

Cryptanalyse — M1MA9W06  
Responsable : G. Castagnos

## Devoir surveillé — 8 novembre 2011

*Durée 1h30*

*accès aux fonctions programmées en TP autorisé, autres documents non autorisés*

**1** Complexité linéaire d'une somme (exercice théorique)

Dans tout l'exercice, on considère deux suites à récurrence linéaire  $s$  et  $s'$  de polynômes de rétroaction  $h(X)$  et  $h'(X)$  primitifs sur  $F_2$ , de degré  $L$  et  $L'$  avec  $L \neq L'$ .

- (a) Quelles sont les périodes et les complexités linéaires de  $s$  et  $s'$ ? Quelle est la complexité linéaire de la suite somme  $s + s'$  définie par  $(s + s')_i = s_i + s'_i, \forall i \geq 0$ ?

Comme les polynômes de rétroaction sont primitifs, les périodes sont respectivement  $2^L - 1$  et  $2^{L'} - 1$  et les complexités linéaires  $L$  et  $L'$ . Pour la suite somme, comme de plus  $L \neq L'$ , d'après le théorème du cours sur la complexité linéaire des LFSR combinés, la complexité linéaire est  $L + L'$ .

- (b) Soit  $S(X)$  la série génératrice associée à la suite  $s$ , définie par  $S(X) = \sum_{n \geq 0} s_n X^n$ . On rappelle que la suite  $S$  est produite par un LFSR de polynôme de rétroaction  $h(X)$  si et seulement si on a le développement en série formelle  $S(X) = g(X)/h(X)$ , où  $g(X)$  est un polynôme de  $F_2[X]$  tel que  $\deg(g) < \deg(h)$ . Démontrer le résultat sur la complexité linéaire de la suite  $s + s'$ .

Soit  $S'(X)$  associée à  $s'$  et  $g'(X)$  tel que  $\deg(g') < \deg(h')$ . On a

$$(S + S')(X) = \frac{g(X)}{h(X)} + \frac{g'(X)}{h'(X)} = \frac{g(X)h'(X) + g'(X)h(X)}{h(X)h'(X)}.$$

Le degré du numérateur de la dernière fraction est strictement inférieur à celui de son dénominateur : le dénominateur a pour degré  $L + L'$ , celui du numérateur est inférieur ou égal à  $\max(\deg(g) + \deg(h'), \deg(g') + \deg(h)) < L + L'$ .

De plus, cette dernière fraction est réduite. Pour voir cela, supposons que  $f$  est un polynôme divisant à la fois le numérateur et le dénominateur. En particulier,  $f$  divise  $hh'$  avec  $h$  et  $h'$  distincts et irréductibles. Donc  $f$  vaut soit  $1, h, h'$  ou  $hh'$ . Le cas  $f = hh'$  est impossible car  $f$  ne peut pas diviser le numérateur de degré strictement inférieur. Supposons  $f = h$ , alors comme  $f$  divise le numérateur,  $f$  divise  $g'h'$  et comme  $h$  est premier à  $h'$ ,  $f$  divise  $g$  : contradiction car  $g$  est de degré strictement inférieur. Le cas  $f = h'$  est symétrique donc  $f = 1$  et la fraction est réduite.

Au final,  $h(X)h'(X)$  est le polynôme de rétroaction minimal de  $s + s'$  qui a pour complexité linéaire le degré de ce polynôme :  $L + L'$ .

## 2 Alternating Step Generator

On considère une variante de l'*Alternating Step Generator*, un chiffrement à flot synchrone additif proposé par Günther en 1987. Ce système utilise trois LFSR :  $LFSR_A$ ,  $LFSR_B$  et  $LFSR_C$ , de longueurs respectives  $L_A$ ,  $L_B$  et  $L_C$ . Les état initiaux des trois LFSR notés  $K_A, K_B, K_C$  constituent la clef secrète. La rétroaction de ces trois LFSR est publique et est choisie telle que les trois LFSR engendrent des m-suites.

Le  $LFSR_C$  est mis à jour à chaque instant  $t$  et sa sortie est utilisée pour contrôler la mise à jour des deux autres LFSR. La sortie de l'*Alternating Step Generator* est la somme de la sortie des  $LFSR_A$  et  $LFSR_B$ .

