{25} \to b_{28}}$
$P^{-1}(R_{16} \oplus R_{16}*)_{b_{29} \to b_{32}} = S_8(E(R_{15}) \oplus K_{16})_{b_{29} \to b_{32}} \oplus S_8(E(R_{15}*) \oplus K_{16})_{b_{29} \to b_{32}}$

— étape 3 : éliminer les équations dont $P^{-1}(R_{16} \oplus R_{16}*)_{b_x \to b_y}$ vaut 0, ainsi que celle dont $S_z(E(R_{15}) \oplus K_{16})_{b_x \to b_y} = S_z(E(R_{15}*) \oplus K_{16})_{b_x \to b_y}$ puisqu'elles n'apporteront aucune information sur la portion de sous clé K_{16}.

— étape 4 : faire une attaque exhaustive sur la sortie de chaque boîte-S de $S_z(E(R_{15}) \oplus K_{16})_{b_x \to b_y}$: pour chaque élément, on déduit les possibles valeurs d'entrée de la boîte-S $E(R_{15}) \oplus K_{16}$. (En effet, avec la configuration non linéaire des boîtes-S, on a 4 valeurs d'entrée possibles par valeur de sortie.) Ainsi, on en déduit une possible K_{16}.

— étape 5 : pour chaque K_{16} possible, calculer $S_z(E(R_{15}*) \oplus K_{16})_{b_x \to b_y}$ et regarder si $P^{-1}(R_{16} \oplus R_{16}*)_{b_x \to b_y} = S_z(E(R_{15}) \oplus K_{16})_{b_x \to b_y} \oplus S_z(E(R_{15}*) \oplus K_{16})_{b_x \to b_y}$. Si c'est le cas, alors K_{16} en question devient une portion de clé candidate pour le chiffré faux associé.

— étape 6 : Pour chaque boîte-S, il y a une liste de clés candidates pour chaque chiffré faux qui agit sur la portion de sous clé. Trouver pour chaque boîte-S l'intersertion des portions de clés candidates. Il y aura donc 1 portion de clé (sur 6 bits) par boîte-S.

— étape 7 : concaténer les 8×6 bits pour obtenir la sous clé K_{16}.

La complexité pour trouver K_{16} correspond à la recherche exhaustive de l'étape 4 :
$O(32 \times 8 \times 2^4 \times 4)$ car pour chaque chiffré faux (dans notre cas 32), on regarde pour chaque boîte-S (8 dans le DES) les 4 entrées possibles (1 sortie vaut 4 entrées différentes dans une boîte-S).
Donc on a une complexité de $O(2^5 \times 2^3 \times 2^4 \times 2^2) = O(2^{14})$ pour trouver K_{16}.

2.2 Question 2

Après avoir implémenter les fonctions "*attack_sbox*", "$find\_intersection$" et "$find\_K16$" du fichier *attack.c* dans la section 6.2, on obtient pour chaque boîte-S les résultats suivants :

1. Boîte-S 1 :
   — Chiffré faux 1 : 0 4 5 10 14 19 20 24 25 30 34 39 ⟶ 12 portions de sous clés possibles
   — Chiffré faux 28 : 0 1 4 5 *a* *b* 16 17 24 25 34 35 ⟶ 12 portions de sous clés possibles
   — Chiffré faux 29 : 0 2 5 7 ⟶ 4 portions de sous clés possibles
   — Chiffré faux 30 : 0 1 4 5 9 d 20 23 24 27 ⟶ 10 portions de sous clés possibles
   — Chiffré faux 31 : 0 8 24 2c ⟶ 4 portions de sous clés possibles
   — Chiffré faux 32 : 0 e 10 1e 20 30 ⟶ 6 portions de sous clés possibles
   Portion de clé trouvée : 0 ⟶ 000000 en binaire
2. Boîte-S 2 :
   — Chiffré faux 24 : 2 3 10 11 12 13 1e 1f 20 21 30 31 34 35 ⟶ 14 portions de sous clés possibles
   — Chiffré faux 25 : 21 23 2d 2f ⟶ 4 portions de sous clés possibles
   — Chiffré faux 26 : 9 d 21 25 ⟶ 4 portions de sous clés possibles
   — Chiffré faux 27 : 10 14 18 1c 21 29 ⟶ 6 portions de sous clés possibles
   — Chiffré faux 28 : 4 f 14 1f 20 21 30 31 ⟶ 8 portions de sous clés possibles
   — Chiffré faux 29 : 1 4 7 17 1a 1f 21 24 27 37 3a 3f ⟶ 12 portions de sous clés possibles
   Portion de clé trouvée : 21 ⟶ 100001 en binaire
3. Boîte-S 3 :
   — Chiffré faux 20 : 2 3 ⟶ 2 portions de sous clés possibles
   — Chiffré faux 21 : 0 2 1d 1f ⟶ 4 portions de sous clés possibles
   — Chiffré faux 22 : 2 6 13 17 20 24 28 2c ⟶ 8 portions de sous clés possibles
   — Chiffré faux 23 : 2 7 a f 21 29 ⟶ 6 portions de sous clés possibles
   — Chiffré faux 24 : 2 d 12 1d 2d 3d ⟶ 6 portions de sous clés possibles
   — Chiffré faux 25 : 2 22 ⟶ 2 portions de sous clés possibles
   Portion de clé trouvée : 2 ⟶ 000010 en binaire

4. Boîte-S 4 :
   - Chiffré faux 16 : c d 10 11 14 15 18 19 20 21 26 27 30 31 32 33 ⟶ 16 portions de sous clés possibles
   - Chiffré faux 17 : 2c 2d 2e 2f 30 31 32 33 ⟶ 8 portions de sous clés possibles
   - Chiffré faux 18 : a e 31 35 ⟶ 4 portions de sous clés possibles
   - Chiffré faux 19 : 0 4 7 8 c f 31 39 ⟶ 8 portions de sous clés possibles
   - Chiffré faux 20 : e f 1e 1f 20 21 30 31 ⟶ 8 portions de sous clés possibles
   - Chiffré faux 21 : 4 5 10 11 24 25 30 31 ⟶ 8 portions de sous clés possibles

   Portion de clé trouvée : 31 ⟶ 110001 en binaire
5. Boîte-S 5 :
   - Chiffré faux 12 : 4 5 8 9 a *b* 1a 1b 28 29 3c 3d –> 12 portions de sous clés possibles
   - Chiffré faux 13 : 8 9 a *b* d f –> 6 portions de sous clés possibles
   - Chiffré faux 14 : 0 4 9 *b* d f 33 37 38 3c –> 10 portions de sous clés possibles
   - Chiffré faux 15 : 3 6 *b* e 13 15 1b 1d 35 3d –> 10 portions de sous clés possibles
   - Chiffré faux 16 : 3 6 a *b* 13 16 1a 1b 27 2a 37 3a –> 12 portions de sous clés possibles
   - Chiffré faux 17 : 6 *b* d 1f 26 2b 2d 3f –> 8 portions de sous clés possibles

   Portion de clé trouvée : b ⟶ 001011 en binaire
6. Boîte-S 6 :
   - Chiffré faux 8 : 4 5 14 15 18 19 ⟶ 6 portions de sous clés possibles
   - Chiffré faux 9 : 4 6 14 16 38 3a ⟶ 6 portions de sous clés possibles
   - Chiffré faux 10 : 0 4 20 21 24 25 29 2d ⟶ 8 portions de sous clés possibles
   - Chiffré faux 11 : 4 c ⟶ 2 portions de sous clés possibles
   - Chiffré faux 12 : 3 4 5 6 13 14 15 16 ⟶ 8 portions de sous clés possibles
   - Chiffré faux 13 : 0 4 1b 20 24 3b ⟶ 6 portions de sous clés possibles

   Portion de clé trouvée : 4 ⟶ 000100 en binaire
7. Boîte-S 7 :
   - Chiffré faux 4 : 2 3 8 9 1c 1*d* ⟶ 6 portions de sous clés possibles
   - Chiffré faux 5 : 1*d* 1f 24 26 ⟶ 4 portions de sous clés possibles
   - Chiffré faux 6 : 19 1*d* ⟶ 2 portions de sous clés possibles
   - Chiffré faux 7 : 6 7 e f 15 1*d* 30 32 34 38 3a 3c ⟶ 12 portions de sous clés possibles
   - Chiffré faux 8 : d e 1*d* 1e 25 28 2a 35 38 3a ⟶ 10 portions de sous clés possibles
   - Chiffré faux 9 : 0 2 4 c 14 18 1*d* 20 22 24 2c 34 38 3d ⟶ 14 portions de sous clés possibles

   Portion de clé trouvée : 1d ⟶ 011101 en binaire
8. Boîte-S 8 :
   - Chiffré faux 1 : 5 7 9 b c e 24 26 39 3b ⟶ 10 portions de sous clés possibles
   - Chiffré faux 2 : 8 b c f 20 24 28 2c 31 35 39 3d ⟶ 12 portions de sous clés possibles
   - Chiffré faux 3 : 10 18 31 34 35 39 3c 3d ⟶ 8 portions de sous clés possibles
   - Chiffré faux 4 : 8 9 18 19 29 39 ⟶ 6 portions de sous clés possibles
   - Chiffré faux 5 : 2 6 9 d 12 15 19 1a 22 26 29 2d 32 35 39 3a ⟶ 16 portions de sous clés possibles
   - Chiffré faux 32 : 38 39 ⟶ 2 portions de sous clés possibles

