AES in a (white)box

Sageloli Eric, Wafo-Tapa Guillaume
Sous la direction du maitre de conférence Emmanuel Fleury

9 octobre 2018

Table des matières

Conventions

— Nous utiliserons les mots nibble, octet (ou byte) et mot (ou word) pour désigner les tailles de variables (4, 8 et 32 bits respectivement).
— Généralement, nous commencerons par numéroter toute suite finie par 0. Notamment, le premier tour de l'algorithme AES sera numéroté par 0 dans nos pseudo-codes.
— La concaténation de deux mots a et b se notera $a \,||\, b$.
— Nous appellerons bloc une suite de 16 bytes. Un bloc pourra être vu comme une suite d'éléments

$$\texttt{state} = (x_0, \cdots, x_{15})$$

ou bien comme un tableau $4 * 4$ de la façon suivante :

$$\texttt{state} = \begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

Ainsi, par « la deuxième colonne de `state` », nous voulons dire les éléments (x_4, x_5, x_6, x_7).

Précisons quelques notations relatives au pseudo-code utilisé :
— Pour signifier l'élément e $(0 \leq e \leq 15)$ de `state` , nous noterons `state[e]` ou bien `state[elt e]` ;
— Pour signifier la colonne c $(0 \leq c \leq 3)$ de `state` , nous noterons `state[col c]`.
— Pour signifier la ligne r $(0 \leq r \leq 3)$ de `state` , nous noterons `state[row r]`.
— Il y a donc une égalité entre `state[col c, row r]` et `state[elt 4*c + r]`.

Certains éléments, comme par exemple la clé étendue `key` du code AES 128, peuvent être vus comme une suite de blocs (`key` en contient 11). Dans ces cas là, on pourra noter `key[bloc b]` pour se référer au b^{e} bloc ($0 \leq b \leq 10$ dans le cas de `key`).

On peut utiliser conjointement ces notations. Par exemple : `key[bloc b, elt e]` signifiera l'élément e du bloc b.

1 Introduction

Un contenu digital est, par essence, copiable à volonté sans perte de qualité. De fait, quiconque souhaite commercialiser un tel contenu et empêcher ses utilisateurs de se l'approprier pour le redistribuer se heurte à un problème évident. Le Digital Rights Management (DRM) essaie d'apporter une solution ; c'est un ensemble de technologies permettant de contrôler et/ou restreindre l'accès à un contenu propriétaire ou protégé par des droits d'auteurs. Cette mission est d'autant plus délicate lorsque le contenu en question s'exécute dans un environnement non contrôlé. C'est le cas par exemple d'un contenu multimédia chiffré qui est déchiffré au visionnage sur la machine de l'utilisateur. Dans un tel contexte, l'utilisateur, qui peut s'avérer malveillant, peut analyser le programme de déchiffrement comme il le souhaite pour tenter d'en trouver la clé. Ce contexte, dit whitebox, diffère du contexte black-box dans lequel l'attaquant ne peut que soumettre des chiffrés et obtenir les clairs correspondants.

Dans le précédent exemple, puisque le logiciel de déchiffrement tourne sur le système client, la clé ne doit en aucun cas apparaître dans l'algorithme. Celui-ci doit donc être obscurci pour la dissimuler.

C'est là l'enjeu de la cryptographie whitebox dont nous allons traiter ici. Plus précisément nous allons parler de l'implémentation en cryptographie whitebox de l'algorithme de chiffrement AES avec clé de 128 bits proposée par Chow et al. dans [9]. Ainsi dans un premier temps nous reviendrons brièvement sur le concept d'obscurcissement. Ensuite nous détaillerons la construction du whitebox AES en partant de l'algorithme AES classique. Pour cela, nous décrirons une à une les différentes couches de protection qui le composent :

— les tables de correspondance ;
— les encodages internes ;
— les mixing bijections ;
— les encodages externes.

L'ajout de chacune sera motivé par la donnée d'une attaque.

Enfin, ce rapport va de pair avec une réalisation en C de cet algorithme ainsi que de certaines des attaques. Nous clôturerons donc ce rapport avec un survol de cette implémentation.

2 Notions d'obscurcissement

2.1 Obscurcisseur universel

Comme mentionné en introduction, AES doit être modifié pour s'exécuter dans un contexte whitebox. Nous allons donc avoir recours à des techniques d'obscurcissement. Si cette dernière notion est intuitive, elle n'est cependant pas simple à formaliser.