Plus précisément, on note  $a^{(t)}, b^{(t)}$  et  $c^{(t)}$  les sorties des  $LFSR_A$ ,  $LFSR_B$  et  $LFSR_C$  au temps  $t$ . On pose  $a^{(-1)} = b^{(-1)} = 0$ . Le générateur fonctionne comme suit : à chaque instant  $t \geq 0$  le  $LFSR_C$  est mis à jour,

- Si le bit de sortie du  $LFSR_C$ ,  $c^{(t)}$  vaut 0 alors le  $LFSR_A$  est mis à jour normalement, et sort  $a^{(t)}$ . Le  $LFSR_B$  n'est pas mis à jour. On pose  $b^{(t)} = b^{(t-1)}$ .
- Sinon quand  $c^{(t)} = 1$  alors c'est le  $LFSR_B$  qui est inchangé, on pose  $a^{(t)} = a^{(t-1)}$ . Le  $LFSR_B$  est mis à jour et sort un bit  $b^{(t)}$ .

Le bit de sortie de l'*Alternating Step Generator* au temps  $t$  est  $a^{(t)} + b^{(t)} \bmod 2$  ;

Dans tout l'exercice, on fixe les choix suivants

- $LFSR_A$  de longueur  $L_A = 8$  et polynôme de rétroaction  $P_A = x^8 + x^4 + x^3 + x^2 + 1$ ;
- $LFSR_B$  de longueur  $L_B = 9$  et polynôme de rétroaction  $P_B = x^9 + x^4 + 1$ ;
- $LFSR_C$  de longueur  $L_C = 7$  et polynôme de rétroaction  $P_C = x^7 + x + 1$ .

où les trois polynômes sont primitifs.

- (a) Donner le code d'une fonction qui simule une étape d'un LFSR : elle doit prendre en entrée un polynôme de rétroaction  $P$  de degré  $d$ , un état au temps  $t$  (supposé de longueur  $d$ ) et ressortir l'état au temps  $t+1$  et le bit de sortie. Donner les 10 premiers bits produits par le  $LFSR_A$  initialisé par  $K_A = [1, 1, 0, 0, 1, 1, 1, 1]$ .

```

PR.<x> = PolynomialRing(GF(2))
PA = x^8 + x^4 + x^3 + x^2 + 1

def LFSR_step(P, state) :
    L = len(state)
    out = state[0]
    state = state[1:] + [sum(P[j+1]*state[L-1-j] for j in range(L))]
    return out, state

KA = Sequence([GF(2)(1), 1, 0, 0, 1, 1, 1, 1])
state = copy(KA)
for i in range(10) :
    out, state = LFSR_step(PA, state)
    print out
retourne 1 0 0 1 1 1 0 0.

```

- (b) Donner le code d'une fonction prenant en entrée les 3 états initiaux  $K_A, K_B$  et  $K_C$  des  $LFSR_A, LFSR_B, LFSR_C$  et un entier  $N$  et rentrant les  $N$  premiers bits de sortie de

l'*Alternating Step Generator*. Donner les 5 premiers bits produits par le générateur initialisé par les clefs

$$\begin{aligned} K_A &= [1, 1, 0, 0, 1, 1, 1, 1] \\ K_B &= [1, 1, 0, 0, 1, 1, 1, 0] \\ K_C &= [1, 1, 0, 0, 1, 1, 1] \end{aligned}$$

```

KA = Sequence([GF(2)(1), 1, 0, 0, 1, 1, 1])
KB = Sequence([GF(2)(1), 1, 0, 0, 1, 1, 1, 0])
KC = Sequence([GF(2)(1), 1, 0, 0, 1, 1, 1])
PB = x^9 + x^4 + 1
PC = x^7 + x + 1
LA=8; LB=9; LC=7
def AS(KA,KB,KC, N) :
    z = []
    a = GF(2)(0); b = a
    stateA = copy(KA)
    stateB = copy(KB)
    stateC = copy(KC)
    for i in range(N) :
        c, stateC = LFSR_step(PC, stateC)
        if c == 0 :
            a, stateA = LFSR_step(PA, stateA)
        else :
            b, stateB = LFSR_step(PB, stateB)
        z.append(a+b)
    return z
print AS(KA, KB, KC, 5)
donne [1, 1, 0, 0, 1].