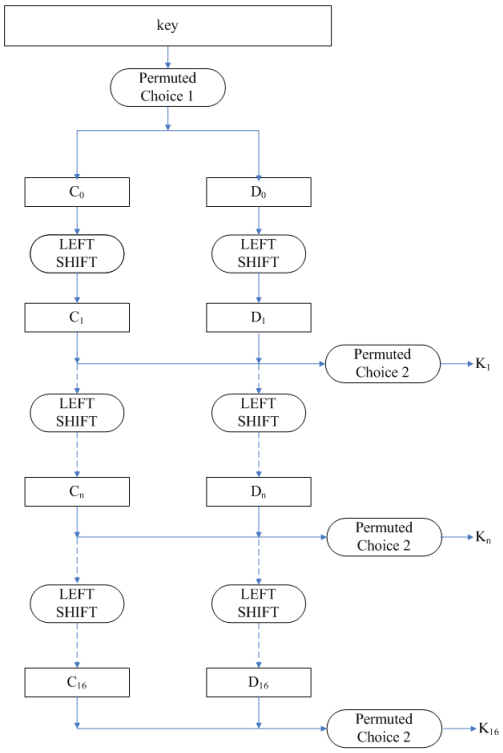
   Portion de clé trouvée : 39 ⟶ 111001 en binaire

On a donc en binaire : 000000 100001 000010 110001 001011 000100 011101 111001
soit 00000010 00010000 10110001 00101100 01000111 01111001 toujours en binaire.
On obtient en hexadécimal pour K_{16} : $K_{16} = 02\ 10\ B1\ 2C\ 47\ 79$

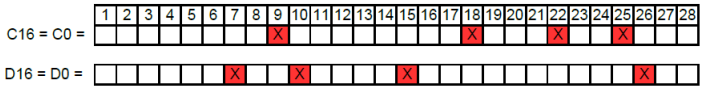
3 Partie 3 : Retrouver la clé complète du DES

3.1 Question 1

Dans la question précédente, nous avons réussi à obtenir la clé secrète K_{16} qui contient 48 bits. En analysant le schéma de création des 16 sous-clés (de 48 bits chacunes) à partir de la clé secrète (de 64 bits avec les 8 bits de parité), on peut en déduire la clé secrète. Cela comporte plusieurs étapes :



— étape 1 : effectuer une permutation inverse $PC_2^{-1}(K_{16})$ afin d'obtenir C_{16} et D_{16}. Il est important de noter que lors de cette permutation inverse, 8 bits sont inconnus. En effet, on passe de 48 bits à $2 \times 28 = 56$ bits. Il y a donc 8 bits qu'on ne peut déduire à partir de K_{16}.

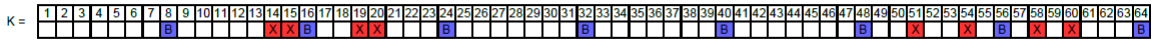


Sur ce schéma, les bits encore inconnus sont marqués par "X".

— étape 2 : déduire C_0 et D_0 à partir de C_{16} et D_{16}. En effet, $C_0 = C_{16}$ et $D_0 = D_{16}$ puisque la somme des shifts circulaires donne 28 qui correspond à la taille des blocs C_i et D_i. Les 8 bits inconnus n'ont donc pas bougé de place dans C_0 et D_0.

— étape 3 : effectuer une permutation inverse $PC_1^{-1}(C_0||D_0)$ afin d'obtenir la clé finale sur 64 bits. On peut noter ici que les 8 bits inconnus sont mélangés dans la clé finale. De plus, 8

bits supplémentaires ont été rajoutés correspondant aux bits de parité. On a donc une clé finale sur 64 bits dont 16 bits sont inconnus.



Sur le schéma ci-dessous, "B" correspond aux bits de parité.

— étape 4 : retrouver les 8 bits inconnus par une recherche exhaustive : $2^8 = 256$ cas possibles. Pour chaque clé possible, on fait le chiffrement DES avec en entrée la clé possible et le message clair fourni. Si la clé testée est celle recherchée, alors le chiffré obtenu sera identique à celui du chiffré correct fourni.

— étape 5 : déduire les 8 bits de parité en s'assurant que chaque octet possède un nombre impair de bits à 1.

La complexité pour trouver K à partir de K_{16} est de $O(2^8)$ car à partir de K_{16}, 8 bits sont encore inconnus pour K sur 56 bits, d'où une recherche exhaustive de $O(2^8)$. A partir de cela, les 8 bits de parité (inconnus sur K de 64 bits) sont à déduire en temps constant.

On aura donc une complexité totale pour trouver K_{16} puis K de $O(2^{14} + 2^8) \simeq O(2^{14})$

3.2 Question 2

Après avoir implémenter les fonctions "$build\_C16\_D16$", "$build\_K56$, "$build\_K$", "set_parity_bits" et "$find\_K$" du fichier *attack.c* dans la section 6.2, on va effectuer les différentes étapes en détaillant :

— On a obtenu précédemment lors de la recherche de la sous clé : $K_{16} = 02\ 10\ B1\ 2C\ 47\ 79$.

Avec l'étape 1 et 2, on obtient :
- $C_0 = C_{16} = 0110\ 00100100\ 00010000\ 00000100 = 6241004$
- $D_0 = D_{16} = 0011\ 00000010\ 01001110\ 11001011 = 3024ECB$
- $C_0||D_0 = 62\ 41\ 00\ 43\ 02\ 4E\ CB$

Les 8 bits inconnus à chercher sont à 0.

— Avec l'étape 3, on obtient la clé DES sur 64 bits avec les 8 bits inconnus à 0 et les bits de parité également à 0 : $K_{64*avant\_recherche\_exhaustive} = 50\ 90\ 0C\ 18\ 02\ 8E\ D8\ 08$

— On réalise l'attaque exhaustive et on trouve la solution en ajustant également les bits de parité et on obtient la solution finale de la clé DES sur 64 bits en hexadécimal :
$K_{64} = 51\ 92\ 2C\ 19\ 02\ 8F\ FD\ 49$

4 Partie 4 : Fautes sur les tours précédents

On rappelle la loi de Moore qui dit que la puissance de calcul double tous les 18 mois. Pour rendre une attaque infaisable dans le monde civil, il faudrait 2^{80} opérations élémentaires et 2^{128} dans le monde militaire.
De plus, nous avons trouvé une complexité de $0(2^{14})$ pour une attaque DFA sur la valeur de sortie R_{15}.

4.1 Faute provoquée sur la valeur de sortie R_{14} du 14^e tour

Pour ce type d'attaque, on obtient 2 paires d'égalités :

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

On obtient donc les équations suivantes :
$R_{15} \oplus R_{15}* = L_{14} \oplus f(K_{15}, R_{14}) \oplus L_{14} \oplus f(K_{15}, R_{14}*) = f(K_{15}, R_{14}) \oplus f(K_{15}, R_{14}*)$

On décompose pour obtenir (E) (en rouge les valeurs inconnues et vert les valeurs connues) :
$P^{-1}(R_{15} \oplus R_{15}*) = S_i(E(R_{14}) \oplus K_{15}) \oplus S_i(E(R_{14}*) \oplus K_{15})$

Or, $R_{14} = L_{15} = R_{16} \oplus f(K_{16}, L_{16})$ donne $P^{-1}(L_{15} \oplus R_{16})_{b_x \to b_y} = S_i(E(L_{16}) \oplus K_{16})_{b_x \to b_y}$ et
$R_{14}* = L_{15}* = R_{16} * \oplus f(K_{16}, L_{16}*)$ donne $P^{-1}(L_{15} * \oplus R_{16}*)_{b_x \to b_y} = S_i(E(L_{16}*) \oplus K_{16})_{b_x \to b_y}$.
On fait donc une attaque exhaustive et on déduit les valeurs possibles pour L_{15} et les $L_{15}*$. Cette attaque a donc une complexité de $O(32 \times 8 \times 2^4 \times 4) = O(2^5 \times 2^3 \times 2^4 \times 2^2) = O(2^{14})$.

Ensuite, pour chaque L_{15} et $L_{15}*$ possible, on fait une attaque de complexité de $O(2^{14})$ pour trouver K_{15} avec l'équation (E).
On a donc une complexité de $O(2^{14} \times 2^{14}) = O(2^{28})$ sur l'attaque par faute provoquée sur la valeur de sortie R_{14} du 14^e tour. Cette attaque reste réaliste car sa complexité est inférieure à $O(2^{80})$.

4.2 Faute provoquée sur la valeur de sortie R_i du i^e tour

Notons $O(2^a)$ la complexité de l'attaque DFA sur DES sur la valeur de sortie de R_{15}. Avec le paragraphe précédent, nous avons élaboré une attaque qui fonctionne sur n'importe quelle valeur de sortie de R_i en multipliant la complexité de l'attaque du tour précédent par $O(2^a)$. On obtient donc les résultats suivants :

— Pour l'attaque sur le 15^e tour, la complexité est de 2^{14}, qui est une attaque réaliste.
— Pour l'attaque sur le 14^e tour, la complexité est de 2^{28}, qui est une attaque réaliste.
— Pour l'attaque sur le 13^e tour, la complexité est de 2^{42}, qui est une attaque réaliste.
— Pour l'attaque sur le 12^e tour, la complexité est de 2^{56}, qui est une attaque réaliste.
— Pour l'attaque sur le 11^e tour, la complexité est de 2^{70}, qui est encore une attaque réaliste.
— Pour l'attaque sur le 10^e tour, la complexité est de 2^{84}, qui est une attaque non réaliste de nos jours puisqu'elle est supérieure à $O(2^{80})$.

