TP : Cryptanalyses différentielle et linéaire du DES

Sébastien Josse

L'objectif de ce TP est d'étudier l'implémentation et la cryptanalyse de l'algorithme DES.

L'implémentation étudiée est celle qui a été produite par D. M. Balenson du NBS/ICST (National Bureau of Standards, Institute for Computer Sciences and Technology) en Septembre 1986.
Les données étalons utilisées pour vérifier la conformité de cette implémentation à ses spécifications proviennent du FIPS PUB (Federal Information Processing Standards Publication) 46 (15 Janvier 1977) fournie par le NIST.

Nous étudions en premier lieu les techniques de **cryptanalyse différentielle** et **linéaire** dans le cadre d'**attaques sur le dernier tour** de versions modifiées du DES comportant un nombre réduit de tours (3/16 et 6/16 pour les cryptanalyses différentielles, 4/16 pour la cryptanalyse linéaire.

1 Implémentation de l'algorithme DES

1.1 Diversification de la clé

La clé K est une chaîne de 64 bits dont 56 définissent la clé ($K \in \mathbb{F}_2^{56}$), et 8 sont des bits de parité (les bits en position 7, 15, ..., 63 sont tels que chaque octet contient un nombre impair de 1). Cette clé est diversifiée en 16 clés de tour K_r, $r = 0, \ldots, 15$ de 48 bits.
Les bits de parité sont ignorés dans le procédé de diversification : étant donnés les 64 bits de la clé K, on enlève les bits de parité et l'on ordonne les autres suivant une permutation `PC1`. On note $\texttt{PC1}(K) = C_0 D_0$, où C_0 est composé des 28 premiers bits de $\texttt{PC1}(K)$ et D_0 des 28 bits restants :

```
void setkey(BYTE *pkey)
{
        INT i, j, k, t1, t2;
        static BYTE key[64];
        static BYTE CD[56];

        unpack8(pkey,key);

        for (i=0; i<56; i++) CD[i] = key[PC1[i]-1];
        ...
```

Ensuite, pour $i = 0, \ldots, 15$, on calcule : $C_{i+1} = \texttt{shift}_i(C_i)$, $D_{i+1} = \texttt{shift}_i(D_i)$ et $K_i = \texttt{PC2}(C_{i+1} D_{i+1})$:

```
        ...
        for (i=0; i<16; i++) {
                for (j=0; j<shifts[i]; j++) {
                        t1 = CD[0];
                        t2 = CD[28];
                        for (k=0; k<27; k++) {
                                CD[k] = CD[k+1];
                                CD[k+28] = CD[k+29];
                        }
                        CD[27] = t1;
```

```
                        CD[55] = t2;
                }
                for (k=0; k<48; k++) KS[j][k] = CD[PC2[k]-1];
        }
}
```

où $\texttt{shift}_i$ est une rotation circulaire d'une ou deux position suivant la valeur de i : on décale d'une position si $i = 0, 1, 8, 15$, et on décale de deux positions sinon :

```
unsigned short shifts[] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };
```

On obtient donc l'implémentation suivante de l'algorithme de diversification de clé :

```
void setkey(BYTE *pkey)
{
        INT i, j, k, t1, t2;
        static BYTE key[64];
        static BYTE CD[56];

        unpack8(pkey,key);

        for (i=0; i<56; i++) CD[i] = key[PC1[i]-1];

        for (i=0; i<16; i++) {
                for (j=0; j<shifts[i]; j++) {
                        t1 = CD[0];
                        t2 = CD[28];
                        for (k=0; k<27; k++) {
                                CD[k] = CD[k+1];
                                CD[k+28] = CD[k+29];
                        }
                        CD[27] = t1;
                        CD[55] = t2;
                }
                for (k=0; k<48; k++) KS[j][k] = CD[PC2[k]-1];
        }
}
```

Exercice 1 : pour coder la fonction permettant de retrouver (par recherche exhaustive) la clé K à partir de la donnée de la clé de tour K_r pour un r donné, on doit en premier lieu être en mesure de retrouver l'emplacement des 48 bits de K_r dans K (ou de manière équivalente les indices des bits inconnus de K) obtenus par inversion de l'algorithme de cadencement de clé (`setkey`) :

```
        void reverse_key_schedule(BYTE r, BYTE Kr[], BYTE key[]);
```

A) Programmer cet algorithme.
B) En déduire le code permettant de retrouver les indices (hors indices correspondants aux bits de parité) des bits de K dont la valeur devra être recherchée par force brute.
C) Donner également le code permettant de retrouver les bits de parité.