```

- (c) Dans la suite de l'exercice, on se propose de faire une cryptanalyse de l'*Alternating Step Generator* et on suppose que l'on a récupéré une liste S de N bits produits par le générateur. Dans le cadre d'une utilisation de ce générateur comme chiffrement symétrique, dans quel scénario d'attaque peut-on récupérer S ?

En chiffrement symétrique, la suite S de N bits est additionnée bit à bit avec le message clair lors du chiffrement. Dans le cadre d'une attaque à clair connu, l'attaquant a accès aux messages clairs et aux chiffrés correspondants. Avec N bits de clairs et de chiffrés, en faisant la somme bit à bit il récupère N bits de la sortie du générateur S.

- (d) Dans cette question, on suppose que l'on connaît la clef  $K_C$ . Montrer en utilisant la matrice de rétroaction  $M_A$  du LFSR<sub>A</sub> que l'on peut à chaque instant  $t \geq 0$  écrire le bit  $a^{(t)}$  comme une équation linéaire d'inconnues les bits de  $K_A$  notés  $K_{A_0}, K_{A_1}, \dots, K_{A_{L_A-1}}$ . On note cette équation

$$a^{(t)} = u_0^{(t)} K_{A_0} + u_1^{(t)} K_{A_1} + \dots + u_{L_A-1}^{(t)} K_{A_{L_A-1}}.$$

Préciser comme trouver la liste des coefficients  $Eq_A^{(t)} := [u_0^{(t)}, u_1^{(t)}, \dots, u_{L_A-1}^{(t)}]$ .

Pour rappel, la matrice de rétroaction d'un LFSR de longueur L et de polynôme de rétro-

action  $P(X) = 1 + c_1X + c_2X^2 + \dots + c_LX^L$  est la matrice  $L \times L$  à coefficients dans  $F_2$  :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \\ c_L & c_{L-1} & \dots & c_3 & c_2 & c_1 \end{pmatrix}$$

Si on connaît la clef  $K_C$  alors on connaît les bits  $c^{(t)}$  et on sait quand le LFSR<sub>A</sub> est itéré. Si  $R_i$  est l'état du LFSR<sub>A</sub> au bout de  $i$  itérations, en considérant les  $R_i$  comme des vecteurs colonnes, on a  $R_{i+1} = M_A R_i$  et  $R_i = M_A^i K_A$ . En particulier, le bit  $a$  sorti à l'itération  $i+1$  étant le bit de poids faible du registre  $R_i$  au temps  $i$ , on a  $a$  qui peut s'exprimer comme combinaison linéaire des bits de  $K_A$ . Cette combinaison étant donné par la première ligne de la matrice  $M_A^i$ . Au temps 0, comme  $a^{(0)}$  est nul on a  $EQ_A^{(0)} := [0, 0, \dots, 0]$ . Ensuite suivant les valeurs de  $c^{(t)}$ , l'équation au temps  $t$  est soit la même qu'au temps précédent, soit on la trouve en calculant les puissances successives de  $A^i$ .

- (e) Déduire de la question précédente le code d'une fonction Sage qui prend en entrée  $K_C$  et un entier  $N$  et qui renvoie la matrice de taille  $N \times L_A$  dont la ligne  $t$  contient les coefficients  $[u_0^{(t)}, u_1^{(t)}, \dots, u_{L_A-1}^{(t)}]$ . Avec les clefs données en (b) et  $N = 5$  que donne cette matrice multipliée par le vecteur constitué de la clef  $K_A$  ?

```

MA = Matrix(GF(2),LA,LA)
for i in range(LA-1) :
    MA[i,i+1] = 1
tmp = PA.list()[1 :]
tmp.reverse()
MA[LA-1] = vector(tmp)

def equationsA(KC, N) :
    Eq = Matrix(GF(2),N,LA)
    SA = MA^0
    equation = vector(GF(2),LA)
    stateC = copy(KC)
    for i in range(N) :
        c, stateC = LFSR_step(PC, stateC)
        if c == 0 :
            equation = SA[0]
            SA = SA*MA
        Eq[i] = equation
    return Eq
print equationsA(KC, 5)*vector(KA)
cette dernière ligne donne (0,0,1,1,1) soit les 5 premières valeurs  $a^{(0)}, a^{(1)}, \dots, a^{(4)}$ .