5 Partie 5 : Contre-mesures

Il existe plusieurs contre-mesures possibles contre ce type d'attaque par fautes sur le DES :

— installer une sorte de "bouclier" contre les perturbations extérieures. En effet, cette attaque par faute peut être réalisée en agissant physiquement sur les composants électroniques, tels que du laser, une hausse de température ou une modification du champ magnétique environnant. Ainsi, un bouclier physique permettrait de contrer ce type d'attaque. Il n'y aura aucun impact sur le temps de calcul par rapport à une implémentation non sécurisée puisqu'on ne change pas le logiciel.

— réaliser deux fois le calcul afin de vérifier que l'on obtient deux fois le même calcul. Cela permettrait d'annuler le calcul si une attaque de ce genre était effectuée. Il y aura un impact sur le temps de calcul : vu qu'on réalisera deux fois le même calcul, alors le temps de calcul va doublé par rapport à une implémentation non sécurisée.

6 Annexe

6.1 main.c

```c
/**
 * \file main.c
 * \brief Represente les fonctions concernant l'attaque DFA sur DES.
 * \author Clement CAUMES
 * */

#include <stdio.h>
#include <stdlib.h>

#include "../inc/attack.h"
#include "../inc/constants.h"
#include "../inc/errors.h"

/**
 * \fn int main
 * \brief Fonction principale du programme qui realise l'attaque.
 * \return renvoie EXIT_FAILURE en cas d'erreur et EXIT_SUCCESS sinon.
 */
int main(){

    // Initialisation des donnees avec un clair, un chiffre vrai et 32 chiffres fautes.
    DATA data = initialize_data();

    // Recherche de K16 (48 bits).
    if (find_K16(&data))
        return err_print(des_errno), EXIT_FAILURE;

    // Recherche de K (64 bits).
    if (find_K(&data))
        return err_print(des_errno), EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

Listing 1 – main.c

6.2 attack.c

```
1  /**
2   * \file attack.c
3   * \brief Représente les fonctions concernant l'attaque DFA sur DES.
4   * \author Clément CAUMES
5   * */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #include "../inc/key_schedule.h"
11 #include "../inc/inner_function.h"
12 #include "../inc/manip_bits.h"
13 #include "../inc/feistel.h"
14 #include "../inc/attack.h"
15 #include "../inc/constants.h"
16 #include "../inc/errors.h"
17
18 /**
19  * \fn DATA initialize_data()
20  * \brief Fonction d'initialisation des données pour réaliser l'attaque.
21  * \return renvoie la structure DATA initialisée avec les données obtenues.
22  */
23 DATA initialize_data(){
24     DATA d;
25     uint64_t chiffre_juste = 0x670994D1365D5EAD; //mon chiffre juste
26     d.chiffre_juste.output = chiffre_juste; //mon chiffre juste
27     d.chiffre_juste.R15 = get_R15(chiffre_juste);
28     d.chiffre_juste.R16 = get_R16(chiffre_juste);
29     d.message_clair=0xFBA2DC5EEAA7FEC2;
30
31     uint64_t chiffre_faux[32] = {
32         0x650C94D5365C5EB9, 0x671B94D5365C5EAD, 0x67099695365D5EAD, 0
       x66599097365C5EAD,
33         0x665994D5345D5EAD, 0x664990D1265F5EAD, 0x660990D1365D5CAD, 0
       x664990D0364D5EAF,
34         0x6F4990D076595EAD, 0x670194D0364D5EAD, 0x67099CD036595EAD, 0
       x270984D8765D5EAD,
35         0x270994D03E195EAD, 0x670984D136555EAC, 0x270994D1365D56AC, 0
       x270984D1325D5EE4,
36         0x470994D1361D4EEC, 0x672994D1325D4EED, 0x6709B4D1325D4FAD, 0
       x6709D5F1325D4FED,
37         0x730994D1165D4EED, 0x7309D4D1367D5EAD, 0x630995D1365D7EAD, 0
       x6309D5C1365D5A8D,
38         0xE30995C1365D1EAD, 0x678994C1365D1AAD, 0x670914C1375D5EAD, 0
       x670D9451365D1ABD,
39         0x670D94D1B75D5AB9, 0x670D94D136DD5EA9, 0x670C94D1365DDEB9, 0
       x671D9491365C5E3D,
40     };
41
42     int i;
43     for(i=0;i<32;i++){
44         d.chiffre_faux[i].output = chiffre_faux[i];
45         d.chiffre_faux[i].R15 = get_R15(chiffre_faux[i]);
```

```
46          d.chiffre_faux[i].R16 = get_R16(chiffre_faux[i]);
47      }
48      return d;
49  }
50
51  /**
52   * \fn uint32_t get_R15(uint64_t cipher)
53   * \brief Fonction qui calcule la valeur de R15 pour un chiffré donné.
54   * \param cipher chiffré dont on veut calculer R15.
55   * \return renvoie R15 de cipher.
56   */
57  uint32_t get_R15(uint64_t cipher){
58      // permutation initiale
59      if(process_permutation(&cipher, IP))
60          return des_errno=ERR_ATTACK, 1;
61
62      // division en L0 et R0
63      uint32_t L16, R16;
64      build_L0_R0(cipher, &L16, &R16);
65
66      //inversion des L16 et R16 donc R16<->L16
67      return R16;
68  }
69
70  /**
71   * \fn uint32_t get_R16(uint64_t cipher)
72   * \brief Fonction qui calcule la valeur de R16 pour un chiffré donné.
73   * \param cipher chiffré dont on veut calculer R16.
74   * \return renvoie R16 de cipher.
75   */
76  uint32_t get_R16(uint64_t cipher){
77      // permutation initiale
78      if(process_permutation(&cipher, IP))
79          return des_errno=ERR_ATTACK, 1;
80
81      // division en L0 et R0
82      uint32_t L16, R16;
83      build_L0_R0(cipher, &L16, &R16);
84
85      //inversion des L16 et R16 donc R16<->L16
86      return L16;
87  }
88
89  /**
90   * \fn uint8_t find_intersection(int selection_key[64])
91   * \brief Fonction qui renvoie l'intersection des clés candidates
92   * lors de l'attaque sur les S Box.
93   * \param selection_key tableau représentant toutes les clés candidates.
94   * \return renvoie l'intersection des clés candidates.
95   */
96  uint8_t find_intersection(int selection_key[64]){
97      uint8_t rang_max=0;
98      uint8_t max=selection_key[0];
99      int i;
100     for(i=0;i<64;i++){
101         if (max<selection_key[i]){
```

```
102             max=selection_key[i];
103             rang_max=i;
104         }
105     }
106     return rang_max;
107 }
108
109 /**
110  * \fn int attack_sbox(DATA* data, uint8_t* uint48_t_part, int num_sbox)
111  * \brief Fonction qui fait l'attaque exhaustive pour trouver la portion de sous clé
112  * associé à la sbox.
113  * \param *data données qui contiennent les informations nécessaires à l'attaque.
114  * \param *uint48_t_part représente la portion de sous clé associé à la sbox et qui
     sera
115  * donc initialisée.
116  * \param num_sbox représente le numéro de la sbox demandé.
117  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
118  */
119 int attack_sbox(DATA* data, uint8_t* uint48_t_part, int num_sbox){
120     // Renvoie une erreur si le numéro de la sbox est incorrect.
121     if ((num_sbox<=0)||(num_sbox>8)) return 1;
122
123     // Initialiation de toutes les variables
124     int w, s, q, v, u, l, h;
125     int candidate_key[32][65];
126     int* selection_key=malloc(64*sizeof(int));
127     for(s=0;s<32;s++){
128         for(w=0;w<64;w++){
129             candidate_key[s][w]=0;
130             selection_key[w]=0;
131         }
132         candidate_key[s][64]=1;
133     }
134
135     int compte=0; int temp=0; int to_do=1; int i=0; int compteur=0;
136     uint32_t R15_chiffre_faux, perm;
137     uint8_t val_sbox1, input1, input2, input3, input4, input5, input6, input7,
    input8;
138     uint8_t k16_val_sbox1[4], k16_val_sbox2[4], val_sbox2[4], bit;
139     uint8_t k16_1, k16_2, k16_3, k16_4, k16_5, k16_6, k16_7, k16_8,
    expand_R15_faux_xor_k16;
140     uint8_t bits6_expand_R15=0; uint8_t bits6_expand_R15_faux; uint8_t boite=0x00;
141     uint64_t expand_R15, expand_R15_faux;
142
143     // Calcul de E(R15)
144     expand(&expand_R15, data->chiffre_juste.R15);
145
146     // Calcul de E(R15)x->y avec [x,y] l'ensemble des bits correspondant à la boite
    s cherchée.
147     l=6;
148     for(u=(48-(num_sbox-1)*6);u>=(43-(num_sbox-1)*6);u--){
149         bit=get_bit_uint64_t(expand_R15, u);
150         set_bit_uint8_t(&bits6_expand_R15, bit, l);
151         l--;
152     }
153
```

