Cryptanalyse de l'AES à 5 tours

Louis Coumau, Axel Durbet et Dhekra Mahmoud

14 mars 2021

Table des matières

Chapitre 1

Introduction

L'AES est considéré aujourd'hui comme le chiffrement symétrique le plus fiable et le plus utilisé et ce, depuis le début des années 2000. Effectivement, non seulement l'algorithme sur lequel il est basé consomme peu de mémoire mais a en plus la particularité de manipuler des fonctions élémentaires basiques.

L'AES a fait l'objet de beaucoup de recherche sur d'éventuelles attaques mais aucune significativement plus efficace que la recherche exhaustive n'a été découverte. Cependant, il existe des attaques plutôt efficaces sur ce que l'on appelle un AES réduit, c'est à dire un AES avec un nombre de tours réduit.

Dans ce rapport, nous commencerons par présenter le chiffrement AES complet. Puis, nous étudierons deux attaques contre l'AES réduit à 5 tours. La première attaque s'appelle l'attaque carré qui exploite la structure dite "carrée" de l'AES. La deuxième est une attaque différentielle dite "*yoyo*" qui exploite des propriétés fondamentales d'un réseau de permutation-substitution utilisé dans plusieurs chiffrements par bloc notamment l'AES.

Dans chaque partie, nous commencerons par introduire les outils mathématiques nécessaires à la formulation théorique de l'attaque à implémenter. Nous présentons ensuite le(s) pseudo-code(s) des algorithme(s) qu'on a implémenté(s). Et nous élaborerons enfin des détails relatifs à notre implémentation ainsi qu'une analyse de sa complexité.

Chapitre 2

Le chiffrement de Rijndael (AES)

2.1 Histoire

L'algorithme de Rijndael [2] est un chiffrement symétrique par bloc multiple de 32 bits qui a été conçu par Joan Daemen et Vincent Rijmen, deux chercheurs belges en 1997. Dans un concours du NIST (National Institute of Standards and Technology) visant à instaurer un nouveau standard car le `DES` (Data Encryption Standard) était devenu trop faible vis à vis des attaques de l'époque. Dans ce concours, la version 128 bits du chiffrement de Rijndael se placera dans les six premières places. Suite à cela, il s'impose comme le successeur du `DES` en 2000 sous le nom d'`AES` (Advanced Encryption Standard) devenant ainsi le deuxième véritable standard de la cryptographie.

Si le chiffrement de Rijndael est si important, c'est qu'il possède de nombreuses de propriétés intéressantes :

- Résistance à toutes les attaques connues à l'époque.
- Il est possible d'implémenter l'`AES` aussi bien sous forme logicielle que matérielle (câblé).
- Rapidité du code sur la plus grande variété de plates-formes (logicielles et matérielles) possible.
- Simplicité dans la conception.
- Besoins en ressources et mémoire très faibles.
- Il n'est pas basé sur un schéma de Feistel.

2.2 Principe

L'AES [2] est un chiffrement par bloc composé de quatre fonctions majeures :

— `SubBytes` (S-box)(2.4)
— `ShiftRows` (2.5)
— `MixColumns` (2.6)
— `AddRoundKey` (2.7)

Ces fonctions sont imbriquées de façon à créer un schéma de chiffrement itérable. Dans le standard, le nombre d'itération est 10,12 ou 14 selon la taille de la clé qui est de 128, 192 ou 256 bits.

Un algorithme de cadencement de clé calcule à partir de la clé K une suite de sous-clés de tour comportant toutes 16 octets qui seront utilisées pour le `AddRoundKey` (2.7).

Le texte à chiffrer est découpé en blocs de 16 octets et, pour chaque bloc clair, on effectue les opérations suivantes pour produire un bloc chiffré de même longueur :

— On initialise un tableau 4×4 avec 16 octets de texte clair.
— On applique successivement sur ce tableau les opérations suivantes :
  1. `AddRoundKey` (2.7)
  2. On effectue "nombre de tours total"-1 tours comportant les 4 étapes `SubBytes` (2.4), `ShiftRows` (2.5), `MixColumns` (2.6), `AddRoundKey` (2.7) dans cet ordre.
  3. Un dernier tour ne comporte plus que 3 étapes : `SubBytes` (2.4), `ShiftRows` (2.5), `AddRoundKey` (2.7).
— Le contenu actuel du tableau donne les 16 octets du texte chiffré.

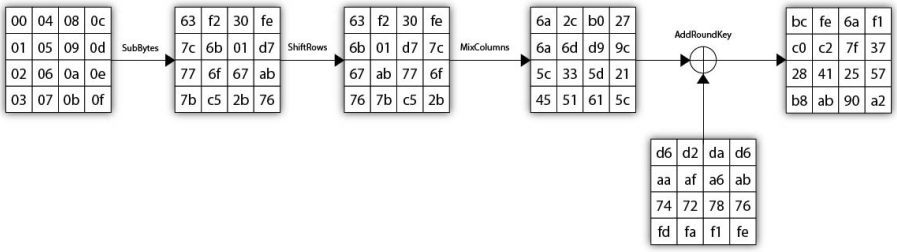


FIGURE 2.1 – Schéma d'une itération de l'AES

2.3 Pré-requis théoriques

2.3.1 Représentation de $\mathbb{F}_{256}$

Nous pouvons représenter ce corps par l'ensemble

$$\mathbb{F}_{256} \simeq (\mathbb{F}_2)^8 \simeq \mathbb{F}_2/\underbrace{(X^8 + X^4 + X^3 + X + 1)}_{m(X)}$$

Effectivement, le polynôme $m(X)$ est irréductible primitif, ainsi, l'isomorphisme existe bien entre les deux ensembles.

2.3.2 Multiplication

Regardons la multiplication par α dans $\mathbb{F}_{256}$. Prenons α, la classe de X, on a donc :

$$\begin{aligned} m(\alpha) = \alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1 = 0 \\ \alpha^8 = \alpha^4 + \alpha^3 + \alpha + 1 \end{aligned}$$

Ainsi, pour appliquer la multiplication de α par un polynôme en α, il suffira de remplacer le monôme α^8 par $\alpha^4 + \alpha^3 + \alpha + 1$.

2.4 SubBytes(S-box)

`SubBytes` est une fonction non linéaire qui opère indépendamment sur chacun des 16 octets. Elle utilise une opération sur le corps fini à 256 éléments ($\mathbb{F}_{256}$) (2.3.1).

Pour la S-box, on utilise la composée $S = F \circ I$ des applications suivantes :

— $I : \mathbb{F}_{256} \longmapsto \mathbb{F}_{256}$

$$x \mapsto \begin{cases} 0 & \text{si } x = 0 \\ x^{-1} & \text{sinon} \end{cases}$$

— $F : \mathbb{F}_{256} \simeq (\mathbb{F}_2)^8 \longmapsto (\mathbb{F}_2)^8 \simeq \mathbb{F}_{256}$

$x \mapsto Ax + B.$

Avec A et B comme suit :

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \text{ et } B = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Pour gagner du temps, on peut ensuite pré-calculer les 256 valeurs possibles et les écrire dans une table de substitution.

		Y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
X	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Exemple d'une Sbox pré-calculée

FIGURE 2.2 – Par exemple, `0xf6` $\xrightarrow{Sbox}$ `0x42`

2.5 ShiftRows

Le `ShiftRows` est une permutation circulaire vers la gauche aux lignes du tableau, respectivement de 0, 1, 2, 3 cases comme illustré ci-dessous :

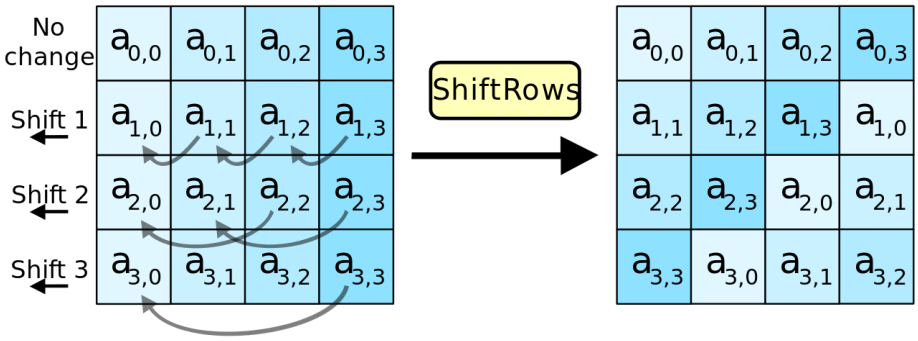


FIGURE 2.3 – Fonctionnement du `ShiftRows`

2.6 MixColumns

Le `MixColumns` est un produit matriciel à coefficient dans $\mathbb{F}_{256}$(2.3.1).

Soit la matrice A l'entrée du `MixColumns` et la matrice B le résultat de la fonction. Alors $M \times A = B$.

Avec $M = \begin{pmatrix} \alpha & \alpha+1 & 1 & 1 \\ 1 & \alpha & \alpha+1 & 1 \\ 1 & 1 & \alpha & \alpha+1 \\ \alpha+1 & 1 & 1 & \alpha \end{pmatrix}$

Mais chaque octet $(b_7, ..., b_0) \in (\mathbb{F}_2)^8$ est identifié avec l'élément $\sum_{i=0}^{7} b_i \times \alpha_i$ (2.3.1). Donc l'octet qui code α est $(0,0,0,0,0,0,1,0) = 2$. On peut donc réécrire la matrice comme :

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

2.7 AddRoundKey

`AddRoundKey` est une addition bit à bit ($\oplus$ ou XOR) de la clé de tour K_i, case par case.

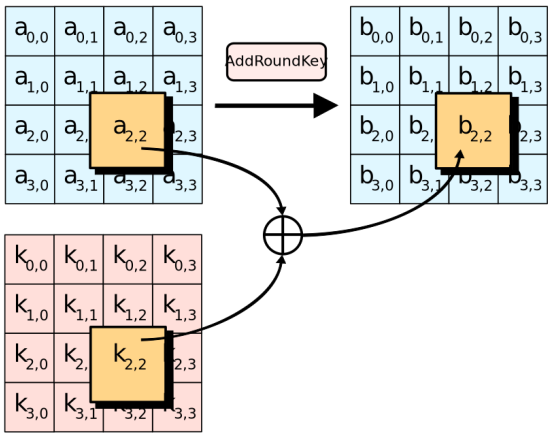


FIGURE 2.4 – `AddRoundKey`

2.8 Calcul des sous-clés

Par la théorie vu précédemment (2.3.2), voyons comment nous pourrions implémenter la multiplication dans $\mathbb{F}_{256}$. Pour cela, nous représenterons nos polynômes par des octets (2.3.1). Par exemple, on aura

$$\alpha^4 + \alpha^3 + \alpha + 1 = (0, 0, 0, 1, 1, 0, 1, 1)\ ^1$$

le calcul va donc se présenter de la façon suivante :

— Lorsque le degrés du produit est inférieur à 8, il n'y a simplement qu'a décaler tous les bits de l'octet représentant le polynôme.

Prenons un exemple avec $\alpha + 1$ que l'on peut représenter par 11_2 :

$$\begin{aligned}(\alpha + 1)\alpha &= \alpha^2 + \alpha \\ &= 110_2\end{aligned}$$

— Et lorsque le degrés est égal à 8, alors, comme dit précédemment, il suffit de remplacer α^8 par $\alpha^4 + \alpha^3 + \alpha + 1$ ce qui revient à appliquer un XOR avec l'octet 11011_2.

Donc, il suffit de décaler vers la gauche notre octet, et y appliquer un XOR avec 11011_2 si nécessaire.

1. Pour plus de simplicité, on omettra les bits de poids fort à 0. On écrira donc, dans ce cas, 11011_2

2.8.1 Rcon

Rcon (Round Constant) est une fonction définit comme suit :

$$Rcon(i) = \alpha^i$$

pour $i \geq 0$. On a donc les valeurs suivantes pour $0 \leq i \leq 9$:

$$\begin{aligned}
&\alpha^0 = 1 \\
&\alpha^1 = 10_2 = 2 \\
&\alpha^2 = 100_2 = 4 \\
&\quad\vdots \\
&\alpha^7 = 10000000_2 = 128 \\
&\alpha^8 = \alpha^4 + \alpha^3 + \alpha + 1 = 11011_2 = 27 \\
&\alpha^9 = 110110_2 = 54
\end{aligned}$$

2.8.2 UnrollKey

Le calcul des sous-clés se fait à partir d'une fonction inversible qui prend en entrée une clé K_i et un entier et qui sors une clé K_{i+1} dépendant de K_i. La fonction implémentée est :

```
bool UnrollKey(uchar *key, uchar round)
```

Posons K_0 la clé initiale et K_{i+1} la clé calculée par la fonction avec les paramètres K_i et i. Posons aussi $K_{i,Cj}$ la colonne j de la clé K_i et K'_i la copie de la clé K_i.

