Projet de Cryptanalyse

Léo BARRÉ, Elliott BLOT

Novembre 2016

Introduction

L'algorithme de chiffrement **A5/2** a été créé en 1999 et fait partie du groupe des algorithmes A5 (A5/1, KASUMI, ...), utilisés dans le cadre des communications GSM, ici dans le but d'exporter le système hors de l'Europe, à la place de **A5/1**.

À peine une semaine après sa diffusion, les cryptanalystes **Goldberg**, **Wagner** et **Green** publièrent une étude mettant en évidence une faible sécurité - suffisamment pour pouvoir être cassé en temps réel avec seulement deux clairs choisis - bien moindre que celle de son prédécesseur.

Depuis le 1er Juillet 2006, l'algorithme a cessé d'être supportée par les téléphones mobiles, puis interdite à l'implémentation l'année suivante, pour des raisons... évidentes.

L'attaque que nous allons étudier ici est celle réalisée par **Barkan**, **Biham** et **Keller** en 2003.

Cette attaque est inspirée de faits réels.

Réponses

1

Nous avons choisi de coder les LFSRs et les A5/2 sous forme de classes d'objet :

LFSR : elle est construite à l'aide d'un polynôme P et l'Initial Value IV de son registre

```
class LFSR:
    def __init__(self,P,IV=None):
        # initializes the register at IV and saves polynom P
    def step(self):
        # updates the register and returns its last bit
```

(***Note*** *: le dernier point de la méthode* ***step*** *est secondaire, les A5/2 n'utilisant jamais le bit de sortie de leurs LFSRs*)

A5/2 : Pour cet objet , nous avons d'abord déclaré quelques variables globales telles que :

- les tailles des entrées :

```
Ksize  = 64
IVsize = 22
```

- les polynômes des LFSRs et la taille de leurs registres :

```
P = [None]*5
P[1] = x^19 + x^18 + x^17 + x^14 + 1  # 0,1,2,5
P[2] = x^22 + x^21 + 1                # 0,1
P[3] = x^23 + x^22 + x^21 + x^8  + 1  # 0,1,2,15
P[4] = x^17 + x^12 + 1                # 0,5

l = [None]*5
for i in range(1,5):
    l[i] = P[i].degree()
```

(***Note*** *: la plupart des tableaux stockant des données inhérentes aux LFSRs des A5/2 auront un premier indice pointant sur un "**None**". Il n'est présent que dans un souci esthétique, pour que les LFSRs du code soient bien numérotées entre 1 et 4*)

- les registres fixés à 1 à la fin de l'initialisation de l'objet :

```
Rinit = [None,3,5,4,6]
```

- ainsi que ceux utilisés par la fonction **Maj** (cette dernière est également codée, mais nous réservons le détail de son implémentation pour une question ultérieure) :

```
Rmaj = [ None,\
         [3,4,6 ],\
         [8,5,12],\
         [4,9,6 ],\
         [6,13,9]]
```

- et enfin, l'objet lui-même, codé sous la forme de la classe **O5**
(car A5 = Audi = O10, donc A5/2 = O10/2 = O5).
Elle est construite à l'aide d'une clé K et sa valeur d'initialisation IV

```
class O5:
    def __init__(self,IV,K):
        # initialisation phase of the A5/2
    def step(self):
        # updates the LFSRs and returns one bit of cipher
    def cipher(self,N,wait=99):
        # generates a cipher of size N
```

(***Note*** *: la possibilité de changer la valeur de **wait**, qui représente le nombre d'exécutions de **step** avant la génération de la chiffre suitante, est présente surtout pour du débugage*)

Pour le test proposé, nous avons donc au préalable stocké les données du test sous les noms **TEST1_K**, **TEST1_IV** et **TEST1_z**. Ce qui nous donne, utilisé sur le terminal sage :

```
sage: TEST1_z == O5( IV=TEST1_IV , K=TEST1_K ).cipher(228)
True
```

2

Dans la mesure où on connaît R_4, on peut tout à fait anticiper quelles LFSRs vont être mises à jour lors de celle de l'A5/2.

En effet, une fois calculé **Maj**($R_{4,6}$, $R_{4,13}$, $R_{4,9}$), on vérifie les éléments faisant partie de la majorité, et on met à jour les autres LFSRs en fonction.