```
154     // Pour chaque chiffré faux.
155     for(i=0;i<32;i++){
156 
157         to_do=1;
158         // Calcul de E(R15*) avec R15* le R15 du chiffré faux.
159         expand(&expand_R15_faux, data->chiffre_faux[i].R15);
160 
161         // Calcul de E(R15*)x->y avec [x,y] l'ensemble des bits correspondant à la
    boite s cherchée.
162         l=6;
163         for(u=(48-(num_sbox-1)*6);u>=(43-(num_sbox-1)*6);u--){
164             bit=get_bit_uint64_t(expand_R15_faux, u);
165             set_bit_uint8_t(&bits6_expand_R15_faux, bit, l);
166             l--;
167         }
168 
169         // Calcul de P-1(R16 XOR R16*).
170         permutation_inv_inner_function(&perm, ((data->chiffre_faux[i].R16) ^ (data->
    chiffre_juste.R16)));
171 
172         // Calcul de P-1(R16 XOR R16*)x->y avec [x,y] l'ensemble des bits
    correspondant à la boite s cherchée.
173         boite=0x00; l=4;
174         for(u=(32-(num_sbox-1)*4);u>=(29-(num_sbox-1)*4);u--){
175             bit=get_bit_uint32_t(perm, u);
176             set_bit_uint8_t(&boite, bit, l);
177             l--;
178         }
179 
180         /*
181          * Si P-1(R16 XOR R16*)x->y=0 alors ce chiffré n'apporte pas d'informations
    sur la
182          * portion de sous clé associée à la boite S recherchée.
183          */
184         if (boite==0) {
185             to_do=0;
186             candidate_key[i][64]=0;
187         }
188 
189         /*
190          * Si E(R15)x->y=E(R15*)x->y alors ce chiffré n'apporte pas non plus d'
    informations sur la
191          * portion de sous clé associée à la boite S recherchée.
192          */
193         if (bits6_expand_R15_faux==bits6_expand_R15){
194             to_do=0;
195             candidate_key[i][64]=0;
196         }
197 
198         // Si ce chiffré approte des informations sur la portion de sous clé.
199         if (to_do==1){
200             // On teste pour Sz(E(R15) XOR K16)x->y possible (avec Sz la boite S num
    éro z).
201             for(val_sbox1=0; val_sbox1<=15 ; val_sbox1++){
202                 /*
```

```
203                  * On fait la procédure inverse de la boite S (4 inputs donne le même output)
204                  * On en déduit donc 4 possibilités pour E(R15) XOR K16.
205                  */
206                 if (num_sbox==1) get_input_sbox(val_sbox1, S1, &input1, &input2, &input3, &input4);
207                 else if (num_sbox==2) get_input_sbox(val_sbox1, S2, &input1, &input2, &input3, &input4);
208                 else if (num_sbox==3) get_input_sbox(val_sbox1, S3, &input1, &input2, &input3, &input4);
209                 else if (num_sbox==4) get_input_sbox(val_sbox1, S4, &input1, &input2, &input3, &input4);
210                 else if (num_sbox==5) get_input_sbox(val_sbox1, S5, &input1, &input2, &input3, &input4);
211                 else if (num_sbox==6) get_input_sbox(val_sbox1, S6, &input1, &input2, &input3, &input4);
212                 else if (num_sbox==7) get_input_sbox(val_sbox1, S7, &input1, &input2, &input3, &input4);
213                 else get_input_sbox(val_sbox1, S8, &input1, &input2, &input3, &input4);
214 
215                 // On en déduit donc 4 portions de clés possibles par Sz(E(R15) XOR K16).
216                 k16_val_sbox1[0]=input1^bits6_expand_R15;
217                 k16_val_sbox1[1]=input2^bits6_expand_R15;
218                 k16_val_sbox1[2]=input3^bits6_expand_R15;
219                 k16_val_sbox1[3]=input4^bits6_expand_R15;
220 
221                 // Pour chaque portions de clés possibles.
222                 for(v=0;v<4;v++){
223                     // On calcule (E(R15*) XOR K16) avec K16 la clé possible en question.
224                     expand_R15_faux_xor_k16=bits6_expand_R15_faux^k16_val_sbox1[v];
225 
226                     // Et on en déduit Sz(E(R15*) XOR K16).
227                     if (num_sbox==1) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S1);
228                     else if (num_sbox==2) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S2);
229                     else if (num_sbox==3) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S3);
230                     else if (num_sbox==4) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S4);
231                     else if (num_sbox==5) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S5);
232                     else if (num_sbox==6) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S6);
233                     else if (num_sbox==7) val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S7);
234                     else val_sbox2[v]=process_S_box_particular(expand_R15_faux_xor_k16, S8);
235 
236                     // Si P-1(R16 XOR R16*)x->y = Sz(E(R15) XOR K16) Sz(E(R15*) XOR K16) avec K16 la clé possible.
237                     if (boite==(val_sbox2[v]^val_sbox1)){
238                         // K16 devient une clé CANDIDATE
```

```
239                         candidate_key[i][k16_val_sbox1[v]]++;
240                         selection_key[k16_val_sbox1[v]]++;
241                     }
242                 }
243             }
244         }
245     }
246 
247     // Affichage des clés candidates.
248     printf("Recherche SBOX : ");
249     for(s=0;s<32;s++){
250         h=0;
251         if (candidate_key[s][64]==1){
252             for(w=0;w<64;w++){
253                 if (candidate_key[s][w]!=0) {
254                     printf_uint8_t_hexa(w);
255                     printf(" ");
256                     h++;
257                 }
258             }
259             printf("--> %d portions de sous clés possibles\n", h);
260         }
261 
262     }
263 
264     // Recherche de l'intersection entre tous les chiffrés qui ont un impact sur la
    portion de sous clé.
265     *uint48_t_part=find_intersection(selection_key);
266     printf("Portion de clé trouvée : %x\n\n", *uint48_t_part);
267 
268     free(selection_key);
269     return 0;
270 }
271 
272 /**
273  * \fn int find_K16(DATA* data)
274  * \brief Fonction qui trouve K16 à partir des données contenues dans *data.
275  * \param *data données qui contiennent les informations nécessaires à l'attaque.
276  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
277  */
278 int find_K16(DATA* data){
279 
280     printf("\nRECHERCHE DE K16 :\n");
281 
282     int i; uint8_t part_k16; uint64_t k16_temp;
283     data->k16.bytes=0x00;
284     // On cherche les 8 portions de sous clé de K16 (1 par Sbox).
285     for(i=1; i<=8;i++){
286         part_k16=0x00; k16_temp=0x00;
287         if (attack_sbox(data, &part_k16, i))
288             return des_errno=ERR_ATTACK, 1;
289         k16_temp=part_k16;
290         k16_temp<<=((8-i)*6);
291         data->k16.bytes|=k16_temp;
292     }
293 
```

```
294     printf("\nK16 trouvée : %LX\n", data->k16.bytes);
295     return 0;
296 }
297 
298 /**
299  * \fn int build_C16_D16(uint48_t k16, uint32_t* C16, uint32_t* D16)
300  * \brief Fonction qui déduit C16 et D16 à partir de k16.
301  * \param k16 sous clé trouvée précedemment.
302  * \param *C16 partie de l'algorithme de key schedule à initialiser.
303  * \param *D16 partie de l'algorithme de key schedule à initialiser.
304  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
305  */
306 int build_C16_D16(uint48_t k16, uint32_t* C16, uint32_t* D16){
307     *C16=0; *D16=0;
308     int i; uint8_t rang; uint8_t bit;
309     for(i=0;i<48;i++){
310         rang=PC2[i];
311         bit=get_bit_uint64_t(k16.bytes, 48-i);
312         if (rang<=28){ //C16
313             if (set_bit_uint32_t(C16, bit, 29-rang))
314                 return 1;
315         }
316         else { //D16
317             if (set_bit_uint32_t(D16, bit, (29-(rang-28))))
318                 return 1;
319         }
320     }
321     return 0;
322 }
323 
324 /**
325  * \fn uint64_t build_K56(uint32_t C0, uint32_t D0)
326  * \brief Fonction qui construit K (sur 56 bits) à partir de C0 et D0.
327  * \param C0 partie de l'algorithme de key schedule avec C0=C16.
328  * \param D0 partie de l'algorithme de key schedule avec D0=D16.
329  * \return renvoie K sur 56 bits (avec 8 bits non encore trouvés).
330  */
331 uint64_t build_K56(uint32_t C0, uint32_t D0){
332     uint64_t K56=0x00;
333     K56=C0;
334     K56<<=28;
335     K56|=D0;
336     return K56;
337 }
338 
339 /**
340  * \fn int build_K(uint64_t* K, uint32_t C16, uint32_t D16)
341  * \brief Fonction qui construit K (sur 64 bits) à partir de C16 et D16.
342  * \param *K clé à trouver sur 64 bits.
343  * \param C16 partie de l'algorithme de key schedule.
344  * \param D16 partie de l'algorithme de key schedule.
345  * \return renvoie K sur 64 bits (avec 16 bits non encore trouvés dont 8 bits de
     parité).
346  */
347 int build_K(uint64_t* K, uint32_t C16, uint32_t D16){
348     int i;
```

```
349     uint8_t rang; uint8_t bit;
350 
351     uint64_t k56 = build_K56(C16, D16);
352     *K=0x00;
353 
354     for(i=0;i<56;i++){
355         rang=PC1[i];
356         bit=get_bit_uint64_t_most(k56, i+1+8);
357         if (set_bit_uint64_t(K, bit, 65-rang))
358             return 1;
359     }
360     return 0;
361 }
362 
363 /**
364  * \fn int set_parity_bits(uint64_t* K)
365  * \brief Fonction qui modifie la clé pour mettre correctement les bits de parité.
366  * \param *K clé à trouver sur 64 bits.
367  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
368  */
369 int set_parity_bits(uint64_t* K){
370     uint8_t bit1, bit2, bit3, bit4, bit5, bit6, bit7, bit8;
371     int i; int j=1; int compteur;
372     for(i=0;i<8;i++){
373         compteur=0;
374         bit1 = get_bit_uint64_t_most(*K, j);
375         if (bit1==1) compteur++;
376         bit2 = get_bit_uint64_t_most(*K, j+1);
377         if (bit2==1) compteur++;
378         bit3 = get_bit_uint64_t_most(*K, j+2);
379         if (bit3==1) compteur++;
380         bit4 = get_bit_uint64_t_most(*K, j+3);
381         if (bit4==1) compteur++;
382         bit5 = get_bit_uint64_t_most(*K, j+4);
383         if (bit5==1) compteur++;
384         bit6 = get_bit_uint64_t_most(*K, j+5);
385         if (bit6==1) compteur++;
386         bit7 = get_bit_uint64_t_most(*K, j+6);
387         if (bit7==1) compteur++;
388 
389         if ((compteur%2)==0) bit8=1;
390         else bit8=0;
391 
392         if (set_bit_uint64_t(K, bit8, ((8*(8-i))-7)))
393             return 1;
394         j+=8;
395     }
396     return 0;
397 }
398 
399 /**
400  * \fn int find_K(DATA* data)
401  * \brief Fonction qui trouve K sur 64 bits.
402  * \param *data données qui contiennent les informations nécessaires à l'attaque.
403  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
404  */
```

