Contrôle continu de calcul sécurisé

Attaque par faute sur le DES

BARBIER Yann 21800844

Table des matières

Question 1 :

Le DES est un algorithme de chiffrement par blocs organisé en réseau de Feistel utilisant des clés maître de 64 bits, dont 1 bit sur 8 n'est pas utilisé (en partant de 1, ce sont les bits 8, 16, 24, 32, 40, 48, 56 et 64) , ce qui en fait une clé maître effective de 56 bits.
Cette clé maître est utilisée par un algorithme de cadencement de clé produisant 16 sous-clés de 48 bits (une par tour).

Une attaque par injection de faute consiste à modifier la valeur d'un bit (par exemple au moyen d'un tir de laser) à un certain moment lors du chiffrement d'un message. Ainsi, l'algorithme de chiffrement donne en sortie un chiffré faux.

Ici, la faute est injectée sur la valeur de sortie de R_{15} du 15ème tour de notre réseau de Feistel.
Nous utiliserons les 32 messages faux pour réduire la complexité d'une attaque par rapport à la recherche exhaustive de complexité $O(2^{56})$, en trouvant la sous-clé K_{16} qui nous donnera des informations sur la clé maître.

Par la suite, IP désignera la fonction de permutation initiale et IP^{-1} son inverse.
Voici les équations décrivant les sorties des tours L_{16} et R_{16} dans le cas correct et le cas faux (" indique le cas faux) :

Cas correct :
$R_{16} = R_{15}$
$L_{16} = L_{15}$ xor $F(R_{15}, K_{16})$

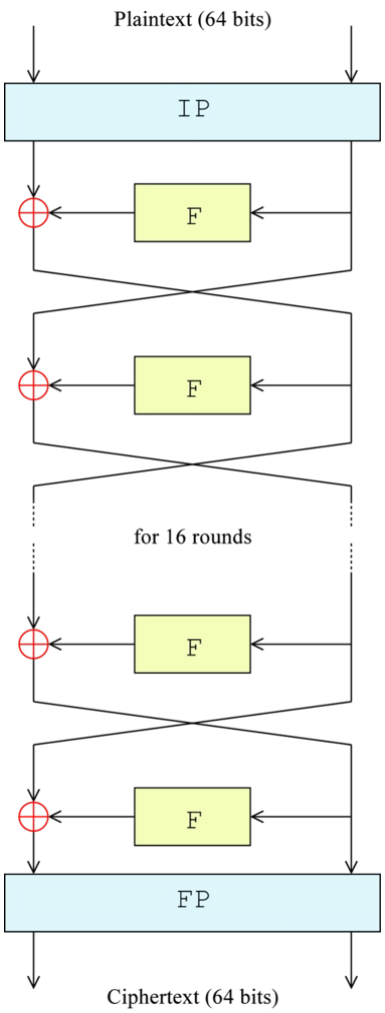
Cas faux :
$R_{16}" = R_{15}"$
$L_{16}" = L_{15}$ xor $F(R_{15}", K_{16})$

Il faut noter que la faute de $R_{15}"$ va se propager vers $L_{15}"$ à travers la fonction F. Ce qui signifie que :
$R_{16}" = R_{15}" = R_{16}$ xor faute $= R_{15}$ xor faute
$L_{16}" = L_{16}$ xor faute" $= L_{15}$ xor $F(R_{15}$ xor faute, $K_{16})$

où faute et faute" sont différents.
Et voici les équations décrivant les chiffrés correct et faux :