Par exemple, avec $R_1 = (x_0, \ \dots \ , x_{18})$, on a :

- si $R_{4,6}$ n'est pas dans la majorité, on retourne $R_1 = (x_0, \ \dots \ , x_{18})$,
- si $R_{4,6}$ est dans la majorité, on retourne $R_1 = (x_1, \ \dots , x_{18}, \ x_0 + x_1 + x_2 + x_5)$

On fait de même avec tous les registres et leur polynôme respectif, et on obtient les registres après mise à jour de l'A5/2, par rapport à leur état précédent.

3

Au préalable, nous avons ajouté différentes variables globales pour les attaques implémentées comme :

- **BPR** lui-même, qui sera utilisé pour représenter les valeurs des registres à l'initialisation, ainsi que les valeurs de bases de K plus tard :

```
BPR = BooleanPolynomialRing(64,'x')
BPRv = BPR.gens()
```

- les indices des registres initialisés à 1 ainsi que ceux de la majoration dans **BPR** :

```
Rinitb = [None]*4
Rinitb[1] = Rinit[1]
Rinitb[2] = Rinit[2] + l[1]
Rinitb[3] = Rinit[3] + l[1] + l[2]

Rmajb = [None]*4
Rmajb[1] = [Rmaj[1][i]                for i in range(3) ]
Rmajb[2] = [Rmaj[2][i] + l[1]         for i in range(3) ]
Rmajb[3] = [Rmaj[3][i] + l[1] + l[2]  for i in range(3) ]
```

Nous avons également codé la fonction **Req2Req**, qui s'occupe de reproduire l'algorithme présenté dans la question précédente :

```
def Req2Req(R4, vals=list(BPR.gens())):
    # returns registers'equations after step according to before step
```

où **vals** représente les valeurs des registres actuels

Quelques exemples d'utilisation :

```
sage: req = Req2Req([0 for i in range(l[4])]); req
[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15,
 x16, x17, x18, x0 + x1 + x2 + x5,
 x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30, x31, x32,
 x33, x34, x35, x36, x37, x38, x39, x40, x19 + x20,
 x42, x43, x44, x45, x46, x47, x48, x49, x50, x51, x52, x53, x54,
 x55, x56, x57, x58, x59, x60, x61, x62, x63, x41 + x42 + x43 + x56]
sage: Req2Req([0 for i in range(l[4])],req)
[x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16,
 x17, x18, x0 + x1 + x2 + x5, x1 + x2 + x3 + x6,
 x21, x22, x23, x24, x25, x26, x27, x28, x29, x30, x31, x32, x33,
 x34, x35, x36, x37, x38, x39, x40, x19 + x20, x20 + x21,
 x43, x44, x45, x46, x47, x48, x49, x50, x51, x52, x53, x54, x55,
 x56, x57, x58, x59, x60, x61, x62, x63, x41 + x42 + x43 + x56,
 x42 + x43 + x44 + x57]
```

4

Dans la mesure où on travaille sur $\mathbb{F}_2$, la fonction **Maj** peut s'exprimer sous la forme d'une équation booléenne.

On peut donc représenter ses différents résultats possibles en fonction de ses arguments a, b et c :

a	b	c	**Maj**(a,b,c)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Ce qui nous permet d'extraire une équation de forme DNF :

$$\begin{aligned}\textbf{Maj}(a,b,c) &= (a\oplus 1)bc \ \oplus\ a(b\oplus 1)c \ \oplus\ ab(c\oplus 1) \ \oplus\ abc \\ &= abc \ \oplus\ bc \ \oplus\ abc \ \oplus\ ac \ \oplus\ abc \ \oplus\ ab \ \oplus\ abc \\ &= \boxed{ab \ \oplus\ bc \ \oplus\ ac}\end{aligned}$$

On voit donc que la fonction **Maj** peut s'exprimer sous la forme d'une équation booléenne quadratique.

Or, si on revient à l'expression des registres sous forme de variables booléennes, et comme tous les éléments de la chiffre suitante sont créés à l'aide de la fonction **Maj** sur ces mêmes registres, on peut donc l'exprimer également sous la forme d'une équation quadratique.