La procédure pour calculer K_{i+1} en fonction de K_i est la suivante :
- On effectue une rotation de 1 cran vers le haut de $K'_{0,C3}$.
- On applique la fonction S-Box sur chaque octets de cette même colonne.
- On calcule un XOR entre la première valeur de la colonne $K'_{0,C3}$ et Rcon(i).
- On effectue les XOR successifs suivants :
  - $K_{i+1,C0} = K'_{i,C3} \oplus K_{i,C0}$
  - $K_{i+1,C1} = K'_{i,C0} \oplus K_{i,C1}$
  - $K_{i+1,C2} = K'_{i,C1} \oplus K_{i,C2}$
  - $K_{i+1,C3} = K'_{i,C2} \oplus K_{i,C3}$

La procédure est aussi représenté dans la Figure 2.5. On y voit, en jaune, les fonctions :

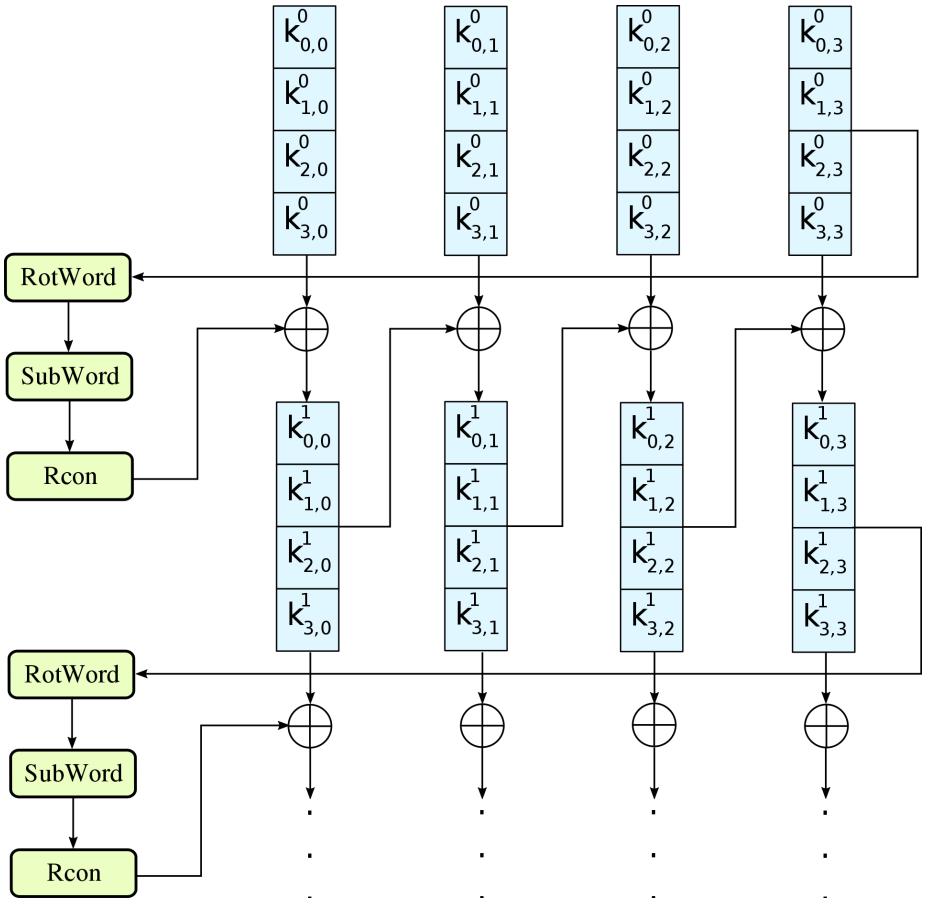


FIGURE 2.5 – Calcul d'une sous-clé

— `RotWord` qui applique un décalage circulaire de bas en haut.
— `SubWord` qui applique la S-Box à chaque octet de la colonne.
— `Rcon` qui XOR Rcon(i) avec le premier octet de la colonne.

Examinons un exemple :
— État initial :

```
        00 01 02 03
        04 05 06 07
K0  =   08 09 0A 0B
        0C 0D 0E 0F
```

— On applique la rotation, la Sbox et Rcon sur la dernière colonne de $K0'$ pour ne pas modifier la clé d'origine :

```
03          07          C5     Rcon[i]          C4
07   RotW   0B   Sbox   2B   ^   00             2B
0B    =>    0F    =>    76        00      =>    76
0F          03          7B        00             7B
```

— On applique ensuite le XOR successif sur les colonnes :

```
                C4     00       C4
K0'C3 ^ K0C0 =  2B  ^  04  =>   2F  = K1C0
                76     08       7E
                7B     0C       77

                C4     01       C5
K1C0 ^ K0C1  =  2F  ^  05  =>   2A  = K1C1
                7E     09       77
                77     0D       7A

                C5     02       C7
K1C0 ^ K0C2  =  2A  ^  06  =>   2C  = K1C2
                77     0A       7D
                7A     0E       74

                C7     03       C4
K1C0 ^ K0C3  =  2C  ^  07  =>   2B  = K1C3
                7D     0B       76
                74     0F       7B
```

On obtient donc :

```
C4 C5 C7 C4
2F 2A 2C 2B
```

```
K1  =   7E 77 7D 76
        77 7A 74 7B
```

Remarque 2.8.1. Si on précalcule la fonction Rcon, il y a autant de valeur à précalculer que de tours prévus pour chiffrer.

Remarque 2.8.2. Si on effectue i tours de l'`AES`, on a à calculer i sous-clés, et donc stocker $i + 1$ clés.

En introduisant la fonction d'extension de la clé K_0, nous avons parlé du fait qu'elle était inversible. Effectivement, chaque étape de cette fonction peut être inversée grâce aux XOR, à l'inverse de la Sbox et à Rcon qui donne toujours la même image en fonction du tour.

Autrement dit, si l'on trouve K_i, $\forall i$ et que l'on connaît la valeur de i, alors on peut remonter jusqu'à K_0.

2.9 Le déchiffrement de l'AES

Le déchiffrement de l'`AES` est composé de quatre fonctions majeures :

— `ISubBytes` (IS-box)(2.10)
— `IShiftRows` (2.11)
— `IMixColumns` (2.12)
— `AddRoundKey` (2.7)

Ces fonctions sont imbriquées de façon à créer un schéma de déchiffrement itérable.

Le texte à déchiffrer est découpé en blocs de 16 octets et, pour chaque bloc chiffré, on effectue les opérations suivantes pour produire un bloc clair de même longueur :

— On initialise un tableau 4×4 avec 16 octets de texte chiffré.
— On applique successivement sur ce tableau les opérations suivantes :
  1. `AddRoundKey` (2.7), `IShiftRows` (2.11) et `ISubBytes` (2.10) dans cet ordre.
  2. On effectue "nombre de tours total"-1 tours comportant les 4 étapes `AddRoundKey` (2.7), `IMixColumns` (2.12) `IShiftRows` (2.11) puis `ISubBytes` (2.10).
  3. Un dernier tour ne comporte plus qu'une étapes : `AddRoundKey` (2.7).
— Le contenu actuel du tableau donne les 16 octets du texte déchiffré.

2.10 ISubBytes (IS-box)

Pour calculer `ISubBytes` , il suffit de lire le tableau de `SubBytes` (2.4) dans l'autre sens car la fonction est bijective.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

FIGURE 2.6 – Par exemple, `0x53` $\xrightarrow{IS-box}$ `0xed`

2.11 IShiftRows

La fonction `IShiftRows` est l'exacte inverse de `ShiftRows` (2.5).

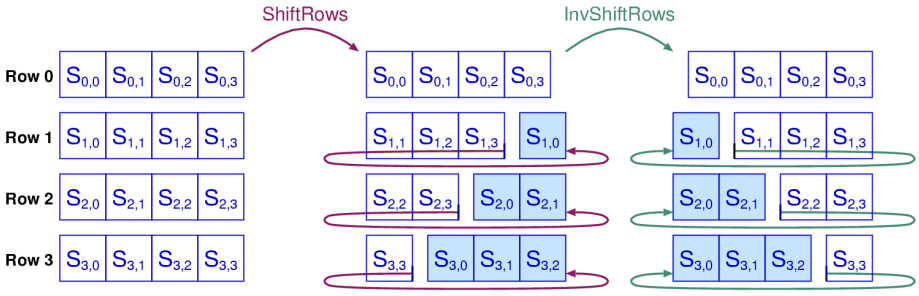


FIGURE 2.7 – Fonctionnement de `IShiftRows` et `ShiftRows`

2.12 IMixColumns

On a vu dans le `MixColumns` (2.6) qu'il suffisait de calculer $M \times A = B$ avec A l'entrée du `MixColumns` et B le résultat.

Ici, c'est B qui va jouer le rôle de l'entrée et A celui de la sortie. On peut donc écrire que l'`IMixColumns` se calcule avec le produit matriciel $M^{-1} \times B = A$.

Il suffit d'inverser M dans $\mathbb{F}_{256}$ et on trouve :

$$M^{-1} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$$

Chapitre 3

Différents types d'attaques

3.1 L'attaque carré

3.1.1 Le chiffrement SQUARE

Square est une construction itérative de chiffrement par bloc. Chaque bloc contient 128 bits et est constitué de 16 octets rangés dans un tableau de taille 16 (soit un octet dans chaque case). La fonction de tour correspondante est constituée de quatre transformations élémentaires :

— `Linear Transformation` :
C'est une transformation linéaire qui opère séparément sur toutes les lignes d'un état (modélisé par un tableau comme expliqué ci-dessus). Chaque octet d'une même ligne serait le résultat d'une combinaison linéaire de tous les octets de la même ligne. Cette transformation correspondrait à une `MixColumns` (2.6) composée avec une transposition dans le cas de l'`AES`.

— `Nonlinear Transformation`
Cette transformation opère sur chaque octet séparément. Elle correspond à une application d'une table de substitution inversible ou S-box.

— `Byte Permutation`
Cette permutation correspond à une transposition. Appliquée à un tableau d'octets, elle échange les lignes et les colonnes.

— `Bitwise RoundKey Addition`
Comme son nom l'indique, c'est tout simplement une addition bit à bit avec la clef de tour. Elle correspond à la fonction `AddRoundKey` (2.7) de l'`AES`.

L'attaque carré fonctionne uniquement sur les chiffrements qui possèdent une structure dite carrée inspirée du chiffrement (du même nom) ci-dessus.

Cette attaque reste valable pour l'AES car il hérite de nombreuses propriétés du chiffrement Square.

3.1.2 Le principe de l'attaque

Ici, nous allons uniquement parler de l'attaque sur l'AES.

L'attaque square est dite CPA (chosen-plaintext attack) ou encore à clair choisi. Ainsi, nous avons accès aux couples (clairs,chiffrés) avec lesquels nous voulons travailler. L'objectif de l'attaque est de retrouver intégralement la clé et donc de casser l'AES à 4 tours puis à 5.
Nous noterons que la fonction G qui produit les sous-clés à partir de la clé initiale est inversible. De ce fait, retrouver une sous-clé permet de remonter à l'aide de G^{-1} jusqu'à la clé initiale.

Dans l'attaque square, nous proposons de retrouver la dernière sous-clé (celle utilisée dans le dernier tour) afin de casser le chiffrement.

Comme c'est une attaque à clair choisi, la première question à se poser est : Quels sont les messages clairs qu'il est intéressant de regarder ?

3.1.2.1 Les Λ-sets

Soit Λ-set un ensemble de 256 états différents de 128 bits chacun. Pour décrire un état (ou un élément de cet ensemble), on adopte la même représentation matricielle que celle présentée pour le chiffrement carrée (soit un tableau bidimentionnel de 4×4 comportant en tout 16 octets). Ainsi, si x est un élément de Λ, l'octet d'indice (i, j) et qui sera représenté par $x_{i,j}$ correspond à l'octet appartenant à la $i$ème ligne et à la $j$ème colonne de l'état x. Avec ces notations, un octet d'indice (i, j) est dit `actif`, si tous les octets de cet indice-là de tous les états d'un même Λ-set, sont différents deux-à-deux. Un octet d'indice (i, j) est dit `passif`, s'il est le même pour tous les états. Ainsi, nous construisons notre ensemble Λ-set avec des octets `actif` et `passif` uniquement. Soit λ l'ensemble des indices des octets dits `actif`, alors un Λ-set est défini comme suit :

$$\forall x, y \in \Lambda : \begin{cases} x_{i,j} \neq y_{i,j} & \text{si } (i,j) \in \lambda \\ x_{i,j} = y_{i,j} & \text{sinon} \end{cases}$$

Voyons un exemple simple d'un Λ-set avec un seul octet actif (l'octet d'indice $(0, 0)$) :

```
00 FF FF FF     01 FF FF FF     FE FF FF FF     FF FF FF FF
```

```
FF FF FF FF     FF FF FF FF     FF FF FF FF     FF FF FF FF
FF FF FF FF ;   FF FF FF FF ... FF FF FF FF ;   FF FF FF FF
FF FF FF FF     FF FF FF FF     FF FF FF FF     FF FF FF FF
```

Dans cet exemple, nous avons choisi de faire varier le premier octet de 0 jusqu'à 255 et de réduire tous les autres octets à la valeur 255 (valeur indicative et pas exhaustive) pour former notre ensemble de Λ-set. Nous remarquons que nos états contiennent toutes les valeurs possibles pour l'octet actif.

Nos ensembles de clairs, avec lesquels nous allons travailler, seront des Λ-sets car ils possèdent de bonnes propriétés que nous pourrons exploitées pour monter notre attaque. En effet, l'application de `SubBytes` (2.4) ou `AddRoundKey` (2.7) sur un Λ-set donne généralement un autre Λ-set avec les indices des octets actifs inchangés. L'application de `MixColumns` (2.6) n'aboutit pas nécessairement à un Λ-set. Mais, une colonne d'entrée avec un seul octet actif donne une colonne de sortie avec les quatre octets actifs. L'application de `ShiftRows` (2.5) donne un autre Λ-set avec les indices des octets actifs changés.

