Exposé de
Sécurité informatique

Sujet : Cryptanalyse Linière

Groupe : 1

Membre :

- Benkedadra Mohamed
- Benkorreche Mohamed El Amine

Cryptanalyse :

La cryptanalyse, est une technique dans le domaine de cryptographie, qui consiste de casser un algorithme. Ça veut dire d'essayer de trouver un message clair (Plain-Text), d'après un message chiffré (Cipher-Text) sans avoir la clé utilisée pour le chiffrement, Cette procédure d'essayé de déchiffrer un message chiffré en message clair sans avoir la clé de chiffrement est appelé un attaque. **Il y existe quatre type d'attaque dans la cryptanalyse :**

- **Attaque sur un texte chiffré seul:** le cryptanalyste possède des exemplaires de message chiffré et il essaye de deviner les messages en clairs.
- **Attaque à texte clair connu:** le cryptanalyste possède des exemplaires de couple de message clair/chiffré et il essaye de deviner la clé, pour qu'il puisse l'utilisé pour déchiffrer de futurs messages chiffré par le même algorithme
- **Attaque à texte clair choisi:** le cryptanalyste choisi des messages en clairs aléatoirement, dans le but de trouver les messages chiffrés et bien comprendre l'algorithme utilisé et ses failles.
- **Attaque à texte chiffré choisi:** le cryptanalyste essai de trouver des combinaisons possibles des messages en clairs pour des messages chiffrés, puis il les utilise pour trouver la clé de chiffrement.

Chiffrement par Bloc :

Avant de comprendre la cryptanalyse linière, il faut comprendre le chiffrement par bloc, parce que généralement, on utilise la cryptanalyse linière sur les algorithmes de chiffrement par bloc. ces algorithmes traitent les données (Texte Claire/Chiffré) en forme de bloc, qui sont généralement de taille fixe. Il combine une **clé K** avec un **message en clair P** pour avoir un **message chiffré C** (chiffrement)**,** ou on peut faire l'inverse pour avoir le message clair (déchiffrement)

Extension de clé :

il faut qu'on assure que la longueur de la clé a une relation avec la longueur du bloc du message, avant de les combiner. Dans certain cas la clé est d'une longueur inférieur de la longueur du bloc, dans ce cas on a besoin d'un algorithme d'extension de clé. Il est préférable que l'algorithme d'extension ne soit pas renversable, parce qu'on a pas besoin de le renverser

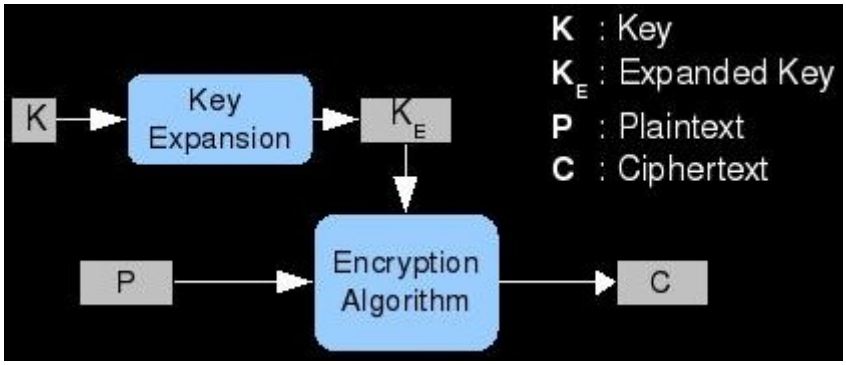


Figure 1

au futur. un exemple d'un algorithme d'extension non sécurisé est de répéter la clé jusqu'a qu'elle atteint la longueur du bloc, Cette procédure nous donne une **clé étendue K^E,** qu'on va utiliser dans l'algorithme de chiffrement .(Figure 1)

Les Réseaux de Substitution-Permutation (SPN) :

Un réseau substitution-permutation (Substitution-Permutation Network) est une des méthodes les plus simples pour approcher le chiffrement par bloc. Ils sont des algorithmes qui se consiste de différentes fonctions groupées dans des ensembles, qu'on appelle **Fonction de Round.** Ses rounds sont enchaînés ensemble pour former l'algorithme. Généralement dans ce cas, l'algorithme d'extension crée des clés pour chaque round (K^{E1}, K^{E2}…etc) depuis la clé originale K. (Figure 2)