Cas correct :
$C = IP^{-1}(R_{16} \| L_{16})$

Cas faux :
$C" = IP^{-1}(R_{16}" \| L_{16}")$



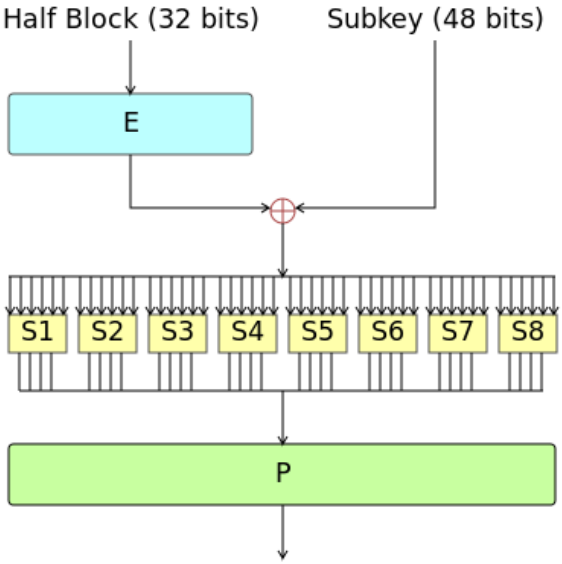
Fonctionnement du DES (source : wikipedia commons)

Nous pouvons à présent commencer à remonter le réseau de Feistel (voir schéma du fonctionnement du DES) :

$R_{16} = IP^{-1}(C_{0\text{-}31})$
$L_{16} = IP^{-1}(C_{32\text{-}63})$
$R_{16}" = IP^{-1}(C"_{0\text{-}31})$
$L_{16}" = IP^{-1}(C"_{32\text{-}63})$

Par combinaison :
R_{16} xor $R_{16}" = R_{16}$ xor R_{16} xor faute = faute
L_{16} xor $L_{16}" = L_{16}$ xor L_{16} xor faute" = faute"

Pour obtenir faute" en fonction de R_{15}, on développe cette dernière équation :
faute" = L_{16} xor $L_{16}"$
faute" = L_{15} xor $F(R_{15}, K_{16})$ xor L_{15} xor $F(R_{15}", K_{16})$
faute" = $F(R_{15}$ xor faute, $K_{16})$ xor $F(R_{15}, K_{16})$



Fonctionnement de la fonction F (source : wikipedia commons)

Pour obtenir K_{16}, il faut savoir comment fonctionne la fonction F (voir schéma ci-contre). Un bloc de 32 bits (ici R_{15}) est étendu en un bloc de 48 bits en dupliquant les bits adjacents à chaque blocs de 4 bits (formant 8 blocs de 6 bits), puis xoré avec une sous-clé de 48 bits (ici K_{16}).
Les blocs de 6 bits passent par des S-box non linéaires et donnent en sortie des blocs de 4 bits, nous redonnant un bloc complet de 32 bits, avant que ce bloc complet ne subisse une permutation de ses blocs de 4 bits.
Nous appellerons E la fonction d'expansion, Sbox la fonction de substitution par les Sboxes et P la fonction de permutation de F.

Ainsi :
$F(R_{15}$ xor faute, $K_{16}) = P(Sbox(E(R_{15}$ xor faute) xor $K_{16}))$
$F(R_{15}, K_{16}) = P(Sbox(E(R_{15}) \text{ xor } K_{16}))$

P est une fonction linéaire, donc on peut calculer P^{-1} et l'appliquer à faute".

faute" = $P(Sbox(E(R_{15}$ xor faute) xor $K_{16})) \text{ xor } P(Sbox(E(R_{15}) \text{ xor } K_{16}))$
P^{-1}(faute") = $Sbox(E(R_{15}$ xor faute) xor $K_{16}) \text{ xor } Sbox(E(R_{15}) \text{ xor } K_{16})$

Connaissant P^{-1}(faute"), R_{15} et R_{15} xor faute, nous pouvons en déduire $E(R_{15})$ et $E(R_{15}$ xor faute).
En ce qui concerne les Sbox, elles ne sont pas linéaires. On peut procéder à une recherche exhaustive sur 6 bits pour retrouver l'entrée de chacune des 8 Sbox grâce à leur sortie, et donc procéder à un calcul de complexité 6 x 8 x 2^6, soit $O(2^{10})$ au lieu de $O(2^{48})$ en cherchant la sous-clé.
Pour déterminer si chaque bloc de 6 bits de la sous-clé K_{16} est juste, il nous faudra suffisamment de R_{15}", 32 dans le cas où l'on ne commet une faute que sur un seul bit de R_{15}, c'est justement le nombre de chiffrés faux dont on dispose.