3.1.2.2 Sur 4 tours

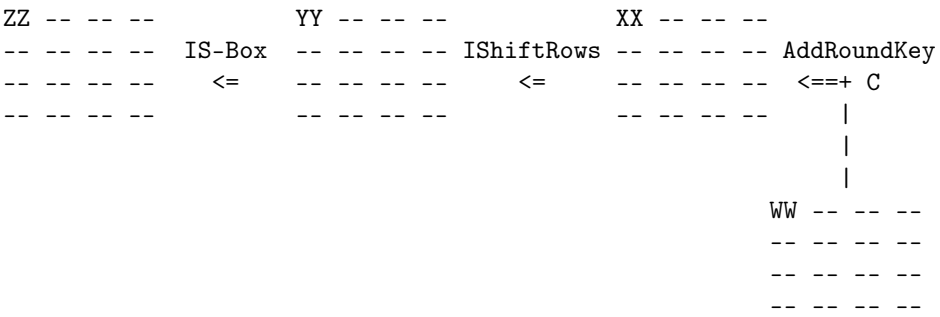
Considérons un Λ-set avec un seul octet actif. Avec les propriétés ci-dessus, notre Λ-set reste un Λ-set jusqu'à l'entrée de `MixColumns` du $3^{\text{ème}}$ tour avec tous les octets actifs.

Mais comme les octets des Λ-sets contiennent toutes les valeurs possibles, elles sont donc équilibrées (balanced). Ainsi, tous les octets à l'entrée du $4^{\text{ème}}$ tour sont équilibrés. Cette propriété est détruite une fois qu'on passe par la S-box.

On se place maintenant en sortie de chiffrement de notre Λ-set. On fait une hypothèse sur un octet de la dernière sous-clé et on applique les fonctions inverses pour retrouver l'état de l'entrée du $4^{\text{ème}}$ tour. Si notre hypothèse de sous-clé est la bonne, les octets correspondants seront équilibrés.

Voyons une représentation de la remonté du tour 4 en agissant sur le premier octet de la clé avec `C` les chiffrés du Λ-set, `WW` l'octet de la clé sur lequel on agit, et `XX`, `YY`, `ZZ` les emplacements [1] de l'octet après modification.

1. On notera que les emplacements restent inchangées car `ShiftRows` n'agit pas sur la première ligne, et donc, `IShiftRows` non plus.



D'après le schéma, une fois arrivé à `ZZ` on sera arrivé à l'entrée du 4ème tour, on pourra ainsi sommer tous les octets `ZZ` de tous les éléments du Λ-set pour vérifier la propriété d'équilibre.

De ce fait, si on attaque le j-ème octet, on calcule : $\sum_{i=1}^{256} a_{i,k}$ avec l'addition bit à bit, $a_{i,k}$ le k-ème octet du i-ème chiffré du Λ-set et

$$k = \begin{cases} j & \text{si } 0 \leq j < 4 \\ (j+1)[\text{mod } 4] + 4 & \text{si } 4 \leq j < 8 \\ (j+2)[\text{mod } 4] + 8 & \text{si } 8 \leq j < 12 \\ (j+3)[\text{mod } 4] + 12 & \text{sinon} \end{cases}$$

(c'est le résultat de `IShiftRows` sur l'octet j).

Si notre j-ème octet de sous-clé n'est pas bon, on a : $\sum_{i=1}^{256} a_{i,k} \neq 0$.

Un seul Λ-set ne suffit pas toujours pour éliminer toutes les hypothèses sauf une. Cependant, en utilisant seulement deux Λ-sets différents, on peut retrouver les octets de la sous-clé un par un en étant sûre d'éliminer toutes les valeurs de clé possible sauf une.

Avec cette méthode, on retrouve rapidement la sous-clé du dernier tour et avec elle, la clé initiale grâce à G^{-1} (3.1.2).

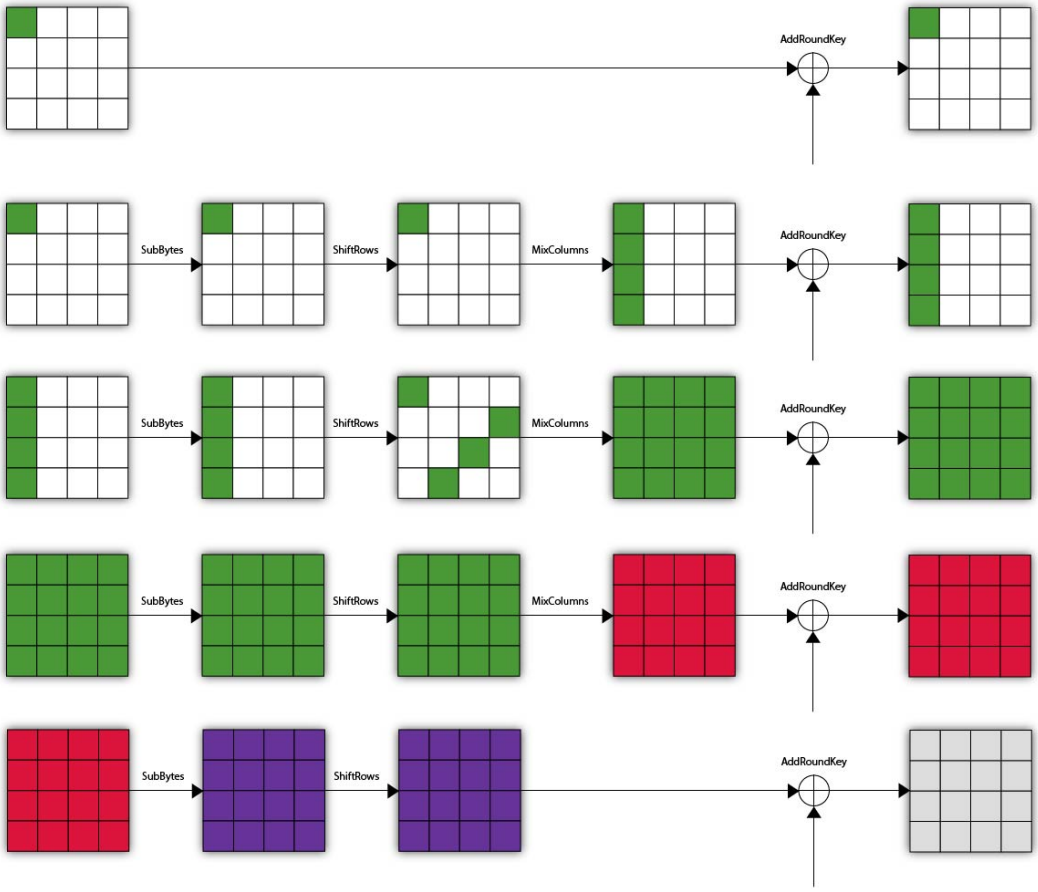


FIGURE 3.1 – Attaque square sur 4 tours.

3.1.2.3 Pseudo-code : Attaque carrée sur l'AES 4 tours

Algorithme 3.1.2.1. [Attaque carrée sur l'AES 4 tours]

Entrée : 2 Λ*-sets (3.1.2.1) différents chiffrés et* G^{-1} *(3.1.2)*

Sortie : La clé

1. *On applique* `IShiftRows` *à tous nos chiffrés.*
2. *Pour chaque octet de la dernière clé, faire :*
   (a) *Assigner une valeur à l'octet de la clé.*
   (b) *Pour chaque chaque chiffré du* Λ*-set 1 faire la somme (de XOR) suivante :* $\sum_{i=1}^{256} ISbox(a_{i,j} \oplus K_j)$ [2] *avec* $a_{i,j}$ *le* j*-ème octet*[3] *du* i*-ème chiffré du* Λ*-set et* K_j *le* j*-ème octet de la clé.*
   (c) *On réitère l'opération sur chaque chiffré du* Λ*-set 2.*
   (d) *Vérifier si les deux sommes sont nulles.*
   — *Si oui : Passez a l'octet de clé suivant.*
   — *Si non : Changer la valeur de l'octet de la clé.*
3. *Remonter avec la sous-clé vers la clé initiale grâce à* G^{-1}.
4. *Retourner la clé.*

---

2. On prend l'inverse de `SubBytes` pour ne pas se tracasser des changement d'indice vu ici : 3.1.2.2
3. Ici, celui de l'hypothèse de clé.

3.1.2.4 Extension : Ajout d'un tour à la fin (AES 5-tours type-1)

La première différence est qu'il faut ici, cinq Λ-sets pour être sûre d'éliminer toutes les valeurs de sous-clé sauf une.

La deuxième différence est qu'on va retrouver les octets de la dernière sous-clé 4 par 4.

Pour cette extension, il faut noter quelque chose de fondamentale, après le $3^{\text{ème}}$ tour, on a perdu les propriétés d'équilibre car on ne fait plus face à des Λ-sets. De ce fait, l'attaque se déroule un peu différemment. En effet, on va d'abord faire une hypothèse sur 4 octets de la sous-clé 5 (la dernière), remonter à la fin du tour 4 (l'avant dernier) et faire une hypothèse sur l'octet de la clé correspondant par les opérations inverses (par exemple : l'octet 1 de la sous-clé 4 correspond aux octets 1,8,11 et 14 de la sous-clé 5). Ainsi, on peut revenir au début du tour 4 et vérifier la propriété d'équilibre car on retrouve un Λ-set. Si elle est vérifié (la somme est nulle) alors nos 4 octets de la clé 5 sont bons. Pour être plus précis, il faut que les 5 Λ-set valident la propriété en même temps pour dire que l'hypothèse est bonne.

On retrouve ainsi les 16 octets de la sous-clé 5 et donc la clé grâce à G^{-1} (3.1.2).

3.1.2.5 Pseudo-code : Attaque carrée sur l'AES 5-tours type-1

Algorithme 3.1.2.2. [Attaque carrée sur l'AES 5 tours]

Entrée : 5 Λ-*sets (3.1.2.1) différents chiffrés et* G^{-1} *(3.1.2)*

Sortie : La clé

1. *Assigner une valeur aux octets* 0, 7, 10 *et* 13 *de la dernière sous-clé ainsi qu'a l'octet* 0 *de la sous-clé précédente et* 0 *sur les autres octets.*
2. *Appliquer les fonctions inverses aux chiffrés suivantes :*
   (a) `AddRoundKey` *avec la dernière clé supposée (la sous-clé 5).*
   (b) `IShiftRows` .
   (c) `ISubBytes` .
   (d) `AddRoundKey` *avec l'avant dernière clé supposée (la sous-clé 4).*
   (e) `IMixColumns` .
   (f) `IShiftRows` .
   (g) `ISubBytes` .
3. *Pour chaque chaque chiffré du* Λ*-set* 1 *faire la somme (de XOR) suivante :* $\sum_{i=1}^{256} a_{i,j}$ *avec* $a_{i,j}$ *le* j*-ème octet (ici* $j = 0$*) du* i*-ème chiffré du* Λ*-set.*
4. *Faire la même chose avec chaque chiffré des autres* Λ*-set.*
5. *Vérifier si les sommes sont nulles.*
   — *Si oui : On continue.*
   — *Si non : Changer les valeurs des octets supposés des clés* 4 *et* 5.
6. *Assigner une valeur aux octets* 1, 4, 11 *et* 14 *de la dernière sous-clé ainsi qu'a l'octet* 1 *de la sous-clé précédente et* 0 *sur les autres octets. On refait la même chose qu'au dessus (étapes* 2,3,4 *et* 5*).*
7. *Assigner une valeur aux octets* 2, 5, 8 *et* 15 *de la dernière sous-clé ainsi qu'a l'octet* 2 *de la sous-clé précédente et* 0 *sur les autres octets. On refait la même chose qu'au dessus (étapes* 2,3,4 *et* 5*).*
8. *Assigner une valeur aux octets* 3, 6, 9 *et* 12 *de la dernière sous-clé ainsi qu'a l'octet* 3 *de la sous-clé précédente et* 0 *sur les autres octets. On refait la même chose qu'au dessus (étapes* 2,3,4 *et* 5*).*
9. *A ce stade, on a retrouver intégralement la dernière clé. On remonte donc avec la sous-clé vers la clé initiale grâce à* G^{-1}.
10. *Retourner la clé.*

3.1.2.6 Pseudo-code : Attaque carrée sur l'AES 5-tours type-1

Algorithme 3.1.2.3. [Attaque carrée sur l'AES 5 tours]

Entrée : 5 Λ-*sets (3.1.2.1) différents chiffrés et* G^{-1} *(3.1.2)*

Sortie : La clé K_0

1. *(Initialisation) On définit les ensembles des positions suivants :*
   (a) $I_0 = [0, 7, 10, 13]$, $I_1 = [1, 4, 11, 14]$, $I_2 = [2, 5, 8, 15]$, $I_3 = [3, 6, 9, 12]$ [4]
   (b) $J = [0, 2, 1, 3]$
2. *(Initialisation) On crée les clés* K_5 *et* K_4 *avec tous les octets à* 0
3. *Pour* $n \in [0, 1, 2, 3]$
4. *Pour toutes les valeurs possibles de* K_5 *aux indices* I_n *:*
5. *Pour toutes les valeurs possibles de* K_4 *à l'indices* J_n *:*
6. $b \longleftarrow 0$
7. *Pour tous les* Λ*-sets :*
8. *Pour tous les chiffrés* C_i *:*
9. $C'_i \longleftarrow$ *On remonte le tour 5 de* C_i *avec la clé* K_5 [5]
10. $C'_i \longleftarrow$ *On remonte le tour 4 de* C'_i *avec la clé* K_4
11. $b \longleftarrow b \oplus C'_i[J_n]$
12. *Si* $b = 0$ *:*
13. *Alors on a trouvé les bonnes valeurs de* K_5
14. *aux indices* I_n*.*
15. *Arrêter ou passer à la valeur de n suivante.*
16. **À cette étape, on a tous les octets de* K_5 ***
17. $K_0 \longleftarrow$ *On remonte la clé* K_5 *avec* G^{-1} *5 fois*
18. *On retourne* K_0

Représentation des ensembles I_n par les positions :

```
    I0              I1              I2              I3

XX __ __ __     __ XX __ __     __ __ XX __     __ __ __ XX
__ __ __ XX     XX __ __ __     __ XX __ __     __ __ XX __
__ __ XX __     __ __ __ XX     XX __ __ __     __ XX __ __
__ XX __ __     __ __ XX __     __ __ __ XX     XX __ __ __
```

4. Ci-après la représentation des positions.
5. On précise que dans ce cas, il s'agit du dernier tour.

3.1.2.7 Extension : Ajout d'un tour au début (AES 5-tours type-2)

L'idée principale de cette extension est le fait de se ramener, après avoir effectué 5 tours, à un état semblable à l'AES 4-tours préalablement attaqué.