```
405 int find_K(DATA* data){
406 
407     // Deduit C16 et D16 à partir de K16
408     uint32_t C16, D16;
409     if (build_C16_D16(data->k16, &C16, &D16))
410         return 1;
411 
412     // Construit K (sur 64 bits) avec 8 bits à trouver et 8 bits de parité non
        encore déduits.
413     if (build_K(&(data->key), C16, D16))
414         return 1;
415 
416     // Faire une recherche exhaustive afin de trouver les 8 bits manquants
417     int j;
418     uint8_t i, bit1, bit2, bit3, bit4, bit5, bit6, bit7, bit8;
419     uint64_t message_clair=data->message_clair;
420     for (i=0;i<0xFF; i++){
421 
422         // recherche exhaustive sur les 8 bits non connus
423         bit1=get_bit_uint8_t(i,8);
424         if (set_bit_uint64_t(&(data->key), bit1, 7))
425             return 1;
426         bit2=get_bit_uint8_t(i,7);
427         if (set_bit_uint64_t(&(data->key), bit2, 14))
428             return 1;
429         bit3=get_bit_uint8_t(i,6);
430         if (set_bit_uint64_t(&(data->key), bit3, 46))
431             return 1;
432         bit4=get_bit_uint8_t(i,5);
433         if (set_bit_uint64_t(&(data->key), bit4, 5))
434             return 1;
435         bit5=get_bit_uint8_t(i,4);
436         if (set_bit_uint64_t(&(data->key), bit5, 50))
437             return 1;
438         bit6=get_bit_uint8_t(i,3);
439         if (set_bit_uint64_t(&(data->key), bit6, 11))
440             return 1;
441         bit7=get_bit_uint8_t(i,2);
442         if (set_bit_uint64_t(&(data->key), bit7, 51))
443             return 1;
444         bit8=get_bit_uint8_t(i,1);
445         if (set_bit_uint64_t(&(data->key), bit8, 45))
446             return 1;
447 
448         //ajout des bits de parité
449         if (set_parity_bits(&(data->key)))
450             return 1;
451 
452         data->message_clair=message_clair;
453         encryption_des(&(data->message_clair), (data->key));
454 
455         // Si le chiffrement du message clair avec la clé donne le chiffré correct
        alors la clé sur 64 bits a été trouvée.
456         if ((data->message_clair)==data->chiffre_juste.output){
457             printf("K trouvée : ");
458             printf_uint64_t_hexa(data->key);
```

```
459
460                 return 0;
461             }
462         }
463         return 1;
464     }
```

Listing 2 – attack.c

6.3 key_schedule.c

```
1  /**
2   * \file key_schedule.c
3   * \brief Représente les fonctions concernant la génération des sous
4   * clés.
5   * \author Clément CAUMES
6   * */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <limits.h>
11 #include <string.h>
12
13 #include "../inc/key_schedule.h"
14 #include "../inc/errors.h"
15 #include "../inc/manip_bits.h"
16 #include "../inc/constants.h"
17
18 /*
19  * Constante PC1.
20  */
21 int PC1[] = { 57, 49,  41, 33,  25,  17,  9,
22               1, 58,  50, 42,  34,  26, 18,
23               10,  2,  59, 51,  43,  35, 27,
24               19, 11,   3, 60,  52,  44, 36,
25               63, 55,  47, 39,  31,  23, 15,
26               7, 62,  54, 46,  38,  30, 22,
27               14,  6,  61, 53,  45,  37, 29,
28               21, 13, 5, 28, 20, 12, 4};
29
30 /*
31  * Constante PC2.
32  */
33 int PC2[] = { 14, 17, 11, 24, 1, 5,
34               3, 28, 15, 6, 21, 10,
35               23, 19, 12, 4, 26, 8,
36               16, 7, 27, 20, 13, 2,
37               41, 52, 31, 37, 47, 55,
38               30, 40, 51, 45, 33, 48,
39               44, 49, 39, 56, 34, 53,
40               46, 42, 50, 36, 29, 32};
41
42 /**
43  * \fn int init_C0_D0(KEY* k, uint64_t init)
44  * \brief Fonction qui initialise Ci et Di de la structure KEY.
45  * \param *key clef qui possède les champs Ci et Di qui seront initialisés.
46  * \param init état de la clef sur 56 bits après la permutation initiale.
47  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
48  */
49 int init_C0_D0(KEY* key, uint64_t init){
50     key->Ci=key->Di=0;
51     int i;
52     uint8_t bit;
53     for(i=1;i<=28;i++){
```

```
54          bit=get_bit_uint64_t(init, i);
55          if(set_bit_uint32_t(&(key->Di), bit, i))
56                  return 1;
57      }
58
59      for(i=29;i<=56;i++){
60          bit=get_bit_uint64_t(init, i);
61          if(set_bit_uint32_t(&(key->Ci), bit, i-28))
62                  return 1;
63      }
64      return 0;
65  }
66
67  /**
68   * \fn int shift_Ci_Di(uint32_t* val, int times)
69   * \brief Fonction qui shift Ci et Di un certain nombre de fois times.
70   * \param *val Ci ou Di à shifter.
71   * \param times nombre de shifts à faire.
72   * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
73   */
74  int shift_Ci_Di(uint32_t* val, int times){
75      int i;
76      for(i=0;i<times;i++){
77          uint8_t bit=get_bit_uint64_t(*val, 28);
78          (*val)<<=1;
79          if (set_bit_uint32_t(val, 0, 29))
80              return 1;
81          if (set_bit_uint32_t(val, bit, 1))
82              return 1;
83      }
84      return 0;
85  }
86
87  /**
88   * \fn int generate_sub_key(uint48_t* sub_key, uint32_t Ci, uint32_t Di)
89   * \brief Fonction qui génère les 16 sous clés en fonction des tours et de Ci et Di.
90   * \param *sub_key sous clés qui seront générées.
91   * \param Ci partie qui change à chaque tour.
92   * \param Di partie qui change à chaque tour.
93   * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
94   */
95  int generate_sub_key(uint48_t* sub_key, uint32_t Ci, uint32_t Di){
96      int i; uint8_t bit;
97      sub_key->bytes=0;
98      for(i=0;i<48;i++){
99          if(PC2[i]<=28){ //Ci
100             bit=get_bit_uint32_t_most(Ci, PC2[i]+4);
101         }
102         else { //Di
103             bit=get_bit_uint32_t_most(Di, PC2[i]-28+4);
104         }
105         if(set_bit_uint64_t(&sub_key->bytes, bit, 48-i))
106                 return 1;
107     }
108     return 0;
109 }
```

```
110
111 /**
112  * \fn int process_Ci_Di(KEY* key)
113  * \brief Fonction qui va générer Ci et Di en plusieurs tours.
114  * \param *key clé qui stockera les 16 sous clés.
115  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
116  */
117 int process_Ci_Di(KEY* key){
118     // initialisation de Vi
119     int Vi[16]; int i; uint32_t* Ci, *Di;
120     for(i=0;i<16;i++){
121         if((i==0)||(i==1)||(i==8)||(i==15)) Vi[i]=1;
122         else Vi[i]=2;
123     }
124     //génération des 16 sous clés
125     for(i=0;i<16;i++){
126         if(shift_Ci_Di(&(key->Ci), Vi[i]))
127             return 1;
128         if(shift_Ci_Di(&(key->Di), Vi[i]))
129             return 1;
130
131         Ci=&(key->Ci); Di=&(key->Di);
132         if(generate_sub_key(&(key->sub_key[i]), *Ci, *Di))
133             return 1;
134     }
135     return 0;
136 }
137
138 /**
139  * \fn int key_schedule (uint64_t* init, KEY* k)
140  * \brief Fonction qui crée les sous clés du DES.
141  * \param *init pointeur sur la clé initiale de 64 bits.
142  * \param *key structure représentant les sous clés DES.
143  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
144  */
145 int key_schedule (uint64_t* init, KEY* key){
146     uint64_t nouv=0; int i;
147     for(i=0;i<56;i++){
148         uint8_t t=get_bit_uint64_t_most((*init),PC1[i]);
149         if(set_bit_uint64_t(&nouv, t, 56-i))
150             return des_errno=ERR_KEY_SCHEDULE, 1;
151     }
152     (*init)=nouv;
153
154     if(init_C0_D0(key, *init))
155         return des_errno=ERR_KEY_SCHEDULE, 1;
156
157     if(process_Ci_Di(key))
158         return des_errno=ERR_KEY_SCHEDULE, 1;
159     return 0;
160 }
```

Listing 3 – key_schedule.c

6.4 festeil.c

```c
/**
 * \file feistel.c
 * \brief Représente les fonctions concernant le processus de Feistel du DES.
 * \author Clément CAUMES
 * */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include "../inc/manip_bits.h"
#include "../inc/errors.h"
#include "../inc/inner_function.h"
#include "../inc/key_schedule.h"
#include "../inc/constants.h"