Il en va de même pour la notion d'obscurcisseur. De manière informelle, c'est un compilateur qui transforme « efficacement » un programme en un nouveau programme ayant les mêmes fonctionnalités mais dont le code est en un sens « illisible ». Par « illisible », la plupart des interprétations entendent qu'un obscurcisseur est une blackbox virtuelle, à savoir que toute information pouvant être extraite à partir du programme obscurci peut être obtenue à partir d'un accès oracle au programme original. Barak et al. ont montré dans [2] qu'un obscurcisseur générique n'existe pas. Plus exactement :

Théorème 1. *(Théorème d'impossibilité)*

Pour tout compilateur , il existe une famille $\mathcal{P}$ de programmes inobscurcissables. C'est-à-dire telle que :

- *A partir de tout programme P' ayant les mêmes fonctionnalités que $P \in \mathcal{P}$, on peut reconstruire le code source de P efficacement ;*
- *Aucun algorithme efficace ne peut reconstruire $P \in \mathcal{P}$ en utilisant un accès oracle à ce dernier (ou avec une probabilité négligeable).*

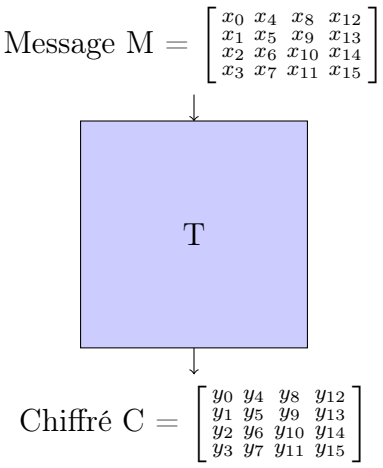
Ainsi, aucun compilateur ne peut être un obscurcisseur générique mais le théorème n'exclut pas l'existence d'un obscurcisseur pour certaines classes de programmes. En particulier on ne peut, à l'heure actuelle, affirmer ou infirmer l'existence d'un obscurcisseur pour les algorithmes de chiffrement par blocs dans le contexte whitebox. Dans notre cas, nous n'allons de toute façon pas chercher à totalement obscurcir le programme mais simplement à dissimuler la clé utilisée dans l'algorithme initial.

2.2 Obscurcisseur AES

Nous allons obscurcir l'algorithme AES en suivant l'implémentation de Chow et al. Plus précisément, nous allons écrire un programme P qui pour une clé k de 128 bits donnée, produit une version obscurcie de l'algorithme E_k de chiffrement AES ; c'est à dire, dans laquelle la clé k, apparaîssant originellement en clair, est dissimulée.

Idéalement, le programme obscurci pourrait utiliser une table de correspondance associant à chacun des 2^{128} messages possibles son chiffrement AES (voir figure suivante).

La clé n'apparaîtrait alors pas dans l'algorithme et on ne pourrait pas avoir plus d'informations avec le code qu'en observant uniquement ses entrées et sorties.



Malheureusement les capacités de stockage requises pour une telle table sont en pratique inatteignables. En effet, un tel tableau aurait 2^{128} entrées et chaque sortie serait écrite sur 16 octets ; il faudrait donc le stocker sur 2^{132} octets. On voit donc que cet algorithme n'est absolument pas calculé « efficacement » par rapport à l'algorithme AES standard.

On va donc, plutôt qu'une grande table de correspondance, utiliser plusieurs tables de tailles raisonnables. On voit que par exemple, si on arrivait à découper l'algorithme en 16 tables comme ici :

il ne faudrait alors que $16 \times 2^8 = 4$Ko.

En réalité, le découpage est plus compliqué et nécessite plus de tables. Toutefois, la taille du programme reste raisonnable : il faut théoriquement 2032 tables demandant environ 500Ko. En pratique, l'AES généré par notre obfuscateur a une taille moyenne de 770Ko.