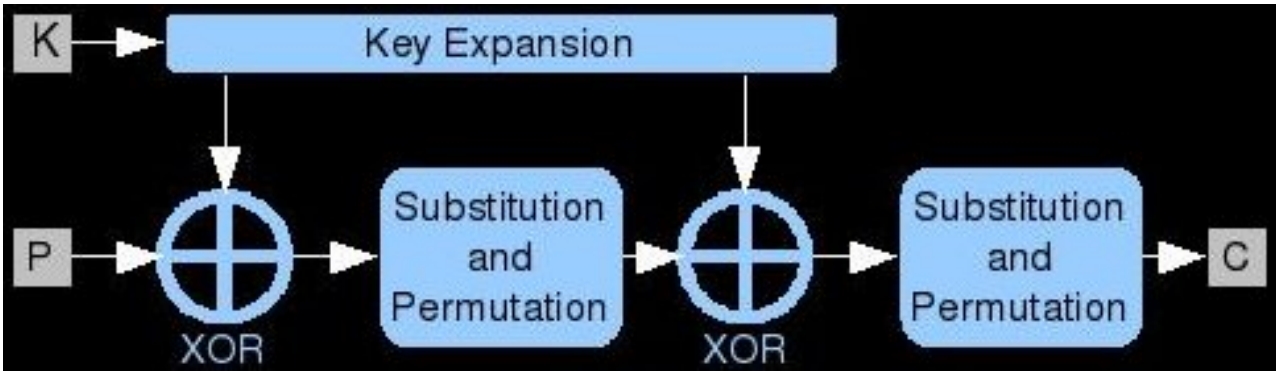


Figure 2

donc on passe un bloc en clair dans l'algorithme, le bloc va passer par un rond, ou il va êtres manipulé, ce processus est répété jusqu'à qu'on a passé sur tous les ronds, et le résultât sera un message chiffré.

Chaque round est définie par :

- **K^R** Clé du round (K^{ER} en cas d'extension)
- **W^{R-1}** État Initial (bloc d'entré du round)
- **W^R** État Finale (bloc de sortie du round)
- **G** Fonction du Rond (G^{-1} lors du déchiffrement)

Ou :

$$W^R = G(W^{R-1}, K^R)$$

Mélange de clés :

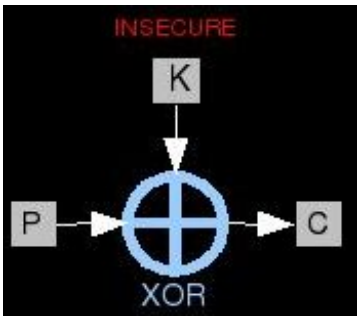


Figure 3

lors d'un round on trouve souvent un mixage de clé, ou en fait généralement un XOR entre la clé du round et le bloc d'entré C = P⊕K, le problème avec cette opération c'est qu'elle est linière, donc l'attaquant peut faire un P = C⊕K s'il connaît la clé, ou il peut casser l'algorithme est trouvé la clé par un K = C⊕P. Pour cela qu'on introduit une Permutation a l'aide d'un P-BOX, ou une Substitution a l'aide d'un S-BOX.

Boites de Permutation :

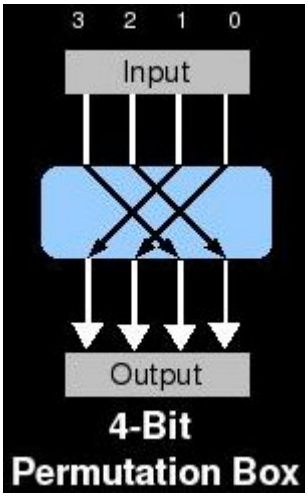


Figure 4

cet élément du round introduit **la Diffusion** a l'algorithme au but de diffuser l'information (le bloc d'entré), pour protégé les données contre les analyseurs des fréquences, qui essaye de trouver des répétitions dans le bloc pour les utilisés pour déchiffrer le message sans clé. Par exemple dans Figure 4 :

3210 devient 1032

Boites de Substitution :