Il s'agit, alors, de choisir un ensemble de clairs qui forme un Λ-set ayant un seul octet actif à la sortie du premier tour. On remarque que si l'état intermédiaire après l'application de MixColumns du premier tour possède un seul octet actif, alors ceci reste aussi valable pour l'entrée du second tour puisque l'application de AddRoundKey sur ce Λ-set donnerait un autre Λ-set avec les mêmes octets actifs. Pour se ramener à ce cas de figure, il faut imposer des conditions particulières sur quatre octets constituant une colonne d'entrée sur laquelle MixColumns agit (MixColumns agit sur chaque colonne indépendemment). Étant donnée que cette dernière opération fait correspondre à chaque octet d'une colonne une combinaison linéaire de tous les octets de cette même colonne, une première condition serait qu'une seule et unique combinaison linéaire doit varier et parcourir toutes les 256 valeurs possibles pour des 256 états différents. La deuxième condition porterait sur les trois autres combinaisons linéaires qui doivent être toujours constantes. Ceci impose les mêmes conditions sur ces quatre octets qui figurent dans différentes positions à l'entrée de ShiftRows du premier tour. Par exemple si l'on souhaite avoir comme octet actif à la sortie de MixColumns du premier tour un octet appartenant à la première colonne, les quatre octets mis en jeu à l'entrée de ShiftRows seraient les octets d'indices $[0, 5, 10, 15]$. Il faut veiller à ce qu'une seule combinaison linéaire de ces quatre octets varie et que toutes les autres soient constantes. Si ces quatre octets de la clef K_0 sont connus à l'avance, ces conditions peuvent être modifiées et imposées directement sur les quatre octets correspondant des clairs choisis (puisque l'application de SubBytes du premier tour sur ce Λ-set donnerait un autre Λ-set avec les mêmes octets actifs).

On considère un ensemble de 2^{32} clairs différents tels qu'une seule colonne soit active à l'entrée de la MixColumns du premier tour (tout en faisant une supposition sur les quatre octets correspondant de la clef K_0). Parmi ces 2^{32} clairs disponibles, un ensemble de 256 clairs serait choisi et qui constituerait un Λ-set avec un seul octet actif à l'entrée du second tour. Ainsi, l'attaque carrée sur AES 4-tours (3.1.2.3) pourrait être faite. Cette attaque devra être répétée à plusieurs reprises avec plusieurs ensembles de clairs jusqu'à obtenir des résultats cohérents.

Cette attaque s'avère très coûteuse en terme de mémoire par rapport à l'attaque sur l'AES 5-tours type-1. Il s'agit de stocker et de manipuler 2^{32} clairs contre au plus 2^{11} pour l'autre attaque. Donc, nous avons pensé à l'améliorer.

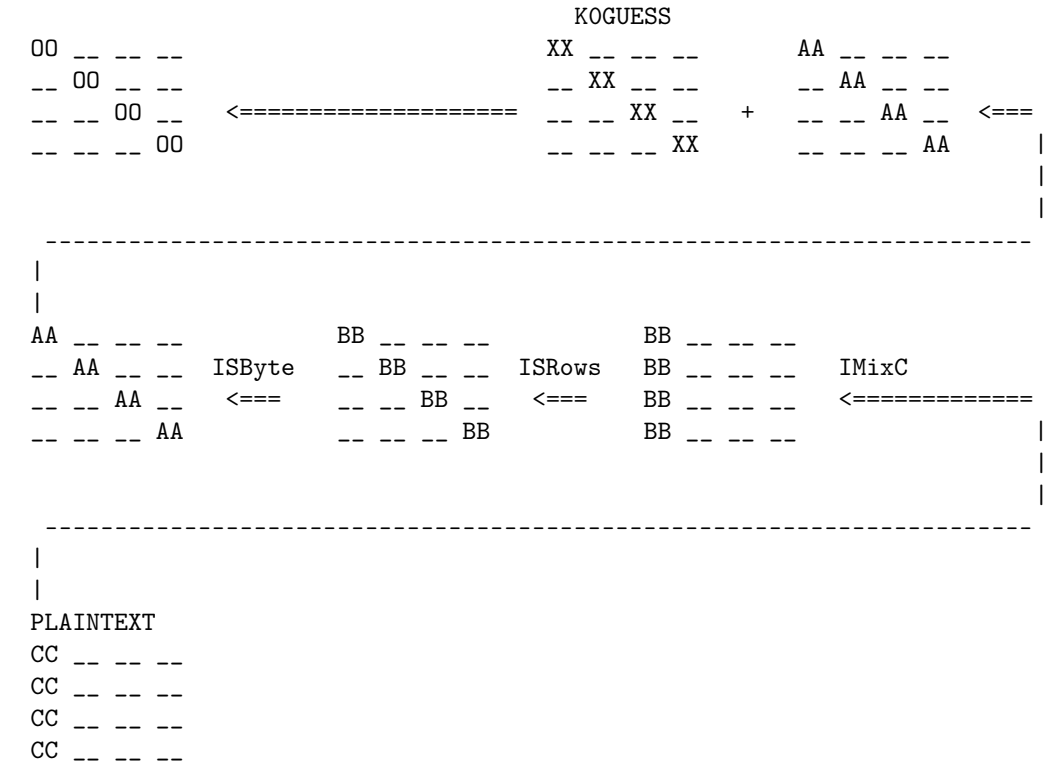
3.1.2.7.1 Amélioration de l'attaque : Nous nous sommes posés la question suivante : Comment pourrait-on construire des ensembles de clairs qui nous

donnent directement des Λ-sets ayant un seul octet actif à la sortie du premier tour ?

On considère un Λ-set de clairs avec un seul octet actif sur la première colonne et on applique dessus les fonctions `IMixColumns` , `IShiftRows` et `ISubBytes` dans cet ordre. Si l'on omet l'étape d'initialisation des clairs avec la clef K_0, la sortie du premier tour serait un Λ-set avec un seul octet actif sur la première colonne (car nos fonctions élémentaires relatives au premier tour vont se compenser avec leurs fonctions inverses appliquées préalablement aux ensembles des clairs). Si l'on considère maintenant la version complète de l'`AES` (l'étape d'initialisation avec la clef K_0 incluse), nos clairs seraient "perturbés" par l'ajout de K_0 et nous nous trouverions, à la sortie du premier tour, avec toute une colonne d'actifs (la première). Il est à noter que nos clairs "modifiés" (les clairs après avoir appliqué les fonctions inverses comme expliqué ci-dessus) forment un Λ-set dont les octets 0, 5, 10 et 15 sont actifs. Donc, il suffit de connaître les octets 0, 5, 10 et 15 de K_0 pour pouvoir annuler l'effet de l'étape d'initialisation sur l'ensemble des clairs.

Ainsi, cette attaque se déroule en deux étapes. La première étape serait : la vérification des quatre octets de la clef K_0. On suppose connaître les octets 0, 5, 10 et 15 de la clef K_0. Les autres octets de la clé supposée sont initialisés à 0. On additionne cette dernière bit à bit avec chacun de nos clairs "modifiés" et on les chiffre en effectuant 5 tours d'`AES`. Si nos suppositions sont bonnes, les octets 0, 5, 10 et 15 de la clef K_0 rajoutés à nos clairs "modifiés" avant leur chiffrement, annuleraient l'effet de l'étape d'initialisation sur notre Λ-set. Et nous nous retrouvons à l'entrée du cinquième tour avec un ensemble d'états dont la somme est équilibrée pour chacun des octets. Donc, il suffit de choisir un seul octet de la clef du tour 5 à deviner et qu'avec lequel nous remontons le dernier tour. Une fois qu'on a vérifié la cohérence de nos hypothèses sur les quatre octets de K_0 et un octet de K_5, on procède à la deuxième étape : attaquer les 15 autres octets de K_5 en suivant la même démarche évoquée dans la partie Attaque carrée sur l'`AES` à 4 tours (3.1.2.3).

Schéma de l'application les fonctions inverses et `AddRoundKey` avec `KOGUESS` sur les plaintexts des Λ-sets.



3.1.2.8 Pseudo-code : Attaque carrée sur l'AES 5-tours type-2

Algorithme 3.1.2.4. [Attaque carrée sur l'AES 5-tours type-2]

Entrée : 4 Λ*-sets (3.1.2.1) différents possédant un seul octet actif sur la première colonne et* G^{-1} *(3.1.2)*

Sortie : La clé secrète

1. *On applique respectivement l'inverse des fonctions* `MixColumns` , `ShiftRows` *et* `SubBytes` *sur chaque élément des* Λ*-sets.*
2. *On initialise les octets de nos clefs* K_0 *et* K_5 *à 0.*
3. *Pour chaque valeur possible des octets* 0, 5, 10 *et* 15 *de* K_0 *faire :*
   (a) *On additionne bit à bit les éléments des* Λ*-sets avec* K_0*. Puis, on les chiffre.*
   (b) *Pour chaque valeur possible* k *de l'octet* 0 *de la clef* K_5 *faire :*
      i. *Pour chaque chiffré* $c_{i,j}$ *du* Λ*-set* i*, on calcule la somme suivante :* $b_i = \sum_{j=1}^{256} ISbox(c_{i,j,0} \oplus k)$ *telle que* $c_{i,j,0}$ *correspond à l'octet* 0 *du* j*-ème chiffré du* i*-ème* Λ*-set.*
      ii. *Si les quatre* b_i *sont nuls alors, on obtient les bonnes valeurs pour les quatre octets de* K_0 *ainsi que la bonne valeur de l'octet* 0 *de* K_5 *et on sort de la grande boucle.*
      iii. *Sinon, on passe à l'octet suivant.*
4. *On récupère les* 15 *autres octets de la clef* K_5 *en suivant la même démarche de l'attaque réalisée sur l'*`AES` *à* 4 *tours (3.1.2.3).*
5. *On retourne la clé.*

3.1.3 Implémentation

Notre implémentation du chiffrement de l'`AES` repose sur la modification du contenu d'une suite de 16 octets[6]. Les fonctions vont donc manipuler des pointeurs. Cette méthode est très bénéfique en terme de mémoire car elle évite d'avoir à allouer de l'espace à chaque manipulation d'un état.

Cependant cela pose un problème dans le cas ou l'on souhaite conserver l'état initial du message modifié. Par exemple, dans l'attaque sur l'`AES` à 5 tours, lorsque l'on applique la remontée des tours 5 et 4 après avoir généré certains octets de K_5 et un octets de K_4, tous les chiffrés des Λ-sets se trouvent remontés, et donc, lorsque l'on passe à d'autres possibilités de K_5 ou K_4, l'attaque ne fonctionne plus car les chiffrés ne sont plus ceux d'origine.

Ainsi, nous avons travaillé avec une structure permettant de manipuler un clair, un chiffré, et un chiffré temporaire.

```
1 typedef struct {
2   uchar plaintext[CELLS];
3   uchar ciphertext[CELLS];
4   uchar ciphertext_tmp[CELLS];
5 } plain_cipher;
```

Grâce à cette structure, il suffit de manipuler les chiffrés temporaires `ciphertext_tmp` et ensuite, lorsqu'ils deviennent inutiles, on réinitialise leurs valeurs afin de préserver celles des chiffrés d'origine `ciphertext`.

Voyons aussi quelques pseudo-algorithmes de certaines fonctions utiles. Pour la suite, nous allons définir quelques notations :

- Λ_l : le l-ème Λ-set.
- $P_{\Lambda_l,i}$: le i-ème plaintext du l-ème λ-set.
- $P[0]$: le premier octet du plaintext P.

On notera que les fonctions agissent directement sur les états, il n'y a donc aucune valeurs à renvoyer.

6. Nous appellerons ceci un état.

3.1.3.1 Génération des clairs

Algorithme 3.1.3.1. [GenPlaintexts]

Entrée : Λ la cible d'un Λ-sets, σ l'indice de l'octet actif, ϵ la valeur fixe des autres octets.

Sortie : La fonction agit directement sur le contenue du Λ-sets à travers sa cible. Elle ne renvoie donc rien.

1. *Pour i de 0 à 256 :*
2. *Pour j de 0 à 16 :*
3. $P_{\Lambda,i}[j] \longleftarrow \epsilon$
4. $C_{\Lambda,i}[j] \longleftarrow \epsilon$
5. $P_{\Lambda,i}[\sigma] \longleftarrow i$
6. $C_{\Lambda,i}[j] \longleftarrow i$

3.1.3.2 Chiffrement des clairs

Algorithme 3.1.3.2. [EncryptPlaintexts]

Entrée : Λ : le Λ-sets

Sortie : De même que pour la fonction 3.1.3.1, elle agit directement sur l'ensemble.