En résumé :
1) On applique IP^{-1} à notre chiffré correct et à nos chiffrés faux pour obtenir R_{15} et 32 R_{15}".

2) Pour chaque chiffré faux, on établit 8 équations telles que :

$P\text{-}1(R_{15} \text{ xor } R_{15}')_{\text{bit i} \rightarrow \text{bit i+3}} = Sbox_k(E(R_{15}) \text{ xor } K_{16})_{\text{bit i} \rightarrow \text{bit i+3}} \text{ xor } Sbox_k(E(R_{15}') \text{ xor } K_{16})_{\text{bit i} \rightarrow \text{bit i+3}}$

où i part de 0 et est incrémenté de 4 jusqu'à 27 et k part de 0 et est incrémenté de 1 jusqu'à 8.

3) On élimine les équations pour lesquelles $P\text{-}1(R_{15} \text{ xor } R_{15}')_{\text{bit i} \rightarrow \text{bit i+3}} = 0$ car elles ne sont pas marquées par l'erreur.

4) Pour chaque $Sbox_k(E(R_{15}) \text{ xor } K_{16})_{\text{bit i} \rightarrow \text{bit i+3}}$, connaissant R_{15}, on déduit les possibles valeurs d'entrée des bits concernés de K_{16} qui nous donnent la bonne sortie avec une attaque exhaustive.

5) Pour tous les K_{16} possibles trouvés, on calcule $Sbox_k(E(R_{15}') \text{ xor } K_{16})_{\text{bit i} \rightarrow \text{bit i+3}}$ et on vérifie que l'équation de l'étape 2 soit vérifiée. Si tel est le cas, alors ce K_{16} est enregistré pour pouvoir retrouver la clé maître K.

6) On compare les portions de K_{16} en entrée de chaque Sbox avec celles des autres chiffrés faux, et on ne garde que les portions qui se retrouvent systématiquement. On peut alors les concaténer pour obtenir K_{16}.

Question 2 :