```

- (f) Indiquer comment modifier la fonction de la question précédente pour prendre en entrée  $K_C$  et un entier  $N$  mais renvoyer la matrice de taille  $N \times (L_A + L_B)$  dont la ligne  $t$  contient les coefficients

$$[u_0^{(t)}, u_1^{(t)}, \dots, u_{L_A-1}^{(t)}, v_0^{(t)}, v_1^{(t)}, \dots, v_{L_B-1}^{(t)}],$$

où  $[v_0^{(t)}, v_1^{(t)}, \dots, v_{L_B-1}^{(t)}]$  sont les coefficients de l'équation linéaire exprimant  $b^{(t)}$  en fonction des bits de  $K_B$ .

On procède de même avec le LFSR<sub>B</sub> pour obtenir la fonction

```
def equationsAB(KC, N) :
    Eq = Matrix(GF(2), N, LB+LA)
    SA = MA^0
    SB = MB^0
    equationA = vector(GF(2), LA)
    equationB = vector(GF(2), LB)
    stateC = copy(KC)
    for i in range(N) :
        c, stateC = LFSR_step(PC, stateC)
        if c == 0 :
            equationA = SA[0]
            SA = SA*MA
        else :
            equationB = SB[0]
            SB = SB*MB
        Eq[i, :LA] = equationA
        Eq[i, LA :] = equationB
    return Eq
```

- (g) En déduire une attaque pour retrouver les trois clefs K<sub>A</sub>, K<sub>B</sub>, K<sub>C</sub> à partir des N premiers bits de sortie du générateur. Quelle est la complexité de cette attaque? Que doit on supposer?

On fait une recherche exhaustive sur la clef K<sub>C</sub>. Pour chaque valeur, la fonction précédente nous donne une matrice M représentant N équations linéaires. Si S désigne le vecteur colonne correspondant à la sortie du générateur, pour la bonne valeur de K<sub>C</sub>, en résolvant le système linéaire MX = S, on doit obtenir le vecteur colonne correspondant à la concaténation de K<sub>A</sub> et K<sub>B</sub>. Pour chaque valeur de K<sub>C</sub>, on cherche donc une solution au système. Si on a une solution alors on a un candidat pour K<sub>A</sub>, K<sub>B</sub>, K<sub>C</sub>, en supposant que l'on a assez d'équations indépendantes pour que le bon système est une solution (c'est à dire L<sub>A</sub> + L<sub>B</sub>). La complexité dominante est la recherche exhaustive sur la clef K<sub>C</sub> en 2<sup>L<sub>C</sub></sup>.

- (h) Récupérer une suite produite par le générateur en tapant dans Sage :

```
load http://www.math.u-bordeaux1.fr/~gcastagn/suite.sage
```

Quelles sont les initialisations K<sub>A</sub>, K<sub>B</sub>, K<sub>C</sub> qui ont produit cette suite? Préciser le code utilisé.

Indications : en Sage, pour trouver la solution d'un système linéaire MX = S où M est une matrice et S un vecteur colonne, on peut utiliser la commande M.solve\_right(S). Cette commande retourne une erreur s'il n'y a pas de solution. Pour utiliser cette commande dans une boucle et continuer en cas d'erreur, on peut se servir du code suivant :

```
try :
    sol = M.solve_right(S)
    :
except ValueError :
    continue
```

```

MB = Matrix(GF(2),LB,LB)
for i in range(LB-1) :
    MB[i,i+1] = 1
tmp = PB.list()[1 :]
tmp.reverse()
MB[LB-1] = vector(tmp)
load http://www.math.u-bordeaux1.fr/~gcastagn/suite.sage
def attack(S) :
    w = vector(S[:LA+LB+10])
    N = len(S)
    for tmp in VectorSpace(GF(2), LC) :
        stateC = list(tmp)
        Eq = equationsAB(stateC, LA+LB+10)
        try :
            sol = Eq.solve_right(w)
            KKA = list(sol[:LA])
            KKB = list(sol[LA:])
            KKC = list(tmp)
            if AS(KKA, KKB, KKC, N) == S :
                return KKA, KKB, KKC
        except ValueError :
            continue

print attack(S)
On trouve ([0,0,1,0,0,1,1,0],[1,0,0,0,1,1,0,0,0],[1,0,0,1,0,1,0]).
```