3 De l'algorithme AES classique vers une implémentation avec tables de correspondance

3.1 Des tours vers les étapes

Nous allons donc chercher à découper l'algorithme AES en plusieurs tables ; à la fin, nous utiliserons uniquement des tables de 256 entrées, ce qui implique la définition suivante :

Définition 1. *Par table de correspondance, ou table, nous entendons un tableau de 256 éléments (i.e ayant en entrée un byte). Nous parlons de table de nibbles (resp. bytes, mots) si elle fait correspondre un nibble (resp. byte, mot) à une entrée.*

Notant `key` la clé étendue, l'algorithme AES est en général décrit ainsi [1] :

```
state = plaintext;

AddRoundKey (state, key[bloc 0]);
for (round = 0; round < 9; round++)
{
  SubBytes (state);
  ShiftRows (state);
  MixColumns (state);
  AddRoundKey (state, key[bloc (round + 1)]);
}
SubBytes (state);
ShiftRows (state);
AddRoundKey (state, key[bloc 10]);

ciphertext = state;
```

Un choix naturel serait donc d'utiliser un tableau pour chacune des quatre fonctions de base : `SubBytes` , `MixColumns` , `ShiftRows` et `AddRoundKey` . Toutefois, elles prennent en entrée 16 bytes, ce qui demanderait beaucoup trop de mémoire. On va donc regarder s'il n'est pas possible de « découper » ces quatre fonctions.

SubBytes :

En se rappelant que la fonction `SubBytes` consiste en l'application de la sbox sur chaque byte d'un bloc, on voit qu'il est tout à fait possible de définir `SubBytes` sur un byte plutôt qu'un bloc.

On se permettra donc à partir de maintenant d'utiliser `SubBytes` sur un byte et plus généralement tout ensemble de bytes (par exemple une colonne, c'est à dire un mot).

1. par exemple dans la spécification du NIST [8].

AddRoundKey :
Telle que nous l'avons écrite, la fonction `AddRoundKey` n'est qu'un xor, il est donc facile de la considérer sur un byte en restreignant la clé de la bonne façon. On peut de même la considérer sur une colonne.

MixColumns :
La fonction `MixColumns` peut facilement se découper en une fonction sur des colonnes. En effet, `MixColumns` est la multiplication par la matrice

$$MC = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

dans le corps de Rijndael. Or, il est clair que :

$$MC \times \texttt{state} = (MC \times \texttt{state[col 0]} \cdots MC \times \texttt{state[col 3]})$$

On se permettra donc à partir de maintenant d'utiliser `MixColumns` sur une colonne.

ShiftRows :
Elle ne semble pas pouvoir se réduire à moins d'un bloc. On doit donc la conserver telle quelle.

On remarque donc que seule `ShiftRows` ne se prête pas à un découpage permettant de la considérer sur des colonnes. Le choix fait par cette implémentation va alors être de laisser cette fonction « en clair ». En effet, faire un tableau serait trop coûteux en mémoire et cette fonction ne transporte pas d'informations sensibles sur la clé. Pour faciliter notre construction, nous allons alors isoler la fonction `ShiftRows` en modifiant la façon dont est écrit l'algorithme AES.

Comme indiqué dans [6], on remarque tout d'abord qu'il est possible d'inverser `Subbytes` et `ShiftRows`, puis de modifier l'appel à la boucle `for` afin d'obtenir le code suivant :

```
state = plaintext;

for (round = 0; round < 9; round++)
{
  AddRoundKey (state, key[bloc round]);
  ShiftRows (state);
  SubBytes (state);
  MixColumns (state);
}
AddRoundKey (state, key[bloc 9]);
ShiftRows (state);
SubBytes (state);
```

```
AddRoundKey (state, key[bloc 10]);

ciphertext = state;
```

Ensuite, on remarque que - notant `shift_key` la clé étendue `key` à qui on a appliqué `ShiftRows` à tous les blocs - ces deux instructions :

```
AddRoundKey (state, key[bloc round]);
ShiftRows (state);
```

ont le même effet que :

```
ShiftRows (state);
AddRoundKey (state, shift_key[bloc round]);
```

De ces remarques, on obtient finalement la version suivante du code :

```
state = plaintext;

for (round = 0; round < 9; round++)
{
  ShiftRows (state);
  AddRoundKey (state, shift_key[bloc round]);
  SubBytes (state);
  MixColumns (state);
}
ShiftRows (state);
AddRoundKey (state, shift_key[bloc 9]);
SubBytes (state);
AddRoundKey (state, key[bloc 10]);

ciphertext = state;
```