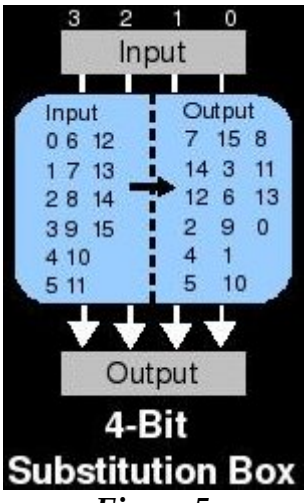


Figure 5

cet élément du round introduit **la Confusion** a l'algorithme, au but d'obscuré la relation entre le message clair et chiffré. Cette méthode ajoute de la sécurité contre la cryptanalyse linière et la cryptanalyse différentielle. Elle seul n'ajoute pas beaucoup de sécurité mais si on combine des S-BOX et des P-BOX avec des Mélanges de clés (XOR) sur plusieurs Rounds, on peut garantir la sécurité. Figure 5 représente le S-BOX suivant:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] => [7, 14, 12, 2, 4, 5, 15, 3, 6, 9, 1, 10, 8, 11, 13, 0]

Linéarité :

Maintenant on a une idée sur les chiffrements par bloc, et on comprend une des méthodes simples (SPN) utilisé dans ce genre d'algorithme, mais avant d'apprendre comment casser ces algorithmes a l'aide de la cryptanalyse linière, il faut comprendre le concept de linéarité. Alors, le concept suivant est un peu dure a expliqué, mais essayé de

comprendre que quelque chose de linière, est quelque chose prédictible. par exemple, le graphe d'une fonction linière ne change jamais de direction. Donc, il est prédictible, on sait ou il va. Mais une fonction non-linéaire change de direction. Donc, elle est non-prédictible. Ce concept est très important, parce que la cryptanalyse linière a le but de trouver de la linéarité dans les composants non-linéaires de l'algorithme, sinon on peut pas l'utilisé.

Cryptanalyse Linière :

la Cryptanalyse Linière est une technique de cryptanalyse qui se base sur les expressions linéaires probabilistes. Elle était inventée par **Mitsuru Matsui,** au but de casser l'algorithme de chiffrement DES. Il l'annonçait lors du «EUROCRYPT 93» au 1993, et elle est plus efficace que la Cryptanalyse Différentielle. La Cryptanalyse Linière consiste d'essayé de trouver une approximation linière pour chaque partie non linière de l'algorithme. Puis, d'essayé d'augmenter la précision de cette approximation au but de trouver la clé. Il faut noter qu'il n y a pas une recette précise pour faire une cryptanalyse linière pour n'importe quel algorithme. Enfin, chaque algorithme est différent, donc il faut bien analyser l'algorithme puis appliqué les principes de la Cryptanalyse Linière. En lissant la définition, Cette technique a l'air d'être dur, mais on va prendre un exemple simple et on va essayer de le casser ensemble pour bien comprendre.

Analyse de l'algorithme :

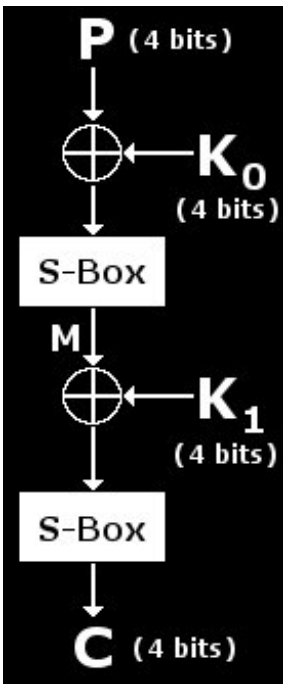


Figure 6

Prenant l'algorithme (SPN) dans Figure 6. Disant qu'on a 16 messages clair de 4bit et leur 16 messages chiffré on a aussi l'algorithme, et on veut casser cet algorithme (trouvé la clé de chiffrement/déchiffrement), donc on cherche **K** ou $K = K_0K_1$ donc on cherche K_0 et K_1. disant que :

P message claire
C message chiffré
M message semi-chiffré
K_0 K_1 les clés des R_0 R_1
S fonction du S-BOX

Voyant comment C est calculés :