1. *Pour i de 0 à 256*
2. *On chiffre $C_{\Lambda,i}$* [7]
3. *On copie l'état de $C_{\Lambda,i}$ dans $Ctmp_{\Lambda,i}$*

---

7. On rappel que le chiffré C est initialisé avec la plaintext.

3.1.3.3 Inversion d'un tour

Algorithme 3.1.3.3. [InvATurn] [8]

Entrée : C : un chiffré, K : la clé à utiliser, r : le numéro du tour

Sortie : P : le clair du chiffré C

1. *Si r=5 (dernier tour)*
2. $C \longleftarrow$ `AddRoundKey`(C, K)
3. $C \longleftarrow$ `IShiftRows`(C)
4. $C \longleftarrow$ `ISubBytes`(C)
5. *Sinon si r=0*
6. $C \longleftarrow$ `AddRoundKey`(C, K)
7. *Sinon*
8. $C \longleftarrow$ `AddRoundKey`(C, K)
9. $C \longleftarrow$ `IMixColums`(C)
10. $C \longleftarrow$ `IShiftRows`(C)
11. $C \longleftarrow$ `ISubBytes`(C)
12. *Retourner C*

3.1.4 Complexité

Pour calculer la complexité, nous allons nous intéresser au nombre total de possibilités à parcourir.

On notera aussi qu'en pratique, notre algorithme interagit avec un oracle pour chiffrer les clairs. Nous ne prendrons donc pas en comptes les chiffrements.

8. En pratique, on évite de retourner un état car cela nécessite de copier de la mémoire à chaque manipulation de la fonction, on privilégie donc la manipulation de pointeurs. Mais on préférera cette forme de fonction pour rendre plus lisible les algorithmes, en l'occurrence, le suivi d'état étape par étape.

Attaque AES 5 tours de type 1

La complexité théorique est de l'ordre de 2^{42}. Effectivement, si l'on reprend notre algorithme, il s'agit de déterminer 4 octets de la clé K_5 et 1 octets de la clé K_4, cela fait :

$$\begin{aligned} 256^4 \times 256 &= 256^5 \\ &= (2^8)^5 \\ &= 2^{40} \end{aligned}$$

Or, cela permet de déterminer seulement 4 octets de la clé K_5, il faut donc répéter l'opération 4 fois ce qui nous donne :

$$2^{40} \times 2^2 = 2^{42}$$

De ce fait, pour ce nombre de tours, elle est plus efficace qu'une recherche exhaustive qui vaudrait, elle :

$$256^{16} = (2^8)^{16} = 2^{128}$$

Attaque AES 5 tours de type 2

L'attaque de type 2 quant à elle, se contente de tester seulement 4 octets de la clé K_0 à partir d'un octets d'un octet de la clé K_5, ainsi, la calcul de la complexité se résume à :

$$(2^8)^4 \times 2^8 = 2^{40}$$

De ce fait, l'attaque de type 2 est en théorie 256 fois plus rapide que celle de type 1.

3.1.5 Test des algorithmes

En pratique, il devient difficile pour un ordinateur de générer plus de 3 octets. Nous avons donc testé notre code en admettant connaître certains octet de clé afin d'éviter qu'il ne doive parcourir toutes les possibilités.

On a testé les algorithmes avec la clé :
$K = (D0, C9, E1, B6, 14, EE, 3F, 63, F9, 25, 0C, 0C, A8, 89, C8, A6)$

Algorithme de type 1

— OTF : octets à forcer par partie (il y a 4 parties qui travaillent chacune sur 5 octets).
— N-L : nombre de Λ-set utilisé.
— S/E : Succès/Échec de l'attaque.
— T : Temps de l'attaque en secondes [9].

OTF	N-L	S/E	T
2	2	S	5,67
2	3	S	8,56
2	4	S	11,39
2	5	S	14,26
3	5	S	3431,75
4	5	S	10 jours*
5	5	S	7 ans*

Algorithme de type 2

— OTF : octets à forcer (il y a 1 parties qui travaille sur 5 octets).
— N-L : nombre de Λ-set utilisé.
— S/E : Succès/Échec de l'attaque.
— T : Temps de l'attaque en secondes.

OTF	N-L	S/E	T
2	4	S	0,06
3	4	S	13,18
4	4	S	$3434 \approx 1$h
5	4	S	256h $\approx$ 10j*

On remarque, d'après les tests, que le type 2 fait aussi bien que le type 1 avec un octet en plus à forcer. De plus, à chaque fois que l'on ajoute un octet à forcer, on multiplie de temps de calcul par environ 256. Ainsi, il est naturel de se dire que l'algorithme de type 2 est 256 fois plus rapide que son prédécesseur.

Voyons donc un tableau récapitulatif :

Attaque	Clairs choisis	Chiffrements	OTF	Complexité	T
Type 1	5×2^8	5×2^8	20	2^{42}	~ 7 ans
Type 2	4×2^8	2^{42}	5	2^{40}	~ 10 jours

9. Les temps suivit d'un * sont estimés.

3.1.5.1 Remarque sur l'attaque de type 2

Comme déjà énoncé dans la partie 3.1.2.7.1, l'attaque se présente en deux parties :

— La première consiste à retrouver la diagonale de la clé K_0.
— La seconde, est de retrouver la clé K_5 à partir de la diagonale de K_0, puis de remonter K_5 en utilisant la propriété d'inversibilité de la fonction G d'extension de clé.

Intéressons nous justement à cette deuxième partie. Si celle-ci peut être représentée par l'attaque carré sur 4 tours 3.1.2.3, on en déduit alors que seuls 2 Λ-sets seront nécessaire pour trouver K_5.

Calculons donc la complexité de cette deuxième étape :

Rappelons que, pour trouver le premier octet de la clé K_5, on génère un octet k_0, puis on applique le XOR suivant pour chaque chiffré $c_{i,j}$ du Λ-set i :

$$b_0 = \sum_{j=1}^{256} ISbox(c_{0,j}[0] \oplus k_0)$$

$$b_1 = \sum_{j=1}^{256} ISbox(c_{1,j}[0] \oplus k_0)$$

telle que $c_{i,j,0}$ correspond à l'octet 0 du j-ème chiffré du i-ème Λ-set.

On génères donc $16 \times 256 = 2^{12}$ octets. C'est négligeable comparé à la complexité de l'attaque qui est en $O(2^{42})$.

On pourrait donc en déduire ceci :

Remarque 3.1.1. Pour une attaque sur un `AES` à 5 tours, déterminer la diagonale de la clé K_0 est équivalent à trouver la clé complète.

3.2 L'attaque *yoyo*

3.2.1 Théorèmes et notations

Les théorèmes et notations sont les mêmes que ceux des articles : [10] et [12].

Définition 3.2.1. *Le vecteur différence à 0* [10] *:*
Soit $\alpha \in \mathbb{F}_q^n$, le vecteur différence à 0 $\nu(\alpha) = (z_0, z_1, z_2, z_3, ..., z_{n-1})$ retourne $z_i = 1$ si la i-ème coordonnée de α est nulle et 0 sinon.
Une autre façon de l'écrire :

$$z_i \mapsto \begin{cases} 1 & si\ \alpha_i = 0 \\ 0 & sinon \end{cases}$$

Par exemple : $\nu((17, 34, 98, 0, 23, 11, 0)) = (0, 0, 0, 1, 0, 0, 1)$.

Lemme 3.2.1. *Soient α et $\beta \in \mathbb{F}_q^n$, le vecteur différence à 0 est conserver au travers du passage dans la boite S. De ce fait, on à :*

$$\nu(\alpha \oplus \beta) = \nu(S(\alpha) \oplus S(\beta))$$

Par exemple :

$$\nu((17, 34) \oplus (0, 34)) = (1, 0)$$
$$\nu(S(17, 34) \oplus S(0, 34)) = \nu((S(17), S(34)) \oplus (S(0), S(34))) = (0, 1)$$

Démonstration. S est une `S-box`, elle est donc bijective car c'est une substitution. De ce fait, si $\alpha_i \oplus \beta_i = 0$ alors, $S(\alpha_i) \oplus S(\beta_i) = 0$ d'où le résultat.
□

Définition 3.2.2. *Pour un vecteur $\omega \in \mathbb{F}_2^n$ et deux états α et $\beta \in \mathbb{F}_q^n$, on définit un nouvel état, $\rho^\omega(\alpha, \beta)$ tel que la i-ème coordonnée de ce nouvel état est :*

$$\rho^\omega(\alpha, \beta)_i = (\alpha_i \times \omega_i) \oplus (\beta_i \times (\omega_i \oplus 1))$$

Une autre façon de le voir :

$$\rho^\omega(\alpha, \beta)_i \mapsto \begin{cases} \alpha_i & si\ \nu_i = 1 \\ \beta_i & sinon \end{cases}$$

10. The zero difference pattern dans [10]

Par exemple, si : $\omega = (0, 1, 1)$, $\alpha = (11, 22, 33)$ et $\beta = (88, 77, 66)$ alors :

$$\rho^{\omega}(\alpha, \beta) = (88, 22, 33)$$

On crée de cette manière un nouvel état en échangeant des colonnes des deux états précédents.

Remarque 3.2.1. On note que $\rho^{\omega}(\alpha, \beta) \oplus \rho^{\omega}(\beta, \alpha) = \alpha \oplus \beta$.

Remarque 3.2.2. Si on pose $\overline{\omega} = (1, 1, ..., 1) \oplus \omega$ alors $\rho^{\omega}(\alpha, \beta) = \rho^{\overline{\omega}}(\beta, \alpha)$.

Décrivons quelques propriétés de la fonction ρ.

Lemme 3.2.2. *Pour un vecteur $\omega \in \mathbb{F}_2^n$ et deux états α et $\beta \in \mathbb{F}_q^n$, la fonction ρ commute avec l'application S d'une S-box.*
De ce fait, on a :

$$\rho^{\omega}(S(\alpha), S(\beta)) = S(\rho^{\omega}(\alpha, \beta))$$

De plus, en utilisant la remarque 3.2.1, on obtient :

$$S(\rho^{\omega}(\alpha, \beta)) \oplus S(\rho^{\omega}(\beta, \alpha)) = S(\alpha) \oplus S(\beta)$$

Démonstration. S opère de manière indépendante sur chaque indice d'un vecteur. D'où le résultat. □

Lemme 3.2.3. *Soit L une transformation linéaire qui agit sur des vecteurs de tailles n, alors :*
$\forall \alpha, \beta \in \mathbb{F}_q^n$ et $\forall \omega \in \mathbb{F}_2^n$ alors :

$$L(\alpha) \oplus L(\beta) = L(\rho^{\omega}(\alpha, \beta)) \oplus L(\rho^{\omega}(\beta, \alpha))$$

Démonstration. Comme L est linéaire, on a que :

$$L(\alpha) \oplus L(\beta) = L(\alpha \oplus \beta)$$

Mais, grâce à la remarque 3.2.1, on obtient :

$$L(\alpha \oplus \beta) = L(\rho^{\omega}(\alpha, \beta) \oplus \rho^{\omega}(\beta, \alpha))$$

D'où :

$$L(\alpha) \oplus L(\beta) = L(\rho^{\omega}(\alpha, \beta) \oplus \rho^{\omega}(\beta, \alpha))$$

On applique une dernière fois la linéarité de L et on obtient :

$$L(\alpha) \oplus L(\beta) = L(\rho^{\omega}(\alpha, \beta)) \oplus L(\rho^{\omega}(\beta, \alpha))$$

□

Grâce aux lemmes 3.2.1, 3.2.2 et 3.2.3, on déduis un théorème qui sert de base à l'attaque *yoyo*.

Théorème 3.2.1. *Soit S une S-`box` et L une application linéaire qui agit sur des vecteurs de tailles n.*
Soit α et $\beta \in \mathbb{F}_q^n$, on pose : $A = \rho^\omega(\alpha, \beta)$ et $B = \rho^\omega(\beta, \alpha)$.

$$\nu(S \circ L \circ S(\alpha) \oplus S \circ L \circ S(\beta)) = \nu(S \circ L \circ S(A) \oplus S \circ L \circ S(B))$$

Démonstration. Par la remarque 3.2.1, on sait que :

$$\alpha \oplus \beta = A \oplus B$$

En utilisant le lemme 3.2.1, on obtient :

$$S(\alpha) \oplus S(\beta) = S(A) \oplus S(B)$$

On applique L de chaque côté :

$$L(S(\alpha) \oplus S(\beta)) = L(S(A) \oplus S(B))$$

Par linéarité de L on a :

$$L(S(\alpha)) \oplus L(S(\beta)) = L(S(A)) \oplus L(S(B))$$

On a donc :

$$\nu(L(S(\alpha)) \oplus L(S(\beta))) = \nu(L(S(A)) \oplus L(S(B)))$$

Mais d'après le lemme 3.2.1, le vecteur différence à 0 est conserver au travers du passage dans la boite S. Ainsi :

$$\nu(S(L(S(\alpha))) \oplus S(L(S(\beta)))) = \nu(S(L(S(A))) \oplus S(L(S(B))))$$

Et donc :

$$\nu(S \circ L \circ S(\alpha) \oplus S \circ L \circ S(\beta)) = \nu(S \circ L \circ S(A) \oplus S \circ L \circ S(B))$$

□

3.2.2 Différenciateur dans AES

3.2.2.1 L'idée principale

L'attaque yoyo est une attaque différentielle. Pour l'exécuter, nous avons besoin d'observer le comportement des différentielles de deux textes au travers des

différentes fonctions de l'AES. Pour étudier ces différentielles, on introduit la notion de différenciateur [11]. Ceux-ci se basent essentiellement sur le théorème 3.2.1.

L'idée principale est de réécrire les étapes de l'AES avec des composées de fonctions de L et S. [12]Dans le cadre de l'AES, dans les énoncés de la partie 3.2.1, on a $q = 2^8$.