Maintenant, nous n'allons plus appliquer `ShiftRows` , `AddRoundKey` et `SubBytes` au state, mais séparément à chacune de ses quatre colonnes. De plus, comme le dernier tour ne contient pas `MixColumns` et que les autres opérations peuvent s'appliquer directement à des bytes, nous allons faire un découpage en bytes pour celui-ci.

```
state = plaintext;

for (r = 0; r < 9; r++)
{
  ShiftRows (state);

  for (c = 0; c < 4; c++)
  {
    AddRoundKey (state[col c], shift_key[bloc r, col c]);
    SubBytes (state[col c]);
    MixColumns (state[col c]);
```

```
  }
}

ShiftRows (state);
for (elt = 0; elt < 16; elt++)
{
  AddRoundKey (state[elt e], shift_key[bloc 9, elt e]);
  SubBytes (state[elt e]);
  AddRoundKey (state[elt e], key[bloc 10, elt e]);
}

ciphertext = state;
```

Hormis pour le dernier tour, chaque tour consiste en l'application de `ShiftRows` puis d'une boucle agissant sur chacune des quatre colonnes du `state` . Nous appellerons **étapes** les opérations appliquées à une colonne du `state` . On dira alors, par abus de language, qu'un tour est constitué de quatre étapes. Une représentation plus visuelle est donnée par la figure 1.

3.2 Mise en place des tables de correspondance

Ici, nous allons détailler deux choses distinctes :
— comment découper en tables de correspondance une étape, ce qui traite les neuf premiers tours ;
— comment découper en tables de correspondance le dernier tour.

3.2.1 Découpage d'une étape :

Ayant fixé un bloc r pour la clé et considérant sans perte de généralité la colonne 0 d'un bloc `state` , nous allons nous intéresser à la décomposition du bloc d'instructions suivant en plusieurs tables :

```
AddRoundKey (state[col 0], shift_key[bloc r, col 0]);
SubBytes (state[col 0]);
MixColumns (state[col 0]);
```

Pour cela, en notant $C_0, \cdots, C_3$ les colonnes de la matrice MC, on voit que :

$$MC \cdot (x_0, x_1, x_2, x_3)^\top = \ x_0 \cdot C_0 \ \oplus \ x_1 \cdot C_1 \ \oplus \ x_2 \cdot C_2 \ \oplus \ x_3 \cdot C_3$$

Ainsi, en posant :

$$A_{r,0}(x) = \texttt{SubBytes}\left(\texttt{shiftkey[bloc r, col 0, row 0]} \oplus x\right) \cdot C_0$$
$$B_{r,0}(x) = \texttt{SubBytes}\left(\texttt{shiftkey[bloc r, col 0, row 1]} \oplus x\right) \cdot C_1$$
$$C_{r,0}(x) = \texttt{SubBytes}\left(\texttt{shiftkey[bloc r, col 0, row 2]} \oplus x\right) \cdot C_2$$
$$D_{r,0}(x) = \texttt{SubBytes}\left(\texttt{shiftkey[bloc r, col 0, row 3]} \oplus x\right) \cdot C_3$$