$M = S(K_0 \oplus P)$
$C = S(K_1 \oplus M)$

On a 2 problèmes : le 1er c'est qu'on n'a pas M. Si on avait pas un S-BOX et on avait M, on pouvait faire $K_0 = M \oplus P$ et $K_1 = M \oplus C$.
Le 2e c'est qu'on a un S-BOX. Comme on a déjà dit, le S-BOX introduit la confusion, donc il introduit la non-linéarité a l'algorithme. Alors on peut pas faire un $K = C \oplus P$ sans faire une approximation linière pour le S-BOX, autrement dit il faut trouver de la linéarité dans le S-BOX. dans notre cas le S-BOX et le seul composant qui introduit la non-linéarité. Si on avait d'autre composant, il faut faire une approximation pour chaqu'un de ces composants.
D'après les informations, on peut déduire le suivant :
- K = 8bit (K_0 4bit et K_1 4bit), donc on a 256-**K-**possible
- On a 16 blocs possible d'input (claire) et 16 blocs possible d'output (chiffré)

Approximation Linière :

on oublie l'algorithme pour un moment et on concentre sur le S-BOX. Disant qu'on a le S-BOX dans Figure 7, on va essayer de le faire une approximation linière. Pour cela, on va utiliser le concept de parité, la parité d'un nombre est toujours égale a 1 ou 0. Pour savoir la parité d'un nombre, on le transforme en binaire puis on compte le nombre des 1, si se nombre et impair le bit de parité égale a 1 sinon (pair) le bit de parité égale a 0. Il y a aussi une 2e méthode, qu'on va utiliser. On peut faire un XOR entre 2bit pour avoir la parité du nombre. par exemple :

Substitution Box for Toy Cipher	
0 -> 9	8-> 13
1 -> 11	9 -> 7
2 -> 12	10 -> 3
3 -> 4	11 -> 8
4 -> 10	12 -> 15
5 -> 1	13 -> 14
6 -> 2	14 -> 0
7 -> 6	15 -> 5

Figure 7

Parité(01) = $0 \oplus 1 = 1$

Pour les nombres ou le nombre de bit et supérieur a 2, on fait la même chose qu'avant pour chaque couple de bit, par exemple :

Parité (10110) = $((0 \oplus 1) \oplus (1 \oplus 1)) \oplus 0 = 1$

Parité (100) = $(1 \oplus 0) \oplus 0 = 1$

Parité (1001) = $(1 \oplus 0) \oplus (0 \oplus 1) = 0$

il y a aussi une autre concept qu'on va utiliser, le masquage bit. un masque est un nombre en bit qui as la même taille que notre nombre, son but est d'altéré un nombre, et nous donner un nombre masqué. un nombre de n bit a 2^n masque possible, par exemple pour la valeur 1010 on peut avoir 16 masque différent, parce qu'elle consiste de 4bit donc 2^4**.** Ses masques possibles sont [0000,0001,0010,….,1111]. on peut utiliser des opérations

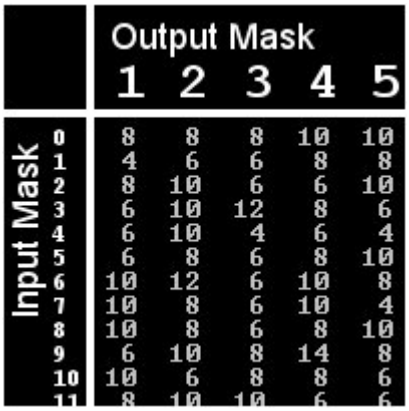
comme AND, OR et XOR pour masquer la valeur par un des masques possible. Dans notre approximation linière, on va utiliser AND, donc :

V-Masqué = V AND M.

par exemple :

valeur = 10011
masque = 11100
valeur masquée = 10000

Maintenant qu'on a bien compris la parité et le masquage bit, on va voir comment les utilisés pour faire une approximation linière.