On rappel qu'une ronde d'AES est composé de : SubBytes (SB), ShiftRows (SR), MixColumns (MC) et AddRoundKey (AK). Pour la suite, on utiliseras les notations ci-dessus entre parenthèses.

3.2.2.2 Une représentation de l'AES particulière

Définition 3.2.3. *$S = SB \circ MC \circ SB$. S est une super S-box. Elle possède toutes les propriété d'une boite S standard.*

On peut représenter une ronde d'AES de la manière suivante :

$$AK \circ MC \circ SR \circ SB$$

De ce fait, on peut en déduire à quoi correspondent 2 rondes :

$$(AK \circ MC \circ SR \circ SB) \circ (AK \circ MC \circ SR \circ SB)$$

Pour simplifier les notations, on pose R^i les i premier tours de l'AES. Par exemple :

$$R^2 = AK \circ MC \circ SR \circ SB \circ AK \circ MC \circ SR \circ SB$$

Proposition 3.2.1. *Dans une analyse différentielle, les AddRoundKey n'interviennent pas.*

Démonstration. Soit $A = (a_0, ..., a_n)$, $B = (b_0, ..., b_n)$ et $K = (k_0, ..., k_n)$.

La différence entre A et B est $C = (a_0 \oplus b_0, ..., a_n \oplus b_n)$.

Si on fait un AddRoundKey avant on a : $A' = (a_0 \oplus k_0, ..., a_n \oplus k_n)$ et $B' = (b_0 \oplus k_0, ..., b_n \oplus k_n)$.

On écrit la différentielle :

$$C' = (a_0 \oplus \cancel{k_0} \oplus b_0 \oplus \cancel{k_0}, ..., a_n \oplus \cancel{k_n} \oplus b_n \oplus \cancel{k_n})$$

$$C' = (a_0 \oplus b_0, ..., a_n \oplus b_n) = C$$

. □

11. Noté Distinguisher dans la documentation : [10]

12. Les notations S et L sont introduites dans le théorème 3.2.1 et sont les même dans l'article [10].

D'après 3.2.1, on peut donc écrire :

$$R^2 = MC \circ SR \circ SB \circ MC \circ SR \circ SB$$

Proposition 3.2.2. *Le* `ShiftRows` *et le* `SubBytes` *commutent.*

Démonstration. Regardons ce qui se passe si on fait `SubBytes` ∘ `ShiftRows` puis `ShiftRows` ∘ `SubBytes` :

```
A B C D                 A B C D             S(A) S(B) S(C) S(D)
E F G H   ShiftRows     F G H E   SubBytes   S(F) S(G) S(H) S(E)
I J K L  ==========>    K L I J   =======>   S(K) S(L) S(I) S(J)
M N O P                 P M N O             S(P) S(M) S(N) S(O)
```

```
A B C D             S(A) S(B) S(C) S(D)                 S(A) S(B) S(C) S(D)
E F G H   SubBytes  S(E) S(F) S(G) S(H)   ShiftRows     S(F) S(G) S(H) S(E)
I J K L   =======>  S(I) S(J) S(K) S(L)   ========>     S(K) S(L) S(I) S(J)
M N O P             S(M) S(N) S(O) S(P)                 S(P) S(M) S(N) S(O)
```

On a donc : `SubBytes` ∘ `ShiftRows` = `ShiftRows` ∘ `SubBytes`. □

Avec 3.2.2, on à :

$$R^2 = MC \circ SR \circ SB \circ MC \circ SB \circ SR$$

Et donc avec 3.2.3 on a :

$$R^2 = MC \circ SR \circ \underbrace{SB \circ MC \circ SB}_{S} \circ SR$$

$$R^2 = MC \circ SR \circ S \circ SR$$

Proposition 3.2.3. *$L = SR \circ MC \circ SR$ est une application linéaire.*

Démonstration. `ShiftRows` et `MixColumns` sont des applications linéaires. La compositions de trois applications linéaires est linéaire. □

On peut donc écrire :

$$R^3 = MC \circ SR \circ SB \circ \underbrace{MC \circ SR \circ S \circ SR}_{R^2}$$

Puis par 3.2.2 on a :

$$R^3 = MC \circ SB \circ \underbrace{SR \circ MC \circ SR}_{L} \circ S \circ SR$$

$$R^3 = MC \circ SB \circ L \circ S \circ SR$$

Si on considère un `AES` à 3 tours, le dernier `MixColumns` peut être enlevé. De plus, on sais faire `ISubBytes` donc, on peut faire notre attaque sur $R^{3'}$ avec :

$$R^{3'} = L \circ S \circ SR$$

Pour peu qu'on travaille avec les `IShiftRows` des messages initiaux, on peut retirer le premier `ShiftRows` pour arrivé à une équation très simple :

$$R^{3''} = L \circ S \tag{3.1}$$

C'est avec cette équation que l'on va construire le différenciateur dans la partie 3.2.2.3.

Avec les notations précédentes, on trouve en ajoutant un tours :

$$R^4 = MC \circ SR \circ SB \circ R^3$$

$$R^4 = MC \circ SR \circ \underbrace{SB \circ MC \circ SB}_{S} \circ L \circ S \circ SR$$

$$R^4 = MC \circ SR \circ S \circ L \circ S \circ SR$$

Par un raisonnement analogue à 3.1, on construit :

$$R^{4'} = S \circ L \circ S \tag{3.2}$$

On remarque que le différenciateur pour le troisième et quatrième tours sera identique.

Définition 3.2.4. *Soit* $Q = SB \circ MC \circ SR$ *et* $Q' = SR \circ MC \circ SB$.

Pour le cinquième tours, on trouve :

$$R^5 = R^4 \circ \underbrace{SR \circ MC \circ SB}_{Q'} \circ SR$$

$$R^5 = S \circ L \circ S \circ Q' \circ SR$$

$$R^{5'} = S \circ L \circ S \circ Q' \tag{3.3}$$

3.2.2.3 Le différenciateur à 3 tours

On définit le `SimpleSwap` une fonction qui échange le premier mot de différent de chaque texte et retourne le texte échangé :

Algorithme 3.2.2.1. [SimpleSwap]

Entrée : Un couple de textes x_0 *et* x_1
Sortie : x'_0

1. $x'_0 = x_1$
2. *Pour* i *allant de* 0 *à* 3 *faire :*
3. *Si* $(x_0)_i \neq (x_1)_i$ *:*
4. $(x'_0)_i = (x_0)_i$
5. *retourner* x'_0
6. *retourner* x'_0

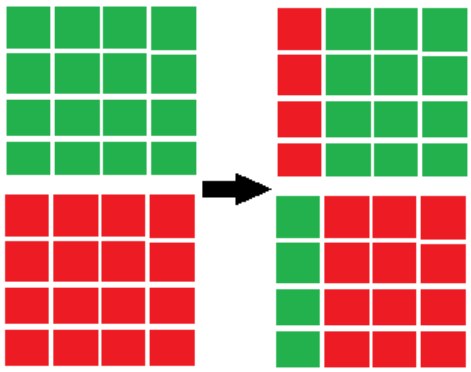


FIGURE 3.2 – Fonctionnement du `SimpleSwap`

Ici, on va se servir de l'équation 3.1 et de `SimpleSwap` 3.2.2.1 le différenciateur suivant :

Algorithme 3.2.2.2. [Différenciateur à 3 tours]

Entrée : Un couple de textes clairs p_0, p_1 *avec* $wt(\nu(p_0 \oplus p_1)) = 3$.
Sortie : 1 *si c'est un* `AES` *et* -1 *sinon.*

1. *On applique* `IShiftRows` *à* p_0 *et* p_1.
2. $c_0 = enc(p_0)$ *et* $c_1 = enc(p_1)$.
3. *On applique* `IMixColumns` *à* c_0 *et* c_1.
4. $c_2 = SimpleSwap(c_0, c_1)$ *et* $c_3 = SimpleSwap(c_1, c_0)$.
5. *On applique* `MixColumns` *à* c_2 *et* c_3.
6. $a_0 = dec(c_2)$ *et* $a_1 = dec(c_3)$
7. *On applique* `ShiftRows` *à* a_0 *et* a_1.
8. *Si* $\nu(a_0 \oplus a_1) = \nu(p_0 \oplus p_1)$:
9. *retourner* 1.
10. *retourner* -1.

Si le chiffrement et le déchiffrement est bien un `AES` à 3 tours alors, il s'écrit comme 3.1. On vérifie donc que notre algorithme de chiffrement suis bien cette équation grâce aux propriétés de ν vu dans la partie 3.2.1.

3.2.2.4 Le différenciateur à 3 tours bis

On rappelle que dans le cadre de notre analyse l'`AES` à trois tours pourrait aussi se réduire à $SB \circ MC \circ SR \circ S$. La figure ci-dessous illustre comment marche le différenciateur à trois tours.

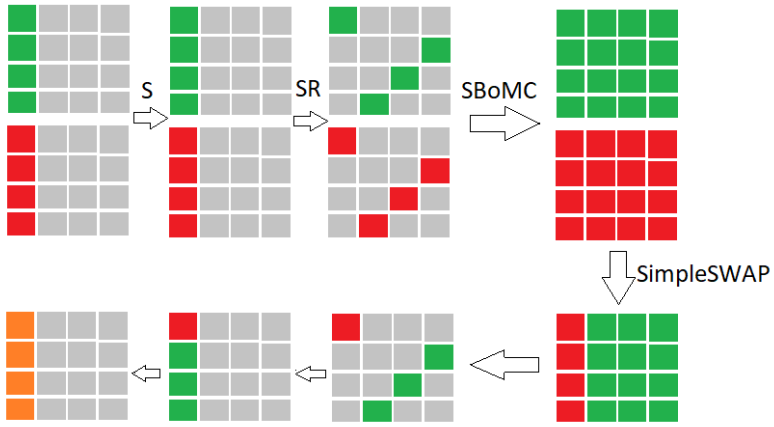


Figure 3.3 – Différenciateur à 3 tours

Étant donné de clairs p_0 et p_1 qui coïncident sur 3 parmi 4 colonnes (on suppose que la colonne "active" est la première), on les chiffrent en c_0 et c_1. A partir de ces deux chiffrés, on construit un nouveau $c_2 = SimpleSwap(c_0, c_1)$. Comme $SB \circ MC$ agit sur chaque colonne indépendemment, c'est-à-dire qu'en appliquant son inverse sur c_2 on se retrouvera avec des colonnes constituées par un octet actif et trois colonnes passifs. En appliquant un `IShiftRows`, on se retrouvera avec une colonne d'actifs et trois colonnes passifs. Pour retrouver les mêmes colonnes passifs de p_0 et p_1, on ré-applique dessus l'inverse de S.

3.2.2.5 Le différenciateur à 5 tours

On rappelle que l'on a préalablement démontré que cinq tour d'`AES` se réduisent à $S \circ L \circ S \circ Q'$. Donc, d'après le théorème 3.2.1, on remarque qu'il suffit de suivre la propagation de la différentielle d'une pair de clairs après Q'. Ainsi, l'idée principale de ce différenciateur repose sur le fait que si la différentielle de deux clairs-après avoir effectué un tour d'`AES` (essentiellement après Q')-est nulle sur t parmi 4 mots, alors la différentielle de la nouvelle pair de clairs créée en appliquant "yoyo" serait nulle sur exactement les mêmes mots (après bien évidemment le premier tour).

3.2.3 Le principe de l'attaque

L'attaque yoyo est une attaque `CPA` (à clair choisi) déterministe. De plus, c'est une attaque total breakdown. En effet, l'objectif de l'attaque est de retrouver l'intégralité de la clé. Pour cela, on va suivre une démarche similaire à celle de l'attaque square 3.1.2.7. En effet, on va chercher en premier lieu à retrouver la diagonale de K_0 puis en déduire K_5 pour enfin remonter jusqu'à K avec G^{-1} (voir : 3.1.5.1). Il est à préciser que tout au long de cette partie, nous ferons fi du premier `ShiftRows` et des deux derniers `ShiftRows` et `MixColumns` pour être cohérent avec la partie théorique présentée préalablement.

Ce qu'on va attaquer est bien l'`AES` à 5-tours. On rappelle que ce chiffrement dans notre cas de figure se réduit à $S \circ L \circ S \circ Q'$. De ce fait, il faut caractériser le vecteur différence à 0 de la différentielle de chaque pair de clair choisie juste après avoir appliqué Q'. Or Q' est la composée de $SR \circ MC \circ SB$. Donc, on peut restreindre notre caractérisation du vecteur différence à 0 juste avant l'application de SR (étant donnée que SR ne fait qu'une simple permutation circulaire fixée des octets de l'entrée).

La fonction $MC \circ SB$ agit sur chaque colonne indépendemment. De ce fait, prenons deux textes clairs p_0 et p_1 dont la première colonne est la seule colonne "active" (différente). Soient $(0, i, 0, 0)$ et $(z, z \oplus i, 0, 0)$ tels que $(z \in (\mathbb{F}_{256})$ et $z \neq 0)$ les premières colonnes de p_0 et p_1 respectivement. Ainsi, on pose : $k = (k_0, k_1, k_2, k_3)$ la première colonne de la clé. Et on rajoute l'étape d'initialisation avec la clef K_0 à notre chiffrement AK.