Partie programmation. Nous coderons nos algorithmes en c.
Le code complet est disponible sur github (voir fin du rapport). Nous ne nous intéressons ici qu'à la fonction principale permettant de retrouver K_{16} sans nous intéresser aux fonctions annexes :

```
uint64_t rechercheK16(uint64_t chiffreCorrect, uint64_t *chiffresFaux)
{
  Message juste;
  Message faux;
  int resultat[NOMBRE_SBOXES][TAILLE_BLOC] = {0};
  bool LPinverse[TAILLE_DEMI_BLOC] = {0};
  bool resultatXorL[TAILLE_DEMI_BLOC] = {0};

  obtenirR16L16(chiffreCorrect, &juste);

  for(int w = 0; w < TAILLE_DEMI_BLOC; w++)
    {
    obtenirR16L16(chiffresFaux[w], &faux);
    xor(resultatXorL, juste.LChiffreBin, faux.LChiffreBin, TAILLE_DEMI_BLOC);
    permutation(LPinverse, resultatXorL, Pinverse, TAILLE_DEMI_BLOC);

    uint64_t bitFaux = bitFaute(juste.RChiffreBin, faux.RChiffreBin);
    permutation(juste.RChiffreBinE, juste.RChiffreBin, E, TAILLE_SOUS_CLE);
    permutation(faux.RChiffreBinE, faux.RChiffreBin, E, TAILLE_SOUS_CLE);

    bool resSbox[TAILLE_SORTIE_SBOX] = {0};
    bool resLeftJuste[TAILLE_SORTIE_SBOX] = {0};
    bool cle[TAILLE_ENTREE_SBOX] = {0};

    for(int i = 0;  i < TAILLE_SOUS_CLE; i++)
       {
      if(E[i] == (bitFaux + 1))
           {
        extraire6Bits(&juste, i / TAILLE_ENTREE_SBOX);
```

```
          extraire6Bits(&faux, i / TAILLE_ENTREE_SBOX);

          for(int y = 0; y < TAILLE_SORTIE_SBOX; y++)
                {
            resLeftJuste[y] = LPinverse[TAILLE_SORTIE_SBOX * (i /
TAILLE_ENTREE_SBOX) + y];
          }

          for(int j = 0;  j < TAILLE_BLOC; j++)
                {
            decimalEnBin(cle, j, TAILLE_ENTREE_SBOX);
            xor(juste.Sbox6BitsXoreBin, juste.Sbox6BitsBin, cle,
TAILLE_ENTREE_SBOX);
            decimalEnBin(cle, j, TAILLE_ENTREE_SBOX);
            xor(faux.Sbox6BitsXoreBin, faux.Sbox6BitsBin, cle,
TAILLE_ENTREE_SBOX);
            SboxFonction(juste.Sbox4BitsBin, juste.Sbox6BitsXoreBin, i /
TAILLE_ENTREE_SBOX);
            SboxFonction(faux.Sbox4BitsBin, faux.Sbox6BitsXoreBin, i /
TAILLE_ENTREE_SBOX);
            xor(resSbox,juste.Sbox4BitsBin, faux.Sbox4BitsBin,
TAILLE_SORTIE_SBOX);

            if(TEgaux(resLeftJuste, resSbox, TAILLE_SORTIE_SBOX))
                  {
              resultat[i / TAILLE_ENTREE_SBOX][TEnUint64(cle,
TAILLE_ENTREE_SBOX)]++;
            }
          }
        }
      }
    }

  return K16EnUint64(resultat);
}
```

Nous décrirons cette fonction par étapes. Tout d'abord, elle prend en paramètre le chiffré correct et un tableau de chiffrés faux, et renvoie la valeur de K_{16}.
Nous commençons par extraire L16 et R16 du chiffré correct en procédant à une IP puis en séparant le message obtenu (stocké dans la structure Message juste).
Ensuite, et pour chaque chiffré faux, nous obtenons L16" et R16" (stocké dans la structure Message faux) et nous effectuons un xor avec leurs équivalents correctes L16 et R16.
Nous effectuons une permutation P^{-1} sur le résultat du xor précédent noté resultatXorL
Nous récupérons ensuite l'indice du but faux en comparant la partie droite (R) du chiffré binaire juste et celui faux, avant de faire effectuer à chacune de ces variables une expansion E.
Ensuite, nous traversons les indices des positions de E de telle sorte que si E indique que le bit i doit se trouver à la position j du bit faux (j – 1 en partant de 0), nous pouvons alors extraire les 6 bits d'entrée de la Sbox concernée par l'indice i que l'on vérifiera par une recherche exhaustive. Il ne nous reste alors plus qu'à répéter cette recherche exhaustive à toutes les Sbox puis à les assembler.

Valeur de K_{16} trouvée en hexadécimal :
`22a29bd77888`

Question 3 :

L'action de récupérer la clé maître en partant de la sous-clé K_{16} dépend de l'algorithme de cadencement de clé. Voici notre implémentation de la recherche de K56bits :

```
uint64_t rechercheK56Bits(uint64_t clair, uint64_t chiffre, uint64_t K16)
{
  Cle k;

  initT(k.cle48Bin, TAILLE_SOUS_CLE);
  initT(k.cle56Bin, TAILLE_CLE_MAITRE_SANS_PARITE);
  initT(k.cle64Bin, TAILLE_BLOC);
  uint64EnBin(k.cle48Bin, K16,12);
  permutation(k.cle56Bin, k.cle48Bin, PC2Inverse, 56);
  permutation(k.cle64Bin, k.cle56Bin, PC1Inverse, 64);

  for(int i = 0; i < (int)pow((double)2, (double)NOMBRE_SBOXES); i++)
    {
     decimalEnBin(k.cle8Bin, i, NOMBRE_SBOXES);

     for(int j = 0; j < NOMBRE_SBOXES; j++)
        {
       k.cle64Bin[position8bit[j] - 1] = k.cle8Bin[j];
     }

     uint64_t cle = TEnUint64(k.cle64Bin, TAILLE_BLOC);

     if(chiffre == fonctionDES(clair, cle))
        {
       return cle;
     }
  }

  return 0;
}
```