/*
 * Constante IP
 */
int IP[64] = { 58, 50, 42, 34, 26, 18, 10, 2,
               60, 52, 44, 36, 28, 20, 12, 4,
               62, 54, 46, 38, 30, 22, 14, 6,
               64, 56, 48, 40, 32, 24, 16, 8,
               57, 49, 41, 33, 25, 17, 9, 1,
               59, 51, 43, 35, 27, 19, 11, 3,
               61, 53, 45, 37, 29, 21, 13, 5,
               63, 55, 47, 39, 31, 23, 15, 7 };

/*
 * Constante IP_inv
 */
int IP_inv[64] = { 40, 8, 48, 16, 56, 24, 64, 32,
                   39, 7, 47, 15, 55, 23, 63, 31,
                   38, 6, 46, 14, 54, 22, 62, 30,
                   37, 5, 45, 13, 53, 21, 61, 29,
                   36, 4, 44, 12, 52, 20, 60, 28,
                   35, 3, 43, 11, 51, 19, 59, 27,
                   34, 2, 42, 10, 50, 18, 58, 26,
                   33, 1, 41, 9, 49, 17, 57, 25 };

/**
 * \fn int process_permutation(uint64_t* data, int* IP)
 * \brief Fonction qui réalise la permutation initiale du DES.
 * \param *data input du DES.
 * \param *IP constante de permutation.
 * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
 */
int process_permutation(uint64_t* data, int* IP){
    int i;
    uint64_t output=0x00;
    for(i=0;i<64;i++){
        uint8_t bit=get_bit_uint64_t_most(*data,IP[i]);
        if(set_bit_uint64_t(&output, bit, 64-i))
```

```
54              return 1;
55      }
56      *data=output;
57      return 0;
58  }
59
60  /**
61   * \fn void build_L0_R0(uint64_t data, uint32_t* L0, uint32_t* R0)
62   * \brief Fonction qui construit L0 et R0.
63   * \param data input qui se trouve juste après la permutation initiale.
64   * \param *L0 partie de gauche initialisée.
65   * \param *R0 partie de droite initialisée.
66   * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
67   */
68  void build_L0_R0(uint64_t data, uint32_t* L0, uint32_t* R0){
69      *L0=data>>32;
70      data<<=32;
71      *R0=data>>32;
72  }
73
74  /**
75   * \fn void rebuild_R16_L16(uint64_t* data, uint32_t L16, uint32_t R16)
76   * \brief Fonction qui construit L16 et R16.
77   * \param *data output qui se trouve juste avant la permutation initiale.
78   * \param L16 partie de gauche.
79   * \param R16 partie de droite.
80   * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
81   */
82  void rebuild_R16_L16(uint64_t* data, uint32_t L16, uint32_t R16){
83      *data=R16;
84      *data<<=32;
85      *data|=L16;
86  }
87
88  /**
89   * \fn int process_round_1_15_encryption(uint32_t* Li, uint32_t* Ri, uint48_t Kiadd1
      )
90   * \brief Fonction qui réalise les rounds du tour 1 au tour 15 de chiffrement.
91   * \param *Li partie de gauche à modifier.
92   * \param *Ri partie de droite à modifier.
93   * \param Kiadd1 Ki+1 sous clé entrantedu tour.
94   * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
95   */
96  int process_round_1_15_encryption(uint32_t* Li, uint32_t* Ri, uint48_t Kiadd1){
97      uint32_t Liadd1 = *Ri;
98      if(inner_function(Kiadd1, Ri))
99          return 1;
100     *Ri= (*Li)^(*Ri); // R(i+1)=Li XOR F(K(i+1), Ri)
101     *Li=Liadd1; // L(i+1)=Ri
102     return 0;
103
104 }
105
106 /**
107  * \fn int process_round_16_encryption(uint32_t* L15, uint32_t* R15, uint48_t K16)
108  * \brief Fonction qui réalise le round du tour 16 de chiffrement.
```

```
109  * \param *L15 partie de gauche à modifier.
110  * \param *R15 partie de droite à modifier.
111  * \param K16 sous clé entrante du tour.
112  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
113  */
114 int process_round_16_encryption(uint32_t* L15, uint32_t* R15, uint48_t K16){
115     uint32_t L16 = *R15;
116     if(inner_function(K16, R15))
117             return 1;
118     *R15= (*L15)^(*R15); // R(16)=F(K16, R15) XOR L(15)
119     *L15=L16; // L(16)=R(15)
120     return 0;
121 }
122 
123 /**
124  * \fn int process_round_1_decryption(uint32_t* L0, uint32_t* R0, uint48_t K16)
125  * \brief Fonction qui réalise le round du tour 1 de dechiffrement.
126  * \param *L0 partie de gauche à modifier.
127  * \param *R0 partie de droite à modifier.
128  * \param K16 sous clé entrante du tour.
129  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
130  */
131 int process_round_1_decryption(uint32_t* L0, uint32_t* R0, uint48_t K16){
132     uint32_t R1=*R0;
133     uint32_t L1=*L0;
134     *R0=L1;
135     if(inner_function(K16, L0))
136         return 1;
137     *L0=(*L0)^(R1);
138     return 0;
139 }
140 
141 /**
142  * \fn int process_round_2_15_decryption(uint32_t* Li, uint32_t* Ri, uint48_t Ki)
143  * \brief Fonction qui réalise le round des tours 2 au tour 15 de dechiffrement.
144  * \param *Li partie de gauche à modifier.
145  * \param *Ri partie de droite à modifier.
146  * \param Ki sous clé entrante du tour.
147  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
148  */
149 int process_round_2_15_decryption(uint32_t* Li, uint32_t* Ri, uint48_t Ki){
150     uint32_t RiMin1 = *Li;
151     if(inner_function(Ki, Li)) //modification de Li
152         return 1;
153     *Li = (*Ri)^(*Li); //L(i+1)=R(i) XOR f(Li, Ki)
154     *Ri = RiMin1; //R(i+1)=L(i)
155     return 0;
156 }
157 
158 /**
159  * \fn int encryption_des(uint64_t* data, uint64_t key_64)
160  * \brief Fonction de chiffrement DES.
161  * \param *data qui sera modifié à la sortie.
162  * \param key_64 clé 64 bits entrante du DES.
163  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
164  */
```

```
165 int encryption_des(uint64_t* data, uint64_t key_64){
166     // permutation initiale
167     if(process_permutation(data, IP))
168         return des_errno=ERR_FEISTEL, 1;
169 
170     // division en L0 et R0
171     uint32_t Li, Ri;
172     build_L0_R0(*data, &Li, &Ri);
173 
174     // création des 16 sous clés
175     KEY key;
176     if(key_schedule (&key_64, &key))
177         return des_errno=ERR_FEISTEL, 1;
178 
179     // Schema de Feistel
180     int i=0;
181     for(i=0;i<15;i++){
182         if(process_round_1_15_encryption(&Li, &Ri, key.sub_key[i]))
183             return des_errno=ERR_FEISTEL, 1;
184     }
185     if(process_round_16_encryption(&Li, &Ri, key.sub_key[15]))
186         return des_errno=ERR_FEISTEL, 1;
187 
188     // permutation initiale inverse
189     rebuild_R16_L16(data, Li, Ri);
190     if(process_permutation(data, IP_inv))
191         return des_errno=ERR_FEISTEL, 1;
192 
193     return 0;
194 }
```

Listing 4 – feistel.c

6.5 inner_function.c

```c
/**
 * \file inner_function.c
 * \brief Représente les fonctions concernant la fonction intérieur du DES.
 * \author Clément CAUMES
 * */

#include <stdio.h>
#include <stdlib.h>

#include "../inc/key_schedule.h"
#include "../inc/errors.h"
#include "../inc/inner_function.h"
#include "../inc/manip_bits.h"
#include "../inc/attack.h"
#include "../inc/constants.h"

/*
 * Constante E.
 */
int 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};

/*
 * Constante S1.
 */
int S1[4][16] = { 14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7,
                   0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
                   4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
                  15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13};

/*
 * Constante S2.
 */
int S2[4][16] = { 15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10,
                   3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
                   0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
                  13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9};

/*
 * Constante S3.
 */
int S3[4][16] = { 10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8,
                  13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
                  13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
                   1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12};

/*
```

```
54   * Constante S4.
55   */
56  int S4[4][16] = { 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15,
57                   13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
58                   10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
59                    3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14};
60
61  /*
62   * Constante S5.
63   */
64  int S5[4][16] = { 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9,
65                   14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
66                    4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
67                   11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3};
68
69  /*
70   * Constante S6.
71   */
72  int S6[4][16] = { 12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11,
73                   10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
74                    9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
75                    4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13};
76
77  /*
78   * Constante S7.
79   */
80  int S7[4][16] = { 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1,
81                   13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
82                    1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
83                    6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12};
84
85  /*
86   * Constante S8.
87   */
88  int S8[4][16] = { 13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7,
89                    1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
90                    7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
91                    2,  1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11};
92
93  /*
94   * Constante P.
95   */
96  int P[32] = { 16, 7, 20, 21,
97                29, 12, 28, 17,
98                 1, 15, 23, 26,
99                 5, 18, 31, 10,
100                2, 8, 24, 14,
101               32, 27, 3, 9,
102               19, 13, 30, 6,
103               22, 11, 4, 25 };
104
105 /**
106  * \fn int expand(uint64_t* expand, uint32_t R)
107  * \brief Fonction d'expansion de Ri-1.
108  * \param *expand valeur d'expansion qui sera initialisée.
109  * \param R valeur d'entrée de la fonction d'expansion (Ri-1).
```