Input Mask	Output Mask 1	2	3	4	5
0	8	8	8	10	10
1	4	6	6	8	8
2	8	10	6	6	10
3	6	10	12	8	6
4	6	10	4	6	4
5	6	8	6	8	10
6	10	12	6	10	8
7	10	8	6	10	4
8	10	8	6	8	10
9	6	10	8	14	8
10	10	6	8	8	6
11	8	10	10	6	6

Figure 8

On va dessiner un tableau (Figure 8 tableau incomplet) ou les colonnes sont les masques de sortie, et les lignes sont les masques d'entrés. Notre S-BOX a 16 combinaisons de valeur entré/sortie, donc pour chaque valeur d'entrée on a 16 masque possible (16 lignes) et pour chaque valeur de sortie on a aussi 16 masque possible (16 colonnes). On va masquer une valeur d'entré possible avec un masque d'entré possible et on va calculer la parité P^1, puis on va masquer une valeur de sortie possible avec un masque de sortie possible et on va calculer la parité P^2, puis on vérifiait si les parités sont égales ($P^1 = P^2$). Si oui, on augmente la case [masque d'entré][masque de sortie] par 1. On fait cette opération pour chaque combinaison (masque d'entré, masque de sortie, combinaison de S-Box), chaque entrée dans le tableau et le numéro de fois que l'approximation linière formé par un couple (masque d'entré, masque de sortie) a été vrai. Si le S-BOX été totalement non-linéaire (sécurité maximal), toutes les entrées seront égales a 8, et la cryptanalyse linéaire ne sera pas efficace.

<u>Meilleure approximation :</u>

donc pour le moment, on n'a pas touché nos données (les messages clairs et chiffré qu'on a). on a juste trouvé une approximation linière pour notre S-BOX. Maintenant, on essaye de trouver la meilleure approximation linière dans le tableau. On trouve la case avec le plus grand nombre. Si on trouve une case avec une valeur de 16, la chance qu'on soit vrai pour le couple (masque d'entré, masque de sortie) de cette case est de 100%. Dans le cas de notre S-BOX c'est la case [11][11] avec une valeur de 14. Ça veut dire que pour le masque d'entré 11 avec le masque de sortie 11, on a trouvé $P^1=P^2$ pour 14 de 16 des combinaisons du S-BOX alors une

chance de ~87% qu'on soit vrai. Les cases [][] et [][] peuvent aussi être choisi comme meilleure approximation parce qu'ils ont des approximations de 14.
Si notre algorithme utilise XOR avant d'utilisé le S-BOX comme dans notre cas, il est possible que l'approximation est inversée. Donc, peut-être que 14 ne veut pas dire ~87% mais veut dire ~12%. Alors, on ne trouve pas juste les cases avec des valeurs supérieures ou égales a 14, mais on trouve aussi les cases avec des valeurs inférieures ou égales a 2. Parce que 2 est l'inverse de 14 sur l'intervalle [0-16].

Casser l'algorithme :

Maintenant, on peut utiliser le couple (masque d'entré, masque de sortie) trouvé comme meilleur approximation, pour trouver K. On commence par calculer le M de chaque P pour chaque K_0 possible.

$$M = S (K_0 \oplus P)$$

On va avoir 16M possible pour chaque P (parce qu'on a 16 K_0 possible). Alors, pour chaque K_0 possible on trouve 16M pour les 16P qu'on a. on crée un tableau a une dimension, chaque case représente un K_0 possible. Après, on teste la série des M pour chaque K_0 avec la série des P, en masquent chaque P par le masque d'entré de la meilleure approximation, et chaque M par le masque de sortie de la meilleure approximation. En même temps, on calcule les parités de chaque valeur masquée et on teste si les parités d'entrés et de sorties sont égales, si oui on augmente la case du K_0 utilisé pour calculer ce M par 1. Finalement, on cherche dans le tableau a une dimension, les cases qui contiennent la plus grande valeur, ou la plus petite valeur (parce que en utilise XOR) et on prend les K_0 associé a ces cases. Maintenant, on calcule K_1 pour chaqu'un des M de ces K_0 , on fait:

$$K1 = M \oplus S(C)$$

On fait ça pour chaque couple (K_0, K_1). en même temps on teste le couple trouver pour tous les P qu'on a. Si on trouve que pour tous les P, tous les C sont correctes pour un couple (K_0, K_1) donné, alors on a trouvé la clé (K_0, K_1), et par conséquence on a trouvé la clé K.