Cette fonction prend en paramètre le message clair, le chiffré correct et K_{16} précédemment calculé pour renvoyer K56 la clé maître de 56 bits.
Nous commençons par initialiser notre structure Cle k contenant les différentes clés dont nous aurons besoin lors du calcul.
Nous effectuons une permutation $PC2^{-1}$ sur k.cle48Bin, la conversion en binaire (tableau de booléens) de K_{16}, avant d'y effectuer une permutation $PC1^{-1}$.
Nous n'avons plus qu'à effectuer une remontée de l'algorithme de cadencement de clés, puis tester notre clé maître en chiffrant notre message clair et en comparant le résultat avec le chiffré correct.

Valeur de K56 trouvée en hexadécimal :
`cd60a664cd0b24c`

Pour compléter, nous avons également calculé la valeur de K64 que voici, toujours en hexadécimal. Il ne nous restait plus qu'à compléter K56 avec les bits de parité :
`dd60b674cd0b34c`

Question 4 :

En effectuant une attaque par faute sur R_{15}, et sachant que nous avons 8 Sbox, nous devons résoudre un problème de complexité $0(2^{10})$ opérations pour trouver K_{16}.

Si l'erreur est effectuée sur R_{14}, l'erreur se propageant, il faut alors faire une recherche exhaustive des bits faux corresondant à la sortie de F du tour précédent, ce qui multiplie le temps de calcul par 2 pour chaque bit faux supplémentaire, soit par 2^4. Nous devons alors résoudre un problème de complexité $2^{10} * 2^4$ soit $O(2^{14})$.
Cette complexité est alors à multiplier par elle-même lorsque l'attaque est effectuée sur le tour précédent, soit une mulitplication par 2 de la puissance.
Au 14ème tour, la complexité vaut : 2^{28}
Au 13ème tour, la complexité vaut : 2^{56}, soit équivalente à la recherche exhaustive de la clé maître.
Au 12ème tour la complexité vaut : 2^{112}, c'est un calcul trop complexe en temps raisonnable.

Question 5 :

Plusieurs moyens existent pour se prémunir d'une attaque par injection de faute.
Tout d'abord, comprenons qu'une telle attaque se réalise physiquement en perturbant le matériel requis lors du chiffrement. Un bon moyen de se prémunir serait alors de protéger les composants et d'installer des verous permettant d'annuler le chiffrement en cas de détections d'altération des composants, de changements brutaux de l'alimentation en énergie etc…
De même, il est possible d'effectuer plusieurs fois le calcul pour vérifier une constance dans les résultats obtenus. Cette solution est en revanche coûteuse car elle mulitplie le temps du chiffrement.
Sachant qu'une attaque par faute est de plus en plus difficile à exploiter lorsque l'erreur est faite sur les premiers tours du DES, nous pouvons aussi décider de ne reproduire que les derniers tours pour vérifier qu'aucune faute n'y ait été introduite sans trop nuire au temps de calcul nécessaire au chiffrement.

Partie code

Le code complet est disponible à l'adresse github suivante :
https://github.com/YannBgit/AttaqueParFauteDES

Le fonctionnement du programme et les instructions pour le lancer sont disponibles dans le README.md. Le code se situe dans le répertoire /src tandis que le message clair, le chiffré correct et les chiffrés faux sont écrits dans le fichier texte du répertoire /ressources.