```
110  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
111  */
112 int expand(uint64_t* expand, uint32_t R){
113     *expand=0;
114     int i; uint8_t bit;
115     for(i=0;i<48;i++){
116         bit = get_bit_uint32_t_most(R, E[i]);
117         if(set_bit_uint64_t (expand, bit, 48-i))
118             return 1;
119     }
120     return 0;
121 }
122 
123 /**
124  * \fn int process_S_box(uint32_t* result, uint48_t elem)
125  * \brief Fonction d'utilisation des 8 SBOX.
126  * \param *result résultat de la procédure d'utilisation des SBOX.
127  * \param elem valeur d'entrée des SBOX.
128  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
129  */
130 int process_S_box(uint32_t* result, uint48_t elem){
131     *result=0;
132     int i, j, k, shift;
133     uint8_t bit1, bit2, input_sbox, x, y, val_sbox;
134     uint32_t output_shift;
135 
136     // pour chaque boite S on récupère l'entrée de celle ci et on modifie les bits
        en sortie.
137     for(i=0;i<8;i++){
138         x=y=0;
139         input_sbox = get_6bits_uint64_t_most (elem, i+1);
140         bit1 = get_bit_uint8_t(input_sbox, 6);
141         bit2 = get_bit_uint8_t(input_sbox, 1);
142         bit1<<=1;
143         y|=bit1;
144         y|=bit2;
145 
146         k=3;
147         for(j=5; j>1;j--){
148             bit1 = get_bit_uint8_t(input_sbox, j);
149             bit1<<=k;
150             x|=bit1;
151             k--;
152         }
153 
154         if(i==0) val_sbox=S1[y][x];
155         else if(i==1) val_sbox=S2[y][x];
156         else if(i==2) val_sbox=S3[y][x];
157         else if(i==3) val_sbox=S4[y][x];
158         else if(i==4) val_sbox=S5[y][x];
159         else if(i==5) val_sbox=S6[y][x];
160         else if(i==6) val_sbox=S7[y][x];
161         else if(i==7) val_sbox=S8[y][x];
162 
163         shift=(8-i-1)*4;
164         output_shift=val_sbox;
```

```
165             output_shift<<=shift;
166             *result|=output_shift;
167         }
168         return 0;
169     }
170 
171     /**
172      * \fn int permutation_inner_function(uint32_t* output, uint32_t input)
173      * \brief Fonction qui fait la permutation finale.
174      * \param *output valeur de sortie qui sera initialisée.
175      * \param input valeur d'entrée de la permutation.
176      * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
177      */
178     int permutation_inner_function(uint32_t* output, uint32_t input){
179         int i;
180         *output=0x00;
181         for(i=0;i<32;i++){
182             uint8_t bit=get_bit_uint32_t_most(input,P[i]);
183             if(set_bit_uint32_t(output, bit, 32-i))
184                 return 1;
185         }
186         return 0;
187     }
188 
189     /**
190      * \fn int permutation_inv_inner_function(uint32_t* output, uint32_t input)
191      * \brief Fonction qui fait la permutation finale inversée.
192      * \param *output valeur de sortie qui sera initialisée.
193      * \param input valeur d'entrée de la permutation inverse.
194      * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
195      */
196     int permutation_inv_inner_function(uint32_t* output, uint32_t input){
197         int i;
198         *output=0x00;
199         for(i=0;i<32;i++){
200             uint8_t bit=get_bit_uint32_t_most(input, i+1);
201             if (set_bit_uint32_t(output, bit, 33-P[i]))
202                 return 1;
203         }
204         return 0;
205     }
206 
207     /**
208      * \fn int get_input_sbox(uint8_t output, int S[4][16], uint8_t* input1, uint8_t*
           input2, uint8_t* input3, uint8_t* input4)
209      * \brief Boite S inverse.
210      * \param output valeur de sortie de la boite S initiale.
211      * \param S boite S.
212      * \param *input1 première valeur possible de l'entrée de la boite S.
213      * \param *input2 deuxième valeur possible de l'entrée de la boite S.
214      * \param *input3 troisième valeur possible de l'entrée de la boite S.
215      * \param *input4 quatrième valeur possible de l'entrée de la boite S.
216      * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
217      */
218     int get_input_sbox(uint8_t output, int S[4][16], uint8_t* input1, uint8_t* input2,
           uint8_t* input3, uint8_t* input4){
```

```
219     int tour=0;
220     uint8_t input; uint8_t i,j; uint8_t bit;
221     input=0;
222     for(i=0;i<4;i++){
223         for(j=0;j<16;j++){
224             input=0;
225             if (S[i][j]==output) {
226                 bit=get_bit_uint8_t(i,2);
227                 if (set_bit_uint8_t(&input, bit, 6))
228                     return 1;
229 
230                 bit=get_bit_uint8_t(i,1);
231                 if (set_bit_uint8_t(&input, bit, 1))
232                     return 1;
233 
234                 bit=get_bit_uint8_t(j,4);
235                 if (set_bit_uint8_t(&input, bit, 5))
236                     return 1;
237 
238                 bit=get_bit_uint8_t(j,3);
239                 if (set_bit_uint8_t(&input, bit, 4))
240                     return 1;
241 
242                 bit=get_bit_uint8_t(j,2);
243                 if (set_bit_uint8_t(&input, bit, 3))
244                     return 1;
245 
246                 bit=get_bit_uint8_t(j,1);
247                 if (set_bit_uint8_t(&input, bit, 2))
248                     return 1;
249 
250                 if (tour==0) *input1=input;
251                 else if (tour==1) *input2=input;
252                 else if (tour==2) *input3=input;
253                 else if (tour==3) *input4=input;
254                 tour++;
255             }
256         }
257     }
258     return 0;
259 }
260 
261 /**
262  * \fn uint8_t process_S_box_particular(uint8_t input, int S[4][16])
263  * \brief Fonction de la boite S.
264  * \param input entrée de la boite S.
265  * \param S boite S.
266  * \return renvoie la sortie de la boite S.
267  */
268 uint8_t process_S_box_particular(uint8_t input, int S[4][16]){
269     uint8_t bit1, bit2;
270     int x, y;
271 
272     bit1=get_bit_uint8_t(input, 6);
273     bit2=get_bit_uint8_t(input, 1);
274 
```

```
275     x=bit1; x<<=1; x|=bit2;
276     y=(0x1e)&input;
277     y>>=1;
278 
279     return (uint8_t)S[x][y];
280 
281 }
282 
283 /**
284  * \fn int inner_function(uint48_t uint48_t, uint32_t* R)
285  * \brief Fonction du DES.
286  * \param sub_key sous clé d'entrée de la fonction.
287  * \param *R valeur qui sera modifiée et sera la sortie de la fonction du DES.
288  * \return renvoie 0 en cas de réussite et 1 en cas d'échec.
289  */
290 int inner_function(uint48_t sub_key, uint32_t* R){
291 
292     // calcul de E(Ri)
293     uint64_t expande;
294     if(expand(&expande, *R))
295         return des_errno=ERR_INNER_FUNCTION, 1;
296 
297     // E(Ri) ^ Ki+1
298     sub_key.bytes ^= expande;
299 
300     // calcul des S-Box
301     uint32_t output_sbox;
302     if(process_S_box(&output_sbox, sub_key))
303         return des_errno=ERR_INNER_FUNCTION, 1;
304 
305     // Permutation
306     uint32_t final;
307     if(permutation_inner_function(&final, output_sbox))
308         return des_errno=ERR_INNER_FUNCTION, 1;
309     *R = final;
310 
311     return 0;
312 }
```

Listing 5 – inner_function.c

6.6 manip_bits.c

```
1  /**
2   * \file manip_bits.c
3   * \brief Représente les fonctions concernant la manipulation de structures bas
       niveau.
4   * \author Clément CAUMES
5   * */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <stdint.h>
10
11 #include "../inc/errors.h"
12 #include "../inc/key_schedule.h"
13
14
15 /**
16  * \fn uint8_t get_bit_uint64_t (uint64_t elem, uint8_t i)
17  * \brief Fonction qui permet d'obtenir le i ème bit de elem en sachant
18  * que le bit de poids faible est à la position i=1.
19  *
20  * \param i numéro du bit à obtenir.
21  * \param elem uint64_t dont l'on cherche le i ème bit.
22  * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
23  * i ème bit de elem.
24  */
25 uint8_t get_bit_uint64_t (uint64_t elem, uint8_t i){
26     if(i<0) return 2;
27     if(i>64) return 2;
28     return (elem>>(i-1))%2;
29 }
30
31 /**
32  * \fn uint8_t get_bit_uint32_t (uint32_t elem, uint8_t i)
33  * \brief Fonction qui permet d'obtenir le i ème bit de elem en sachant
34  * que le bit de poids faible est à la position i=1.
35  *
36  * \param i numéro du bit à obtenir.
37  * \param elem uint32_t dont l'on cherche le i ème bit.
38  * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
39  * i ème bit de elem.
40  */
41 uint8_t get_bit_uint32_t (uint32_t elem, uint8_t i){
42     if(i<0) return 2;
43     if(i>32) return 2;
44     return (elem>>(i-1))%2;
45 }
46
47 /**
48  * \fn uint8_t get_bit_uint8_t (uint8_t elem, uint8_t i)
49  * \brief Fonction qui permet d'obtenir le i ème bit de elem en sachant
50  * que le bit de poids faible est à la position i=1.
51  *
52  * \param i numéro du bit à obtenir.
```