Le Code (C) :

```
#include <stdio.h>

int sBox[16] = {9, 11, 12, 4, 10, 1, 2, 6, 13, 7, 3, 8, 15, 14, 0, 5};
int revSbox[16] = {14, 5, 6, 10, 3, 15, 7, 9, 11, 0, 4, 1, 2, 8, 13, 12};

int approxTable[16][16];

int knownP[500];
int knownC[500];

int numKnown = 0;

int applyMask(int value, int mask) {
   int interValue = value & mask;
   int total = 0;

   while(interValue > 0) {
      int temp = interValue % 2;
      interValue /= 2;

      if (temp == 1)
         total = total ^ 1;
   }
   return total;
}

void findApprox() {
   int c, d, e;
   for(c = 1; c < 16; c++)                    //Masque de Sortie
      for(d = 1; d < 16; d++)                  //Masque d'éntrée
         for(e = 0; e < 16; e++)               //Combination SBOX
            if (applyMask(e, d) == applyMask(sBox[e], c))
               approxTable[d][c]++;
}

void showApprox() {
   printf("Meilleure approximation Linière: \n");
   int c, d, e;
   for(c = 1; c < 16; c++)
      for(d = 1; d < 16; d++)
         if (approxTable[c][d] == 14)
            printf("  %i : %i -> %i\n", approxTable[c][d], c, d);

   printf("\n");
}

int roundFunc(int input, int subkey) {
   return sBox[input ^ subkey];
}
```

```
void fillKnowns() {
   int subKey1 = rand() % 16;
   int subKey2 = rand() % 16;
   int total = 0;
   int c;

   printf("Génération Données:  K1 = %i, K2 = %i\n", subKey1, subKey2);
   for(c = 0; c < numKnown; c++) {
      knownP[c] = rand() % 16;
      knownC[c] = roundFunc(roundFunc(knownP[c], subKey1), subKey2);
   }
   printf("Génération Données:  %i Pair ont été générer\n\n", numKnown);
}

void testKeys(int k1, int k2) {
   int c;
   for(c = 0; c < numKnown; c++)
      if (roundFunc(roundFunc(knownP[c], k1), k2) != knownC[c])
         break;

   printf("* ");
}

int main() {
   printf("Cryptanalyse Linière\n");
   printf("-----------------------------------------------\n\n");
   srand(time(NULL));

   findApprox();
   showApprox();

   int inputApprox = 11;
   int outputApprox = 11;

   numKnown = 16;
   fillKnowns();

   int keyScore[16];
   printf("Attaque Linière:  En Utilisant l'Approximation = %i -> %i\n", inputApprox, outputApprox);
   int c, d;
   for(c = 0; c < 16; c++) {
      keyScore[c] = 0;
      for(d = 0; d < numKnown; d++) {
         int midRound = roundFunc(knownP[d], c);
         if ((applyMask(midRound, inputApprox) == applyMask(knownC[d], outputApprox)))
            keyScore[c]++;
         else
            keyScore[c]--;
      }
   }
```

```
    int maxScore = 0;
    for(c = 0; c < 16; c++) {
        int score = keyScore[c] * keyScore[c];
        if (score > maxScore) maxScore = score;
    }

    int goodKeys[16];
    for(d = 0; d < 16; d++)
        goodKeys[d] = -1;

    d = 0;
    for(c = 0; c < 16; c++)
        if ((keyScore[c] * keyScore[c]) == maxScore) {
            goodKeys[d] = c;
            printf("Linear Attack:  Candidate for K1 = %i\n", goodKeys[d]);
            d++;
        }

    int guessK1, guessK2;
    for(d = 0; d < 16; d++) {
        if (goodKeys[d] != -1) {
                int k1test = roundFunc(knownP[0], goodKeys[d]) ^ revSbox[knownC[0]];
                int tested = 0;
                int e;
                int bad = 0;
                for(e = 0;e < numKnown; e++) {
                    int testOut = roundFunc(roundFunc(knownP[e], goodKeys[d]), k1test);
                    if (testOut != knownC[e])
                        bad = 1;
                }
                if (bad == 0) {
                    printf("Linear Attack:  Found Keys! K1 = %i, K2 = %i\n", goodKeys[d], k1test);
                    guessK1 = goodKeys[d];
                    guessK2 = k1test;
                }
        }
    }
}
```