1.2 Algorithme principal

Soit $M = (x, y) \in \mathbb{F}_2^{64}$ un message clair, une permutation initiale IP est appliquée à (x, y) : $(x_0, y_0) = \mathtt{IP}(x, y)$, $x_0, y_0 \in \mathbb{F}_2^{32}$.

```
void DES(BYTE *in, BYTE *out) {
        static BYTE block[64];
        static BYTE LR[64];
        unpack8(in,block);                              // BYTE --> 8 BITS
        for (j=0; j<64; j++) LR[j] = block[IP[j]-1]; // permutation initiale
        ...
```

Une fonction g_{K_r} est appliquée à chaque tour r de la manière suivante :

$$(x_r, y_r) = g_{K_r}(x_{r-1}, y_{r-1}) = (y_{r-1}, x_{r-1} \oplus f_{K_r}(y_{r-1})).$$

```
        for (i=0; i<16; i++) {
                ...
                for (j=0; j<32; j++) {
                      t = LR[j+32];
                      LR[j+32] = LR[j] ^ f[j];
                      LR[j] = t;
                }
       }
```

Une permutation finale est appliquée.

```
        ...
        for (j=0; j<64; j++) block[j] = LR[RFP[j]-1]; // permutation finale
        pack8(out,block);                               // 8 BITS --> BYTE
}
```

Si on note i la fonction définie par $i(x, y) = (y, x)$, on a : $(x', y') = \mathtt{IP}^{-1}(y_{16}, x_{16}) = \mathtt{IP}^{-1} \circ i(x_{16}, y_{16})$. Le chiffrement correspond donc à l'opération :

$$(x', y') = \mathtt{IP}^{-1} \circ i \circ g_{K_{16}} \circ \ldots \circ g_{K_1} \circ \mathtt{IP}(x, y).$$

Exercice 2 : démontrer que l'algorithme de déchiffrement est le même que celui de l'algorithme de chiffrement, moyennant l'inversion de l'ordre des sous-clés.

La fonction f_{K_r} introduit une couche non linéaire dans chaque tour r de l'algorithme DES. Elle prend en entrée $R \in \mathbb{F}_2^{32}$, lui applique une fonction d'expansion E, avant de l'additionner modulo 2 à la clé de tour $K_r \in \mathbb{F}_2^{48}$.

```
        for (i=0; i<16; i++) {
                for (j=0; j<48; j++) preS[j] = LR[E[j]+31] ^ KS[i][j];
                ...
```

Le résultat $\mathtt{E}(R) \oplus K_r$ est séparé en 8 blocs de 6 bits $B_i = b_0 \ldots b_5$, $i = 1, \ldots, 8$. Pour chaque bloc B_i, le calcul $S_i(B_i) = \mathtt{S\_Box}_i[b_0 b_5][b_1 b_2 b_3 b_4] = c_0 c_1 c_2 c_3 = C_i$ est effectué.

```
                ...
                for (j=0; j<8; j++) {
                     k = 6*j;
```

```
                    t = preS[k];
                    t = (t<<1) | preS[k+5];
                    t = (t<<1) | preS[k+1];
                    t = (t<<1) | preS[k+2];
                    t = (t<<1) | preS[k+3];
                    t = (t<<1) | preS[k+4];
                    t = S[j][t];
                    k = 4*j;
                    f[k] = (t>>3) & 1;
                    f[k+1] = (t>>2) & 1;
                    f[k+2] = (t>>1) & 1;
                    f[k+3] = t & 1;
                }
                ...
```

Une permutation P est effectué à la fin de chaque tour. On obtient l'implémentation suivante pour le DES :

```
void DES(BYTE *in, BYTE *out) {
        static BYTE block[64];
        static BYTE LR[64];
        unpack8(in,block);

        for (j=0; j<64; j++) LR[j] = block[IP[j]-1]; // permutation initiale

        for (i=0; i<16; i++) {
                for (j=0; j<48; j++) preS[j] = LR[E[j]+31] ^ KS[i][j];

                for (j=0; j<8; j++) {
                     k = 6*j;
                     t = preS[k];
                     t = (t<<1) | preS[k+5];
                     t = (t<<1) | preS[k+1];
                     t = (t<<1) | preS[k+2];
                     t = (t<<1) | preS[k+3];
                     t = (t<<1) | preS[k+4];
                     t = S[j][t];
                     k = 4*j;
                     f[k] = (t>>3) & 1;
                     f[k+1] = (t>>2) & 1;
                     f[k+2] = (t>>1) & 1;
                     f[k+3] = t & 1;
                }

                for (j=0; j<32; j++) {
                      t = LR[j+32];
                      LR[j+32] = LR[j] ^ f[P[j]-1];
                      LR[j] = t;
                }
       }

        for (j=0; j<64; j++) block[j] = LR[RFP[j]-1]; // permutation finale
        pack8(out,block);                             // 8 BITS --> BYTE
}
```