```
53   * \param elem uint8_t dont l'on cherche le i ème bit.
54   * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
55   * i ème bit de elem.
56   */
57  uint8_t get_bit_uint8_t (uint8_t elem, uint8_t i){
58      if(i<0) return 2;
59      if(i>8) return 2;
60      return (elem>>(i-1))%2;
61  }
62
63  /**
64   * \fn uint8_t get_bit_uint64_t_most (uint64_t elem, uint8_t i)
65   * \brief Fonction qui permet d'obtenir le i ème bit de elem en sachant
66   * que le bit de poids fort est à la position i=1.
67   *
68   * \param i numéro du bit à obtenir.
69   * \param elem uint64_t dont l'on cherche le i ème bit.
70   * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
71   * i ème bit de elem.
72   */
73  uint8_t get_bit_uint64_t_most (uint64_t elem, uint8_t i){
74      if(i<0) return 2;
75      if(i>64) return 2;
76      return (elem>>(64-i))%2;
77  }
78
79  /**
80   * \fn uint8_t get_bit_uint32_t_most (uint32_t elem, uint8_t i)
81   * \brief Fonction qui permet d'obtenir le i ème bit de elem en sachant
82   * que le bit de poids fort est à la position i=1.
83   *
84   * \param i numéro du bit à obtenir.
85   * \param elem uint32_t dont l'on cherche le i ème bit.
86   * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
87   * i ème bit de elem.
88   */
89  uint8_t get_bit_uint32_t_most (uint32_t elem, uint8_t i){
90      if(i<0) return 2;
91      if(i>32) return 2;
92      return (elem>>(32-i))%2;
93  }
94
95  /**
96   * \fn uint8_t get_6bits_uint64_t_most (uint48_t elem, uint8_t i)
97   * \brief Fonction qui permet d'obtenir le i ème bloc de 6 bits de elem en sachant
98   * que le groupe de 6 bits de poids fort est à la position i=1.
99   *
100  * \param i numéro de du groupe de 6 octets à obtenir.
101  * \param elem uint64_t dont l'on cherche le i ème octet.
102  * \return renvoie 0, 1, (ou 2 en cas d'erreur) en fonction du
103  * i ème bit de elem.
104  */
105 uint8_t get_6bits_uint64_t_most (uint48_t elem, uint8_t i){
106     if(i<0) return 2;
107     if(i>8) return 2;
108     else{
```

```
109             uint8_t num_bloc = 9-i;
110             uint8_t result=0x00;
111             uint8_t bit = num_bloc*6-5;
112             uint8_t bit2;
113             int j;
114             for(j=0;j<6;j++){
115                 bit2=get_bit_uint64_t(elem.bytes, bit);
116                 bit2<<=j;
117                 result|=bit2;
118                 bit++;
119             }
120             return result;
121         }
122     }
123 
124     /**
125      * \fn int set_bit_uint64_t (uint64_t* elem, uint8_t bit, uint8_t pos)
126      * \brief Fonction qui initialise le bit à la position pos de elem.
127      * Le bit de poids faible a la position pos=1.
128      * \param elem élément à modifier
129      * \param bit valeur du futur bit à changer.
130      * \param pos position du bit à changer.
131      * \return renvoie 0 en cas de réussite et 1 en cas d'échec
132      * (change la valeur de l'erreur).
133      */
134     int set_bit_uint64_t (uint64_t* elem, uint8_t bit, uint8_t pos){
135         if(bit==1){
136             uint64_t mask=1ULL;
137             mask <<= (pos-1);
138             (*elem)|=mask;
139         }
140         else if(bit==0){
141             uint64_t mask1=0xFFFFFFFFFFFFFFFF;
142             uint64_t mask2=1ULL;
143             mask2 <<= (pos-1);
144             mask1 ^= mask2;
145             (*elem)&=mask1;
146         }
147         else {
148             return 1;
149         }
150         return 0;
151     }
152 
153     /**
154      * \fn int set_bit_uint32_t (uint32_t* elem, uint8_t bit, uint8_t pos)
155      * \brief Fonction qui initialise le bit à la position pos de elem.
156      * Le bit de poids faible a la position pos=1.
157      * \param elem élément à modifier
158      * \param bit valeur du futur bit à changer.
159      * \param pos position du bit à changer.
160      * \return renvoie 0 en cas de réussite et 1 en cas d'échec
161      * (change la valeur de l'erreur).
162      */
163     int set_bit_uint32_t (uint32_t* elem, uint8_t bit, uint8_t pos){
164         if(bit==1){
```

```
165         uint32_t mask=1ULL;
166         mask <<= (pos-1);
167         (*elem)|=mask;
168     }
169     else if(bit==0){
170         uint32_t mask1=0xFFFFFFFF;
171         uint32_t mask2=1ULL;
172         mask2 <<= (pos-1);
173         mask1 ^= mask2;
174         (*elem)&=mask1;
175     }
176     else {
177         return 1;
178     }
179     return 0;
180 }
181 
182 /**
183  * \fn int set_bit_uint8_t (uint8_t* elem, uint8_t bit, uint8_t pos)
184  * \brief Fonction qui initialise le bit à la position pos de elem.
185  * Le bit de poids faible a la position pos=1.
186  * \param elem élément à modifier
187  * \param bit valeur du futur bit à changer.
188  * \param pos position du bit à changer.
189  * \return renvoie 0 en cas de réussite et 1 en cas d'échec
190  * (change la valeur de l'erreur).
191  */
192 int set_bit_uint8_t (uint8_t* elem, uint8_t bit, uint8_t pos){
193     if(bit==1){
194         uint32_t mask=1ULL;
195         mask <<= (pos-1);
196         (*elem)|=mask;
197     }
198     else if(bit==0){
199         uint32_t mask1=0xFF;
200         uint32_t mask2=1ULL;
201         mask2 <<= (pos-1);
202         mask1 ^= mask2;
203         (*elem)&=mask1;
204     }
205     else {
206         return 1;
207     }
208     return 0;
209 }
210 
211 /**
212  * \fn void printf_uint64_t_binary(uint64_t key);
213  * \brief Fonction qui affiche en binaire un uint64_t.
214  *
215  * \param key uint64_t à afficher.
216  */
217 void printf_uint64_t_binary(uint64_t key){
218     char* chain; int i; uint8_t bit;
219     chain=malloc(65*sizeof(char));
220 
```

```
221     for(i=0;i<64;i++){
222         bit=key%2;
223         if(bit==0) chain[63-i]='0';
224         else chain[63-i]='1';
225         key>>=1;
226     }
227     chain[64]='\0';
228     printf("%s\n", chain);
229     free(chain);
230 }
231 
232 
233 /**
234  * \fn void printf_uint32_t_binary(uint32_t key)
235  * \brief Fonction qui affiche en binaire un uint32_t.
236  *
237  * \param key uint32_t à afficher.
238  */
239 void printf_uint32_t_binary(uint32_t key){
240     char* chain; int i; uint8_t bit;
241     chain=malloc(33*sizeof(char));
242     for(i=0;i<32;i++){
243         bit=key%2;
244         if(bit==0) chain[31-i]='0';
245         else chain[31-i]='1';
246         key>>=1;
247     }
248     chain[32]='\0';
249     printf("%s", chain);
250     free(chain);
251 }
252 
253 /**
254  * \fn void printf_uint8_t_binary(uint8_t key)
255  * \brief Fonction qui affiche en binaire un uint8_t.
256  *
257  * \param key uint8_t à afficher.
258  */
259 void printf_uint8_t_binary(uint8_t key){
260     char* chain; int i; uint8_t bit;
261     chain=malloc(9*sizeof(char));
262     for(i=0;i<8;i++){
263         bit=key%2;
264         if(bit==0) chain[7-i]='0';
265         else chain[7-i]='1';
266         key>>=1;
267     }
268     chain[8]='\0';
269     printf("%s", chain);
270     free(chain);
271 }
272 
273 /**
274  * \fn void printf_uint64_t_hexa(uint64_t key)
275  * \brief Fonction qui affiche en hexa un uint64_t.
276  *
```

```
277  * \param key uint64_t à afficher.
278  */
279 void printf_uint64_t_hexa(uint64_t key){
280     printf("%016llX", key);
281 }
282 
283 /**
284  * \fn void printf_uint8_t_hexa(uint8_t key)
285  * \brief Fonction qui affiche en hexa un uint8_t.
286  *
287  * \param key uint8_t à afficher.
288  */
289 void printf_uint8_t_hexa(uint8_t key){
290     printf("%x", key);
291 }
292 
293 
294 /**
295  * \fn void printf_uint32_t_hexa(uint32_t key)
296  * \brief Fonction qui affiche en hexa un uint32_t.
297  *
298  * \param key uint32_t à afficher.
299  */
300 void printf_uint32_t_hexa(uint32_t key){
301     printf("%16lX", key);
302 }
```

Listing 6 – manip_bits.c

6.7 errors.c

```c
/**
 * \file errors.c
 * \brief Représente les fonctions concernant la gestion des erreurs.
 * \author Clément CAUMES
 * */

#include <stdio.h>
#include <stdlib.h>

#include "../inc/errors.h"

/* Initialisation. */
enum err_code des_errno = ERR_NONE;

/**
 * \fn void err_print(enum err_code err)
 * \brief Fonction d'affichage de l'erreur lancée.
 *
 * \param err numéro de l'erreur lancée.
 */
void err_print(enum err_code err)
{
    static const char *err_desc[] = {
        /* ERR_NONE */                  "aucune erreur",
        /* ERR_KEY_SCHEDULE */          "erreur génération des sous clés"
        /* ERR_INNER_FUNCTION */        "erreur de la fonction intérieure"
        /* ERR_FEISTEL */               "erreur de feistel"
        /* ERR_ATTACK */                "erreur lors de l'attaque DFA"
        /* ERR_OTHER */                 "erreur inconnu"
    };

    /* Vérification de la valeur de "err". */
    err = (unsigned int)err <= ERR_OTHER ? err : ERR_OTHER;
    fprintf(stderr, "Erreur %d : %s.\n", err, err_desc[err]);
}
```

Listing 7 – errors.c