5

La fonction permettant d'exprimer la chiffre suitante sous forme d'équations sur les registres est **Req2Zeq**:

```
def Req2Zeq(R4,N, vals=list(BPRv),wait=99):
    # returns the equation of the cipher
```

(***Note*** *: on remarquera que cette fonction se débarrasse déjà des variables représentant les portions des registres fixées à 1 à l'initialisation. Cela a été fait uniquement dans le but de simplifier la fonction de linéarisation qui sera présentée plus tard.*)

6

Nous allons faire la liste des monômes de degré non nul potentiellement présents dans les équations de la chiffre suitante, sachant que :

- la chiffre suitante étant créée exclusivement grâce à la fonction **Maj**, le degré des monômes sera majoré à 2,
- la fonction **Maj** n'étant utilisée que sur des éléments de même registre, les monômes de degré 2 ne sont constitués que de variables d'un même registre,
- on sait qu'à l'initialisation, certains éléments des registres sont fixés à 1, ce qui signifie que 3 variables (à savoir x_3, x_{23} et x_{45}), vaudront 1 et pourront être réduites des équations finales.

Ce dernier point revient à considérer, non pas 19, 22 et 23 variables, respectivement associées aux 1er, 2ème et 3ème registre, mais 18, 21 et 22.

On a donc :

les monômes de degré 1	:	$18+21+22$	$=61$
les monômes de degré 2 de R_1	:	$\binom{18}{2}$	$=153$
les monômes de degré 2 de R_2	:	$\binom{21}{2}$	$=210$
les monômes de degré 2 de R_3	:	$\binom{22}{2}$	$=231$

Ce qui nous fait un total de $\boxed{61+153+210+231=655}$ monômes possibles dans les équations de la chiffre suitante.

Cela nous confirme qu'on peut transformer les équations quadratiques exprimant la chiffre suitante en un système linéaire, bien plus facile à résoudre (si tant est qu'on possède une chiffre suitante de taille supérieure ou égale à 655).

On crée donc, d'abord, le vecteur **M** qui contiendra tous les monômes.
(***Note*** *: au sein du vecteur, les monômes sont placés dans leur ordre d'apparition dans la fonction* ***.monomials****, ce afin de de réduire la complexité de la fonction de linéarisation*)

7

La fonction **linearisator** se charge de... ben... linéariser les équations du système donné en argument par rapport au vecteurs des monômes associé :

```
def linearisator(eqs,var):
    # returns the linearization ('God this word is ugly...) matrix of eqs through var
    # and a vector containing the constants. Reducted to F2.
    # WARNING : var must follow the same order logic as .monomials()
```

La fonction est plus générale pour des utilisations ultérieures, mais dans le cas actuel, on obtient l'effet désiré en tapant, par exemple si on cherche à le faire avec les données du 2ème test, on tape :

```
sage: m,v = linearisator(Req2Zeq(TEST2_R4,700),M)
sage: m
700 x 655 dense matrix over Finite Field of size 2 (use the
.str() method to see the entries)
sage: len(v)
700
```

Les temps de calculs obtenus pour cette fonction sont dans l'ordre des 10s.

8

Nous avons codé la fonction **Z2Rs** qui s'occupe de retourner directement les états initiaux des registres :

```
def Z2Rs(R4,Z,wait=99):
    # returns R1, R2 and R3 according to R4 and Z
```

Si on utilise cette fonction sur les données du test 2, ici **TEST2_R4** et **TEST2_z**, on obtient :

```
sage: Z2Rs(TEST2_R4,TEST2_z)
([1,0,0,1,1,0,1,1,0,0,1,0,1,1,0,0,0,1,1],
 [0,0,0,0,1,1,0,1,1,1,1,0,1,1,0,1,0,0,0,1,1,0],
 [0,0,1,0,1,0,1,1,1,0,1,0,0,1,1,1,0,0,1,1,1,0,0])
```

Le contenu des registre etait donc:
$R_1 = (1,0,0,1,1,0,1,1,0,0,1,0,1,1,0,0,0,1,1)$
$R_2 = (0,0,0,0,1,1,0,1,1,1,1,1,0,1,1,0,1,0,0,0,1,1,0)$
$R_3 = (0,0,1,0,1,0,1,1,1,0,1,0,0,1,1,1,0,0,1,1,1,0,0)$

9

On veut maintenant pouvoir exprimer les états initiaux des registres en fonction des données d'initialisation, à savoir K et IV.

On considère, pour la kème LFSR de l'A5/2, A_k la matrice de rétroaction, et on considérera son registre comme un vecteur.