Si on note $S = (S_1, \ldots, S_8)$ et K_r l'opération $K_r(A) = A \oplus K_r$, on a donc :

$$f_{K_r} = \mathtt{P} \circ S \circ K_r \circ \mathtt{E}.$$

Exercice 3 : compléter l'implémentation fournie (fonctions `pack` et `unpack`, permettant de transformer un tableau de 8 octets codants 8 bits par octet en un tableau de 64 octets codants 1 bit par octet et inversement :

```
void pack8(BYTE *packed,BYTE *binary);
void unpack8(BYTE *packed,BYTE *binary);
```

).

Vérifier la conformité de votre implémentation à ses spécifications à l'aide des données étalon suivantes (en base 16) :

$IN =$ `01 23 45 67 89 AB CD EF`,
$KEY =$ `13 34 57 79 9B BC DF F1`
$OUT =$ `85 E8 13 54 0F 0A B4 05`

Coder à partir de la fonction de chiffrement la fonction de déchiffrement. Vérifier que votre implémentation de la fonction de déchiffrement est correcte en utilisant les données étalons.

Exercice 4 : modifier l'implémentation de la fonction DES standard pour limiter à r le nombre de tours.

```
void des(BYTE in[], BYTE out[], INT rounds);
```

Exercice 5 : en utilisant cette fonction et les résultats de l'exercice 1, coder la fonction permettant de retrouver (par recherche exhaustive) la clé K à partir de la donnée de la clé de tour K_r pour un r donné (et d'un couple clair/chiffré obtenu par l'algorithme DES réduit à r tours avec la clé K).

2 Attaques sur le dernier tour

Les **attaques sur le dernier tour** d'un algorithme de chiffrement itératif sont fondées sur une étude du chiffrement réduit G, c'est-à-dire de la fonction de chiffrement F amputée de sa dernière itération F_{k_r}. Il est possible de retrouver la sous-clé utilisée au dernier tour dès lors que l'on dispose d'un moyen (un filtre ou détecteur de chiffrement réduit) pour distinguer le chiffrement réduit G d'une permutation aléatoire. L'algorithme d'une telle attaque est le suivant : pour chaque couple (m, c) de clair/chiffrés, pour chacune des valeurs possibles k de la sous-clé k_r, calculer $y = F_k^{-1}(c)$, puis :

- appliquer le détecteur au couple (m, y).
  - Si $k = k_r$, on a : $y = F_k^{-1}(F(m)) = G(m)$.
  - Sinon, y est l'image par m d' une permutation aléatoire.
- Si le chiffrement réduit est détecté, k est un candidat pour k_r.

Dans le cas de la **cryptanalyse différentielle**, le détecteur exploite le fait qu'il existe a et b tels que $G(x \oplus a) \oplus G(x) = b$ pour une grande proportion des valeurs de x. L'attaque est une attaque à clair choisi.

Dans le cas de la **cryptanalyse linéaire**, le détecteur exploite le fait qu' il existe $i_1, \ldots, i_J, j_1, \ldots, j_J$ tels que :

$$x[i_1] \oplus \ldots \oplus x[i_I] \oplus G(j_1) \oplus \ldots \oplus G(j_J) = e(k_1, \ldots, k_{r-1})$$

pour une grande proportion des valeur de x. L'attaque est une attaque à clair connu.

2.1 Cryptanalyse différentielle

La cryptanalyse différentielle est une attaque à clair choisi. Cette attaque exploite le fait suivant : soit $\Delta(B')$ l'ensemble des couples (B, B^*) en entrée d'une boîte de confusion S, dont le ou-exclusif vaut B'. Si pour ces couples on calcule le ou-exclusif C' de $S(B)$ et $S(B^*)$, on observe une distribution non-uniforme des ou-exclusifs de sortie sur les valeurs possibles.