Regardons maintenant l'évolution de la différentielle de ces deux premières colonnes de p_0 et p_1 en sortie de $MC \circ SB \circ AK$. On obtient les équations suivantes :

$$\begin{cases} 2b_0 \oplus 3b_1 = y_0 \\ b_0 \oplus 2b_1 = y_1 \\ b_0 \oplus b_1 = y_2 \\ 3b_0 \oplus b_1 = y_3 \end{cases} \tag{3.4}$$

Avec :

$$\begin{cases} b_0 = s(k_0) \oplus s(z \oplus k_0) \\ b_1 = s(k_1 \oplus z \oplus i) \oplus s(i \oplus k_1) \end{cases} \tag{3.5}$$

On va s'intéresser plus particulièrement à la troisième équation qui constitue précisément le troisième octet relatif à la première colonne de la différentielle des deux clairs p_0 et p_1 après l'application de $MC \circ SB \circ AK$:

$$s(k_0) \oplus s(z \oplus k_0) \oplus s(k_1 \oplus z \oplus i) \oplus s(i \oplus k_1) = y_2$$

On remarque que y_2 est nulle lorsque $i = k_0 \oplus k_1$ ou $i = k_0 \oplus k_1 \oplus z$. De ce fait, on peut faire l'observation suivante qui sera utile à notre attaque : lorsque l'on parcourt toutes les valeurs possibles de i allant de 0 jusqu'à 255, on tombera nécessairement sur la bonne valeur de $k_0 \oplus k_1$ pour laquelle y_2 s'annule. Autrement dit, on a pu trouver une relation linéaire entre le premier et le deuxième octet de la clef. Donc, il suffit de faire une recherche exhaustive sur trois octets de la clef pour en trouver quatre. Mais, nous sommes en train d'exploiter une propriété relative à une valeur intermédiaire (après un tour de chiffrement) à laquelle on n'a pas accès. Et c'est ici où réside la subtilité de l'attaque *yoyo* !

Soient p_0 et p_1 deux clairs construits comme indiqué ci-dessus. On suppose de plus que ces deux clairs sont construits avec la bonne valeur de $i = k_0 \oplus k_1$. Soit $x_j = MC \circ SB \circ AK(p_j)$. Ainsi, le troisième octet de la première colonne de la différentielle entre x_0 et x_1 est nul. Autrement dit, le vecteur différence à 0 des deux premières colonnes est égal à $(0, 0, 1, 0)$. Or, si on définit deux nouveaux états (comme expliqué dans la première partie de ce chapitre) à partir de x_0 et x_1, on verra la conservation du vecteur différence à 0 au travers $S \circ L \circ S$. Donc, pour créer de nouveaux états compatibles avec notre analyse, on utilise notre fonction `SimpleSwap` (3.2.2.1) pour générer de nouveaux chiffrés à partir des chiffrés de p_0 et p_1. Puis, on les déchiffre et on ré-applique `SimpleSwap` sur les nouveaux clairs. On fait une supposition sur une colonne de la clef que l'on additionne bit à bit avec nos nouveaux clairs, on applique dessus $MC \circ SB$ et

on examine le vecteur différence à 0 relatif aux colonnes "actives". S'il est égal à $(0, 0, 1, 0)$, on détient possiblement les bons octets de la clef. Ainsi, il faut construire plusieurs pairs de clairs pour s'assurer de l'exactitude de nos hypothèses sur la clef.

En suivant le raisonnement ci-dessus, on doit retrouver les octets de la clef colonne par colonne. Or, pour suivre le raisonnement théorique de la première partie, on a dû omettre le première `ShiftRows` de notre `AES`. Donc, concrètement on fait des suppositions sur des colonnes inversement "shiftées" (après avoir appliqué `IShiftRows`). Donc, pour notre implémentation de l'attaque, on va attaquer d'abord les octets $0, 10$ et 15 pour retrouver l'octet 5. Une fois qu'on a la diagonale, on retrouve la clé (voir : 3.1.5.1).

3.2.4 Pseudo-code

Ici, nous utiliserons les fonctions *Encryption* et *Decryption* comme les chiffrement/déchiffrement `AES` sans leur premier et dernier `ShiftRows` /`IShiftRows`

Algorithme 3.2.4.1. [Attaque Yoyo]

Entré : $\mathcal{P}_0$, $\mathcal{P}_1$ [13]
Sortie : $D(K_0)$, *la diagonal de la clé* K_0

1. *Pour* i *de* 0 *à* 255 *:*
2. *On définit* S *une liste de couple*
3. *Pour* j *de* 0 *à* 4 *:*
4. $p_0 \longleftarrow \mathcal{P}_0[i], p_1 \longleftarrow \mathcal{P}_1[i]$
5. $c_1 \longleftarrow Encryption(p_0), c_0 \longleftarrow Encryption(p_1)$
6. ${c_1}' \longleftarrow SimpleSwap(c_1, c_0), {c_1}' \longleftarrow SimpleSwap(c_1, c_1)$
7. ${p_0}' \longleftarrow Decryption({c_1}'), {p_1}' \longleftarrow Decryption({c_1}')$
8. $p_0 \longleftarrow SimpleSwap({p_0}', {p_1}'), p_1 \longleftarrow SimpleSwap({p_1}', {p_0}')$
9. *On ajoute le couple* (p_0, p_1) *à* S
10. *On définit* K *une clé*
11. *Pour* $k0$ *de* 0 *à* 255
12. *Pour* $k2$ *de* 0 *à* 255
13. *Pour* $k3$ *de* 0 *à* 255
14. $K[0] = k0$, $K[5] = (k0 \oplus i)$, $K[10] = k2$, $K[15] = k3$
15. *Pour tous les couples* (p_0, p_1) *de* S
16. $K' \longleftarrow$ `ShiftRows`(K)
17. ${p_0}' \longleftarrow$ `AddRoundKey`$(p_0, K'), {p_1}' \longleftarrow$ `AddRoundKey`(p_1, K')
18. ${p_0}' \longleftarrow$ `SubBytes`$({p_0}'), {p_1}' \longleftarrow$ `SubBytes`$({p_1}')$
19. ${p_0}' \longleftarrow$ `MixColumns`$({p_0}'), {p_1}' \longleftarrow$ `MixColumns`$({p_0}')$
20. *Si* ${p_0}' \oplus {p_1}' = 0$
21. **On a trouvé la diagonal de* $K0$ ***
22. *On retourne* K

---

13. Définit un peu après à la partie 3.2.5.1.

3.2.5 Implémentation

Avant de parler de l'implémentation de l'algorithme, nous allons aborder quelques fonctions nécessaires.

Comme dit plus haut, l'attaque *yoyo* est une attaque différentielle. On va donc commencer par générer des clairs bien choisis.

3.2.5.1 La création des ensembles

Pour faire fonctionner l'attaque nous créons 2 ensembles de 256 états que nous appellerons $\mathcal{P}_0$ et $\mathcal{P}_1$. Prenons $\mathcal{P}_0[i] := p_0$ ($\mathcal{P}_1[i] := p_1$) le i-ème élément de $\mathcal{P}_0$ ($\mathcal{P}_1$ respectivement) et p_0^0 (p_1^0) la première colonne de l'état p_0 (p_1 respectivement).

Nous savons que l'attaque se base sur la manipulation et la différences de la première colonne des p_0 et p_1. La structure des états de ces ensembles correspondent à :

$$p_0^0 = (0, i, 0, 0), p_1^0 = (1, 1 \oplus i, 0, 0), \forall i \in [0, 255]$$

C'est à dire :

```
p0              p1
0  0  0  0      1  0  0  0
i  0  0  0    1+i  0  0  0
0  0  0  0      0  0  0  0
0  0  0  0      0  0  0  0
```

Algorithme 3.2.5.1. [GenPlaintexts-yoyo]

Entré : Les cibles de $\mathcal{P}_0$ et $\mathcal{P}_1$

1. *Pour i de 0 à 255 :*
2. *Pour j de 0 à 15 :*
3. $\mathcal{P}_0[i][j] \longleftarrow 0$, $\mathcal{P}_1[i][j] \longleftarrow 0$
4. $\mathcal{P}_1[i][0] \longleftarrow 1$
5. $\mathcal{P}_0[i][4] \longleftarrow i$, $\mathcal{P}_0[i][4] \longleftarrow i \oplus 1$

3.2.5.2 Le chiffrement/déchiffrement variant

On a aussi vu dans la partie théorique que les chiffrements avec lesquels on travail n'ont pas de `ShiftRows` dans le premier, et le dernier tour. On va donc simplifier l'attaque en les redéfinissant.

Algorithme 3.2.5.2. [EncryptionExp]

Entré : P un clair
Sortie : C le chiffré de P

1. $P' \longleftarrow IShiftRows(P)$
2. $C \longleftarrow Encryption(P)$
3. $C \longleftarrow IShiftRows(C')$
4. *Retourner C*

Algorithme 3.2.5.3. [DecryptionExp]

Entré : C un chiffré
Sortie : P le clair de C

1. $C' \longleftarrow ShiftRows(C)$
2. $P \longleftarrow Decryption(C')$
3. $P \longleftarrow ShiftRows(P)$
4. *Retourner P*

3.2.5.3 Échanges des colonnes

Comme expliqué plus haut, dans l'attaque, nous aurons besoins de la fonction `SimpleSwap`. Rappelons que cette fonction, en prenant deux états en entrée, inverse la première colonne différente, et renvoie le premier état modifié.

Par exemple, avec 2 états sous forme de 4 colonnes pour obtenir le `SimpleSwap` de $S0$:

```
S0 :              S1 :

 A  B  C  D        A' B' C' D'      = SimpleSwap(S0,S1) =>   A' B  C  D
```

Or, dans l'attaque, il faut appliquer deux fois la fonction pour obtenir aussi le `SimpleSwap` de $S1$. Ainsi, nous allons créer une fonction permettant de calculer directement les deux états.

Algorithme 3.2.5.4. [SimpleSwapCol]

Entré : S_0, S_1

Sortie : $Swap_0$, $Swap_1$

1. $S_0 \longleftarrow Swap_0$, $S_1 \longleftarrow Swap_1$
2. *Pour col de* 0 *à* 3 *:*
3. *Si* $S_0^{col} \neq S_1^{col}$
4. $Swap_0^{col} \longleftarrow S_1^{col}$
5. $Swap_1^{col} \longleftarrow S_0^{col}$
6. *On retourne* $Swap_0$ *et* $Swap_1$

3.2.5.4 Attaque

Maintenant que nous avons définit certaines fonctions, voyons l'algorithme de l'attaque :

Algorithme 3.2.5.5. [Attaque type 1]

Entré : $\mathcal{P}_0$, $\mathcal{P}_1$

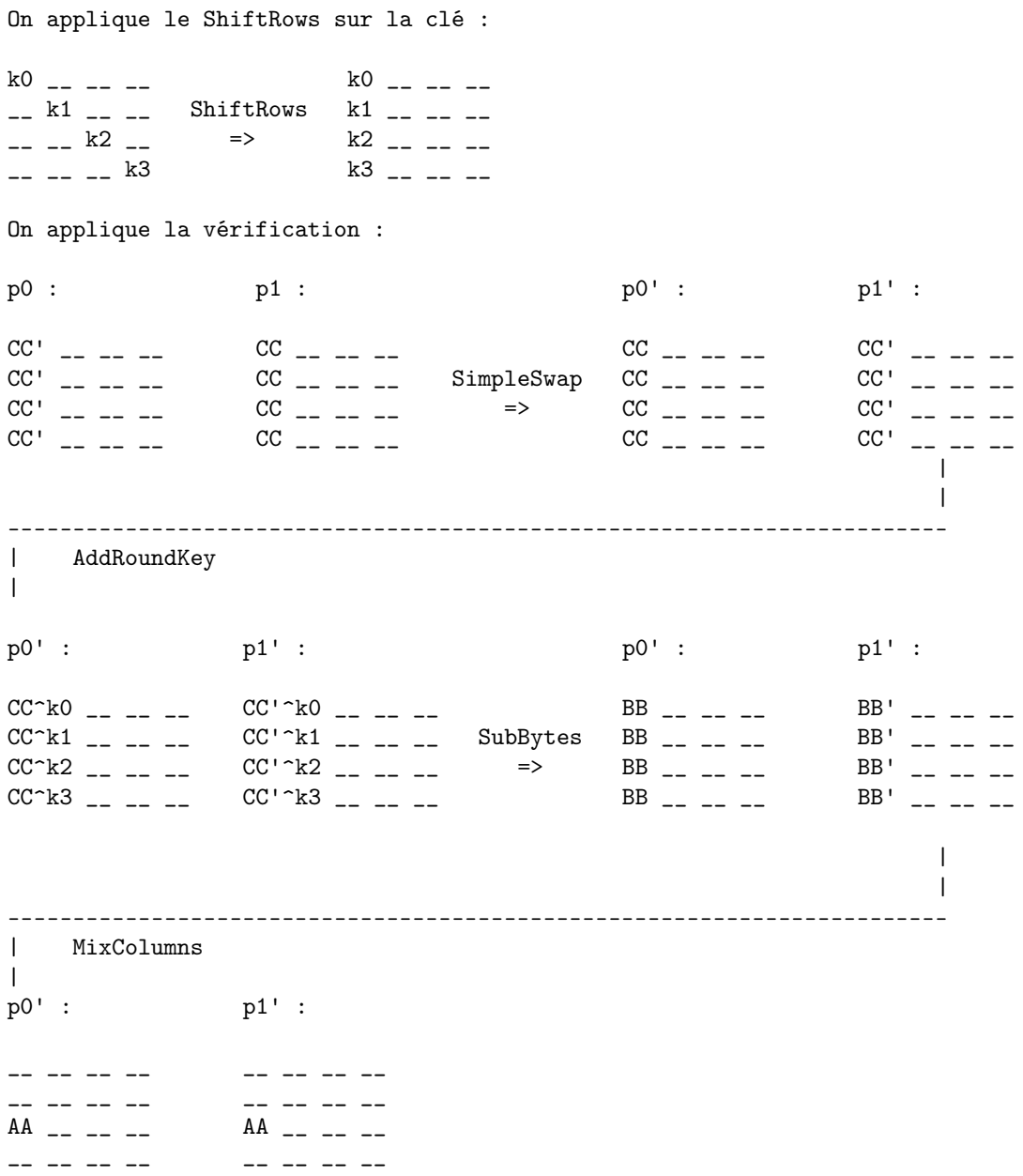
Sortie : $D(K_0)$, *la diagonal de la clé* K_0