L'initialisation du registre par K/IV est une simple boucle en deux étapes :

- mettre la LFSR à jour
- ajouter un élément de K/IV

On va donc se contenter de montrer que, pour V un vecteur initial, faire passer V à travers une boucle d'initialisation via une clé K (toute ressemblance avec un K déjà existant serait fortuit) lui donne, à terme, la valeur :

$$V' = A_k^n V + \sum_{i=0}^{n-1} A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-1-i} \end{pmatrix}$$

On regarde pour une clé de taille minimale, c'est à dire 1.

On a :

$$V' = A_k V + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_0 \end{pmatrix}$$

Ce qui est exactement **UNE** étape de la boucle.

C'est donc bon à l'initialisation.

On suppose maintenant que la règle fonctionne jusqu'à $n \geq 1$

On a ainsi à terme de l'initialisation :

$$V' = A_k^n V + \sum_{i=0}^{n-1} A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-1-i} \end{pmatrix}$$

Maintenant supposons qu'on veuille connaître l'initialisation via une clé K' de taille $n+1$, dont les n premiers termes sont les mêmes que ceux de K. Selon le système, une telle initialisation revient à ajouter une étape supplémentaire qu'à celle de K.

On a donc :

$$
\begin{aligned}
V'' &= \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_n \end{pmatrix} + A_k V' \\
&= \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_n \end{pmatrix} + A_k \left(A_k^n V + \sum_{i=0}^{n-1} \left(A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-1-i} \end{pmatrix} \right) \right) \\
&= \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_n \end{pmatrix} + \sum_{i=0}^{n-1} \left(A_k^{i+1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-1-i} \end{pmatrix} \right) + A_k^{n+1} V \\
&= \left(A_k^0 \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-0} \end{pmatrix} \right) + \sum_{i=1}^{n} \left(A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{n-i} \end{pmatrix} \right) + A_k^{n+1} V \\
&= A_k^{n+1} V + \sum_{i=0}^{(n+1)-1} \left(A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{(n+1)-i} \end{pmatrix} \right)
\end{aligned}
$$

la règle est donc héréditaire.

Du coup, pour $n = 64$, et $V = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$ on a bien : $\boxed{X = \sum_{i=0}^{63} A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{63-i} \end{pmatrix}}$

et pour $n = 22$ et $V = X$, on a : $\boxed{A_k^{22} X + \sum_{i=0}^{21} A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{K}_{21-i} \end{pmatrix}}$

10

Pour obtenir K via R_4 et une chiffre suitante, nous avons créé quelques fonctions :

- **Keq2Req**, pour obtenir l'expression des registres en fonction de K :

```
def Keq2Req(IV,K=BPRv):
    {...}
```

- **Rs2K**, pour retrouver K étant donnés les registres initialisés :

```
def Rs2K(IV,R1,R2,R3,R4):
    {...}
```

- **O5_getK**, qui fait fonctionner **Rs2K** après avoir récupéré les registres via **Z2Rs** :

```
def O5_getK(IV,R4,Z,wait=99):
    {...}
```

Du coup, pour retrouver la clé d'initialisation ayant donné les données du 2ème test, il suffit de taper :

```
sage: K = O5_getK([0 for i in range(IVsize)],TEST2_R4,TEST2_z); K
[0,1,0,1,1,0,0,0,1,1,0,0,1,0,1,1,0,0,1,1,1,1,0,0,0,0,1,0,0,1,1,0,
 0,0,1,0,1,1,0,1,0,1,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0,1,1,0,0,0,0,1]
sage: O5(K=K,IV=[0 for i in range(IVsize)]).cipher(700) == TEST2_z
True
```

La clé du 2ème test est donc :

$$\mathrm{K} = \begin{pmatrix} 0,1,0,1,1,0,0,0,1,1,0,0,1,0,1,1,0,0,1,1,1,1,0,0,0,0,1,0,0,1,1,0, \\ 0,0,1,0,1,1,0,1,0,1,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0,1,1,0,0,0,0,1 \end{pmatrix}$$

11

À la question 9, nous avons remarqué que l'état initialisé d'un registre ressemblait à cette équation :

$$A_k^{22} X + \sum_{i=0}^{21} A_k^i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \mathrm{IV}_{21-i} \end{pmatrix}$$

avec X, l'état du registre après l'initialisation par K.

Dans la mesure où K est le même pour chacune des chiffre suitante générées, et que l'IV de celle générant z_0 est nulle, c'est à dire que l'état des registres sera $\mathrm{R}_{0_k} = A_k^{22} X$.