Considérons le dernier tour de l'algorithme DES. L'attaquant construit l'ensemble $IN(E', C')$ des B tels que le ou-exclusif de $S(B)$ et $S(B \oplus E')$ égale C'. On vérifie facilement que J appartient à $E \oplus IN(E', C')$. Ce dernier ensemble constitue donc un ensemble de sous-clés J candidates. En corrélant les ensembles obtenus pour un certain nombre de couples clair/chiffré, l'attaquant est capable de retrouver la sous-clé. À partir d'un certain nombre de tours, il n'est plus possible de prévoir la valeur de C' qu'avec une certaine probabilité p. Donc dans une fraction $1-p$ des cas, on obtient un résultat aléatoire sans intérêt à la place des valeurs possibles de la sous-clé. Il est donc nécessaire d'effectuer une opération de filtrage afin de rejeter les mauvaises paires.

Exercice 6 : l'objectif de cet exercice est d'implanter la fonction suivante :

```
void crypta_des(INT r, INT N)
```

où `r` désigne le nombre de tours et `N` le nombre de couples (clair,chiffré) utilisés. Cette fonction sera appelée depuis la fonction `main()`, après mise à la clé de la version modifiée du DES (par utilisation de la fonction `setkey()`).

1) La première étape consiste à générer aléatoirement des nombres compris entre 0 et $2^8 - 1$. En utilisant la fonction `rand()` fournie par la bibliothèque `math`, implanter (éventuellement sous forme de macro) la fonction `rand_num(n)`. Cette fonction sera utilisée pour `n`$= 255$.

2) En utilisant la fonction `rand_num(n)`, générer les couples clairs satisfaisant aux caractéristiques différentielles suivantes :

- Dans le cas où `r`$= 3$, on va utiliser `N`$= 8$ couples clairs $(L_0^1 R_0^1, L_0^2 R_0^2)$ satisfaisant : $R_0^1 \oplus R_0^2 = 0$. La caractéristique différentielle sur 1 tour correspondante est de probabilité $p = 1$.

- Dans le cas où `r`$= 6$, on va utiliser `N`$= 600$ couples clairs $(L_0^1 R_0^1, L_0^2 R_0^2)$ satisfaisant : $L_0^1 \oplus L_0^2 = 0x40080000$ et $R_0^1 \oplus R_0^2 = 0x04000000$, puis $L_0^1 \oplus L_0^2 = 0x00200008$ et $R_0^1 \oplus R_0^2 = 0x00000400$. Les deux caractéristiques différentielles sur 3 tours correspondantes sont de probabilité $p = \frac{1}{4}.1.\frac{1}{4} = \frac{1}{16}$.

3) Pour chacun de ces couples clairs, calculer le couple de chiffrés $(L_r^1 R_r^1, L_r^2 R_r^2)$ correspondant.

4) L'étape suivante consiste à calculer les effectifs de la table `J[8][64]` par application itérée (`N` fois) de la fonction suivante :

```
do_job_des(INT r, BYTE *out11, BYTE *out12, BYTE *in11, BYTE *in12)
```

Cette fonction prend en argument le nombre de tours `r` et les clairs/chiffrés correspondants aux caractéristiques différentielles.

Dans le cas où $r = 6$, on filtrera les sous-clés K_6 candidates en utilisant les caractéristiques différentielles.

5) La dernière partie de la fonction `crypta_des` consiste pour chaque `i`$= 0, \ldots, 7$, à calculer l'indice `JM[i]`$=$`j` correspondant au plus grand des `J[i][j]`, `j`$= 0, \ldots, 63$. `JM[i]` nous fournis à chaque fois 6 bits de la sous-clé K_r.

6) Il ne nous reste plus ensuite qu'à retrouver (par recherche exhaustive) la clé K à partir de la donnée de la clé de tour K_r pour un r donné, en invoquant la fonction `Kr2K()`.

2.2 Cryptanalyse linéaire

La cryptanalyse linéaire est une attaque à clair connu. Cette attaque exploite le fait suivant : la table $NS(S)$ d'une S-Box présente des valeurs trop élevées. L'élément le plus grand de la table $|NS(S) - 2^m|$ est égal à $\frac{1}{2}L(S)$ où $L(S)$ est la linéarité de la S-Box.

Le détecteur exploite la relation linéaire correspondante entre l'entrée et la sortie de la S-Box pour distinguer l'algorithme de chiffrement d'une permutation aléatoire.

Références