1. *Pour* i *de* 0 *à* 255 *:*
2. *On définit* S *une liste de couple*
3. *Pour* j *de* 0 *à* 4 *:*
4. $p_0 \longleftarrow \mathcal{P}_0[i], p_1 \longleftarrow \mathcal{P}_1[i]$
5. $c_0 \longleftarrow \texttt{EncryptionExp}(p_0), c_1 \longleftarrow \texttt{EncryptionExp}(p_1)$
6. ${c_0}', {c_1}' \longleftarrow \texttt{SimpleSwapCol}(c_0, c_1)$
7. ${p_0}' \longleftarrow \texttt{DecryptionExp}({c_1}'), {p_1}' \longleftarrow \texttt{DecryptionExp}({c_0}')$
8. $p_0, p_1 \longleftarrow \texttt{SimpleSwapCol}({p_0}', {p_1}')$
9. *On ajoute le couple* (p_0, p_1) *à* S
10. *On définit* K *une clé*
11. *Pour* k_0 *de* 0 *à* 255 *:*
12. *Pour* k_2 *de* 0 *à* 255 *:*
13. *Pour* k_3 *de* 0 *à* 255 *:*
14. $K[0] = k_0$, $K[5] = (k_0 \oplus i)$, $K[10] = k_2$, $K[15] = k_3$
15. *Pour tous les couples* (p_0, p_1) *de* S
16. $K' \longleftarrow \texttt{ShiftRows}(K)$
17. ${p_0}' \longleftarrow \texttt{AddRoundKey}(p_0, K'), {p_1}' \longleftarrow \texttt{AddRoundKey}(p_1, K')$
18. ${p_0}' \longleftarrow \texttt{SubBytes}({p_0}'), {p_1}' \longleftarrow \texttt{SubBytes}({p_1}')$
19. ${p_0}' \longleftarrow \texttt{MixColumns}({p_0}'), {p_1}' \longleftarrow \texttt{MixColumns}({p_0}')$
20. *Si* ${p_0}'[8] + {p_1}'[8] = 0$
21. **On a trouvé la diagonal de* K_0 ***
22. *On retourne* K

Après observation, on peut séparer cet algorithme en deux parties :
— La création de la liste S de la ligne 2 à 9
— Et la plus grosse, la génération de la clé.

Dans cette deuxième partie, on va appeler *La vérification* les étapes de la ligne 16 à 20.

Voici donc un schéma représentant l'étape de la vérification (On suppose dans ce cas que i est correct et que les premières colonnes de $p0$ et $p1$ sont différentes) :



3.2.5.5 Optimisation de la vérification

On remarque que lors de la vérification, seul le 3-ème octet de la première colonne nous intéresse. Ainsi, le calcul sur les autres valeurs sont inutiles.

Avant de vérifier l'égalité des deux valeurs, on applique un `MixColumns` sur les deux états, or, on sais que le `MixColumns` correspond à un produit [14] matriciel. On peut donc se restreindre à calculer uniquement la valeur que nous souhaitons.

Posons le produit appliqué par `MixColumns` suivant :

`MixColumns` :

```
alpha              state               result
02 03 01 01        AA BB CC DD         AA' BB' CC' DD'
01 02 03 01   x    EE FF GG HH    =    EE' FF' GG' HH'
01 01 02 03        II JJ KK LL         II' JJ' KK' LL'
03 01 01 02        MM NN OO PP         MM' NN' OO' PP'
```

On a dit précédemment que la valeur que l'on souhaite calculer est le 3-ème octet de la première colonne du produit, soit `II'` dans notre cas. Nous aurons donc uniquement besoins de la troisième ligne de `alpha` et de la première colonne de `state`.

On a donc

$$\texttt{II'} = \texttt{AA} \oplus \texttt{EE} \oplus 2 \times \texttt{II} \oplus 3 \times \texttt{MM}$$

Aussi, juste avant le `MixColumns` , on applique un `SubBytes` sur toutes les valeurs de l'état, on peut donc aussi réduire ceci uniquement sur la première colonne de l'état. On arrive au produit vectoriel suivant :

```
                   Sbox[AA]
                   Sbox[EE]
                   Sbox[II]
(01 01 02 03)  x   Sbox[JJ]
```

Pour finir, on peut appliquer un XOR avec la clé directement sur la première colonne :

```
                   Sbox[AA + k0]
                   Sbox[EE + k1]
                   Sbox[II + k2]
(01 01 02 03)  x   Sbox[JJ + k3]
```

14. On rappel qu'il s'agit du produit dans le corps $\mathbb{F}_{256}$

Ainsi, au lieu de parcourir nos états 3 fois, on applique seulement :
— 4 XOR pour le `AddRoundKey` contre 16
— 4 `Sbox` pour le `SubBytes` contre 16
— 3 XOR et 2 produits pour le `MixColumns` contre 3×16 XOR et 2×16 produits.

En sachant que cette vérification est exécutée dans 3 boucles imbriquées, le gain n'est pas négligeable.

On peut maintenant proposer un algorithme qui prend en entré un état et une clé, et retourne le 3-ème octet de la première colonne après y avoir appliqué : `MixColumns` $\circ$ `SubBytes` $\circ$ `AddRoundKey` sur l'état.

Algorithme 3.2.5.6. [ComputeVerif]

Entré : S, K

Sortie : $(\texttt{MixColumns} \circ \texttt{SubBytes} \circ \texttt{AddRoundKey}(S, K))[8]$

1. $a \longleftarrow \texttt{Sbox}(S[0] \oplus K[0])$
2. $b \longleftarrow \texttt{Sbox}(S[4] \oplus K[4])$
3. $c \longleftarrow \texttt{Sbox}(S[8] \oplus K[8])$
4. $d \longleftarrow \texttt{Sbox}(S[12] \oplus K[12])$
5. *On retourne* $a \oplus b \oplus 2 \times c \oplus 3 \times d$

On en déduit le changement suivant :

Algorithme 3.2.5.7. [Modification 1]

14. $K[0] = k0$, $K[5] = (k0 \oplus i)$, $K[10] = k2$, $K[15] = k3$
15. *Pour tous les couples* (p_0, p_1) *de* S *:*
16. $\quad K' \longleftarrow \texttt{ShiftRows}(K)$
17. $\quad$ *Si* $\texttt{ComputeVerif}(p_0, K') = \texttt{ComputeVerif}(p_1, K')$ *:*
18. $\quad\quad$ **On a trouvé la diagonal de* K_0 ***
19. $\quad\quad$ *On retourne* K

Comme dit précédemment, il est intéressant de supprimer des opérations inutiles dans les boucles imbriqué.

En observant l'algorithme, on se rend compte que l'on pourrait retirer le `ShiftRows` de la clé K à condition d'allouer les valeurs sur les colonnes. Ce qui nous donne la modification de algorithme suivante :

Algorithme 3.2.5.8. [MODIFICATION 2]

14. $K[0] = k0$, $K[4] = (k0 \oplus i)$, $K[8] = k2$, $K[12] = k3$
15. *Pour tous les couples* (p_0, p_1) *de* S *:*
16. *Si* `ComputeVerif`$(p_0, K) =$ `ComputeVerif`(p_1, K)
17. **On a trouvé la diagonal de* K_0 ***
18. *On retourne* K

3.2.6 Complexité

Si on considère la même unité que dans la partie sur l'attaque Square, on va dénombrer les octets générés. Ainsi, dans cette attaque, on génère k_0, k_2 et k_3 i fois. On en déduit donc ce produit :

$$(2^8)^4 = 2^{32}$$

3.2.7 Test des algorithmes

Pour tester notre implémentation, nous utiliserons la même notation que dans la partie sur l'attaque square.

De même, pour l'instant, nous testerons l'algorithme avec la clé :
$K = (D0, C9, E1, B6, 14, EE, 3F, 63, F9, 25, 0C, 0C, A8, 89, C8, A6)$

- OTF : Octets à forcer (il y a 1 parties qui travaille sur 3 octets).
- S/E : Succès/Échec de l'attaque.
- T : Temps de l'attaque en secondes.

OTF	S/E	T
1	S	0,00s
2	S	0,08s
3	S	20,12s

On remarque que les temps sont très petits, on va donc pouvoir un peu plus exploiter l'exécution.

Pour vérifier l'efficacité et le temps d'exécution de notre implémentation, nous avons créée un fichier python permettant d'exécuter l'attaque sur une clé aléatoire, de mesurer le temps d'une exécution et de détecter un échec, et ce, un certain nombre de fois.

Pour que le temps de la mesure reste suffisamment raisonnable, nous effectuerons 100 exécutions sur un clé aléatoire.

On obtient le tableau suivant :

Attaque	Nbr de clés	Nbr de couples $(p0, p1)$	Succès (%)	Temps moyen
Type 1	100	5	100%	24.16s

Avant de pouvoir comparer nos 3 attaques, nous allons compter :

— Les clairs choisis : Dans cette méthode, on utilise seulement 2 ensembles de 256 clairs. On est donc à $2^8 \times 2 = 2^{10}$

— les Chiffrements : Ici, il s'agit évidemment du `EncryptExp`, que l'on applique autant de fois qu'il y a d'éléments dans S, soit 5×2 et cela, 2^8, donc $2^8 \times 5 \times 2 = 2^{11.32}$

Nous pouvons donc présenter le tableau récapitulatif des deux attaques :

Attaque	Clairs choisis	Chiffrements	OTF	Complexité	T
Square Type 1	$2^{10.32}$	$2^{10.32}$	20	2^{42}	~ 7 ans
Square Type 2	2^{10}	2^{42}	5	2^{40}	~ 10 jours
Yoyo	2^{10}	$2^{11.32}$	3	2^{32}	24.16 secondes

Chapitre 4

Conclusion

Depuis 20 ans que `AES` est en service, la cryptanalyse de ce système de chiffrement a beaucoup évolué.

En effet, la première attaque connue est l'attaque exhaustive. Celle-ci se fait en temps constant quelque soit le nombre de tours d'`AES` qu'on choisit d'attaquer.

Ensuite, avec le papier qui présente l'`AES` on parle de l'attaque `Square` qui rend l'`AES` à 4 tours inutilisable. Cependant, sur 5 tours, il faudrait encore un peu plus de 10 jours pour retrouver une clé ce qui est long mais raisonnable.

Ensuite vient l'attaque `yoyo`. Elle ne nécessite que quelques secondes (moins d'une minute) pour retrouver la clé utilisée pour chiffrer.

De ce fait, ce modèle réduit de l'`AES` est inutilisable en pratique pour chiffrer des données de manière efficaces.

D'autres attaques proposent d'attaque le modèle à 5 tours comme celle présenter dans le papier [1] et qui commencent doucement à s'adapter pour un nombre de tours supérieur.

L'avantage de l'attaque `yoyo` c'est qu'elle est envisageable jusqu'à un nombre de tour élevé comme le montre le document [12] ce qui lui donne la possibilité d'être utilisé pour un plus gros `AES`.

Cependant l'`AES` classique à 12 tours reste sécurisé mais ses versions avec moins de tours ne le sont plus. Faudrait-il envisager d'augmenter le nombre de tours pour faire face aux futures découvertes et à l'augmentation de la puissance de calcul des ordinateurs ?

Bibliographie

[1] Orr Dunkelman, Nathan Keller, Eyal Ronen et Adi Shamir, *The Retracing Boomerang Attack*.
https://eprint.iacr.org/2019/1154.pdf.

[2] Jean-Paul Cerri, *RIJNDAEL – AES*.
https://www.math.u-bordeaux.fr/ jcerri/cryptologie18/rijndael.pdf

[3] Kit Choy Xintong, *Understanding AES Mix-Columns Transformation Calculation*.
https://www.angelfire.com/biz7/atleast/mix_columns.pdf

[4] Article de securiteinfo L'AES par SoGoodToBe, *Advanced Encryption Standard*.
https://www.securiteinfo.com/cryptographie/aes.shtml

[5] Wikipédia, *Le chiffrement de Rijndael*.
https://fr.wikipedia.org/wiki/Rijndael

[6] Acrypta, *Fonctionnement d'AES*.
http://www.acrypta.com/telechargements/fgc/annexes/fgc_annexe_1.pdf

[7] Joan Daemen, Lars Knudsen et Vincent Rijmen, *The block cipher Square*.
https://link.springer.com/chapter/10.1007/BFb0052343

[8] David Wong, *Attacking 4 rounds with the Square attack*.
https://www.davidwong.fr/blockbreakers/square_2_attack4rounds.html

[9] Vincent Rijmen et Joan Daemen, *Rijndael Ammended*.
https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf

[10] Sondre Rønjom, Navid Ghaedi Bardeh et Tor Helleseth, *yoyo Tricks with AES*.
https://eprint.iacr.org/2017/980.pdf

[11] J.M. Dutertre, *Synthèse AES 128*.
https://www.emse.fr/ dutertre/documents/synth_AES128.pdf

[12] Dhiman Saha, Mostafizar Rahman et Goutam Paul, *New yoyo Tricks with AES-based Permutations* in IACR Transactions on Symmetric Cryptology ISSN 2519-173X, Vol. 2018, No. 4, pp. 102–127.
https://doi.org/10.13154/tosc.v2018.i4.102-127