Ainsi on aura, $\forall k$, :

$$R_{1_k} = R_{0_k} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$$

et

$$R_{2_k} = R_{0_k} + A_k \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix} = R_{0_k} + \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

car aucun des polynômes de rétroaction ne possède de monôme de degré 1.

12

Nous disposons de 3 chiffres suitantes de taille 228 chacune.

Si on exprime chacune d'entre elle sous forme d'équation, comme plus tôt, en initialisant les valeurs des registres de départ à celles explicitées dans la question précédente, on se retrouve avec un système à $228 \times 3 = 684 > 655$, ce qui est suffisant pour s'assurer d'une unique réponse par linéarisation.

13

La fonction dans le .sage qui se charge de faire tout ce travail est la dernière fonction d'attaque du programme, **GSM_getK** :

```
def GSM_getK(R4, z0, z1, z2, wait=99):
    # given z0, z1, z2, and the initial R4 which created z0,
    # returns the key K
```

Cette fonction, une fois utilisée avec les données du 3ème et dernier test, donne :

```
sage: K = GSM_getK(TEST3_R4,TEST3_z0,TEST3_z1,TEST3_z2); K
[1,0,1,0,1,1,1,0,1,1,1,1,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0,0,1,
 0,1,0,0,0,1,1,1,1,1,1,0,0,1,0,1,1,1,0,1,1,1,1,0,1,1,0,1,1,0,1,1,0]
sage: IV0 = [0 for i in range(IVsize)]
sage: IV1 = [0 for i in range(IVsize-1)] + [1]
sage: IV2 = [0 for i in range(IVsize-2)] + [1,0]
sage: O5(K=K,IV0).cipher(228) == TEST3_z0
True
sage: O5(K=K,IV1).cipher(228) == TEST3_z1
True
sage: O5(K=K,IV2).cipher(228) == TEST3_z2
True
```

La clé de la 3ème batterie de test est donc :

$$\mathrm{K} = \begin{pmatrix} 1,0,1,0,1,1,1,0,1,1,1,1,0,0,0,0,0,0,1,1,0,0,1,1,0,0,0,1,1,0,0,1, \\ 0,1,0,0,0,1,1,1,1,1,1,0,0,1,0,1,1,1,0,1,1,1,1,0,1,1,0,1,1,0,1,1,0 \end{pmatrix}$$

14

L'attaque que nous avons vu tout au long de ce projet nous a montré que pour retrouver la clé de l'A5/2, il suffisait de connaître IV et l'initialisation de son 4ème registre.

Ainsi, si on considère que seul IV nous est dévoilé, on peut tenter une recherche exhaustive de K via R_4.

Des $2^{64} \geq 10^{19}$ possibilités d'une recherche irréfléchie, on se retrouve à seulement $2^{17} - 1 = 131\,071$.

Certes, si on multiplie toutes ces tentatives par les 10s passées à résoudre un système, on reste quand même environ 2 semaines à attendre de pouvoir craquer un système qui a été changé très certainement 2h après le début de l'attaque, mais c'est toujours mieux que celui qui est parti pour 5 billions d'années.

Après, les tests ont été effectués sur un ordinateur portable qui pleure quand deux pages internet sont ouvertes donc, si ça se trouve, avec quelques PS3 ça devrait être plus rapide.

(***Note*** *: Après quelques tests au CREMI, il s'avère que l'obtention de la clé ne requiert que 5s. Du coup, ça ne prendra qu'une semaine. Champagne !*)

(***Note*** *: Louis nous dit qu'il fait mieux. Louis adore détruire nos espoirs... *rebouche la bouteille**)

15

Dans la mesure où l'origine des attaques vient du contrôle de R_4 sur l'évolution des LFSRs de l'A5/2, une bonne idée serait de supprimer justement cette $4^{\text{ème}}$ LFSR, qui ne sert même pas à proprement parler à écrire la chiffre suitante, et d'effectuer la fonction **Maj** sur 1 élément de chaque registre. Ce faisant, il serait plus compliqué de casser le code avec un seul registre connu post initialisation, puisqu'on ne pourrait clairement déterminer leur évolution.

C'est d'ailleurs, à peu près, ce que fait A5/1, son prédécesseur, ce qui confirme une critique faite sur l'A5/2, considéré comme bien plus faible que son grand frère, déjà lui-même relativement cassable.

Il est à noter, d'ailleurs, que toutes ces attaques ont inspiré à Barkhan, Biham et Keller pour étendre leur concept aux autres protocoles de GSM (dont A5/1).

The End