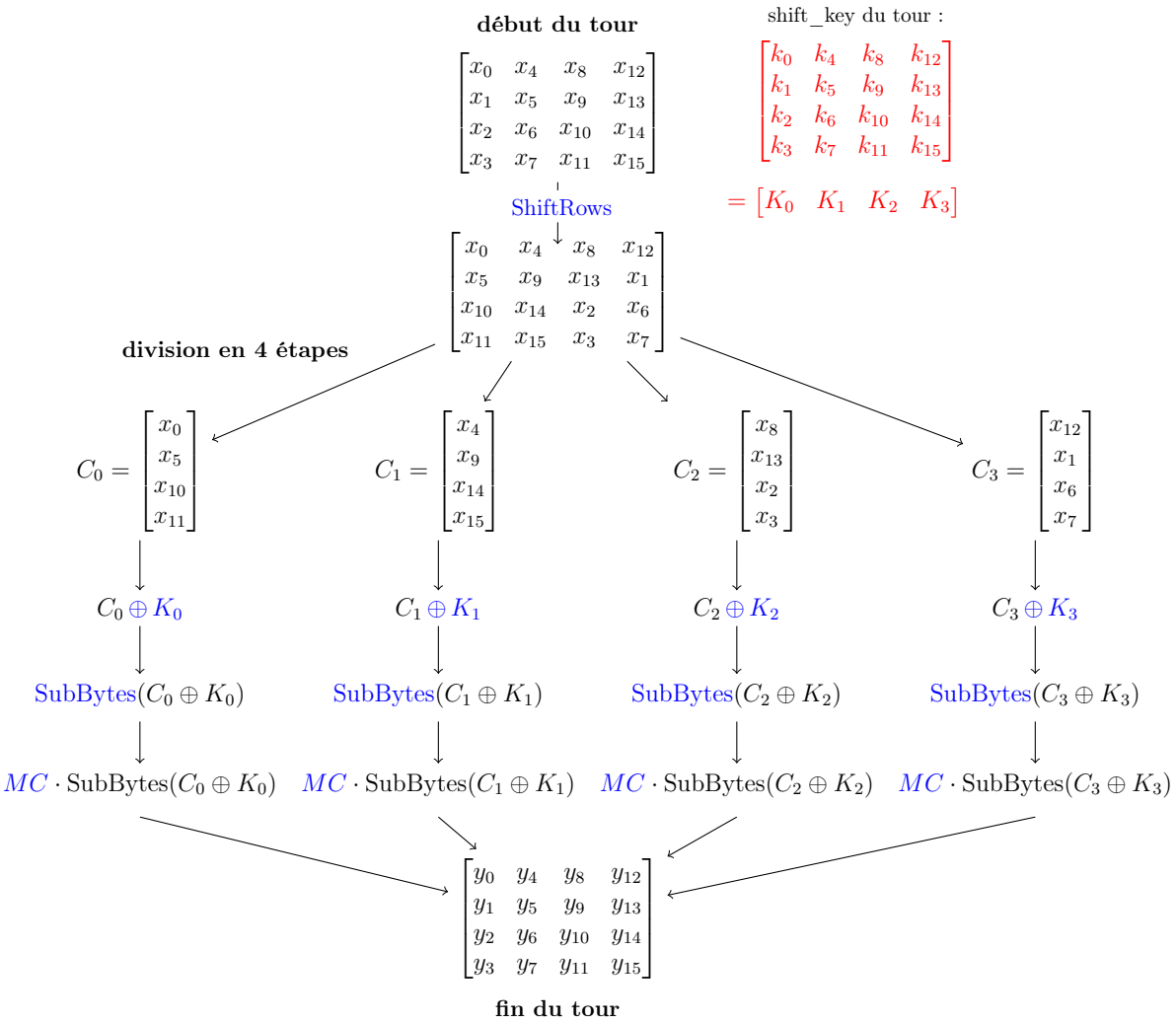


FIGURE 1 – Déroulé d'un tour (hormis le dernier)

on constate que le bloc d'instructions est équivalent à :

$$\begin{aligned} &\big[A_{r,0}(\texttt{state[col 0]}) \oplus B_{r,0}(\texttt{state[col 0]})\big] \\ \oplus\ &\big[C_{r,0}(\texttt{state[col 0]}) \oplus D_{r,0}(\texttt{state[col 0]})\big] \end{aligned} \tag{1}$$

ce qui va permettre de découper ce bloc en une série de tables de correspondance. Indiquons tout de suite le nom des tables dont nous allons avoir besoin :

— $A_{0,c}, B_{0,c}, C_{0,c}$ et $D_{0,c}$ qui sont des tables de mots ;
— des tables de nibbles LXOR1[0], ..., LXOR1[7] pour le premier xor (« left xor ») ;
— des tables de nibbles MXOR1[0], ..., MXOR1[7] pour le second xor (« middle xor ») ;

— des tables de nibbles RXOR1[0], ..., RXOR1[7] pour le dernier xor (« right xor »).

Un xor de deux mots nécessite huit tables XOR. En effet, une table XOR effectue un xor entre deux nibbles. En écrivant une entrée comme la concaténation $a \,||\, b$ des nibbles a et b, on a :

$$\mathrm{XOR}(a \,||\, b) = a \,\oplus\, b$$

Ainsi, les mots sortant des tables A, B, C et D seront chacun séparés en huit nibbles qui seront ensuite injectés (par couple) dans les tables XOR. Le détail au niveau des tables LXOR1 est illustré figure 2. Les liens entre les différentes tables sont représentés sur la figure 3.

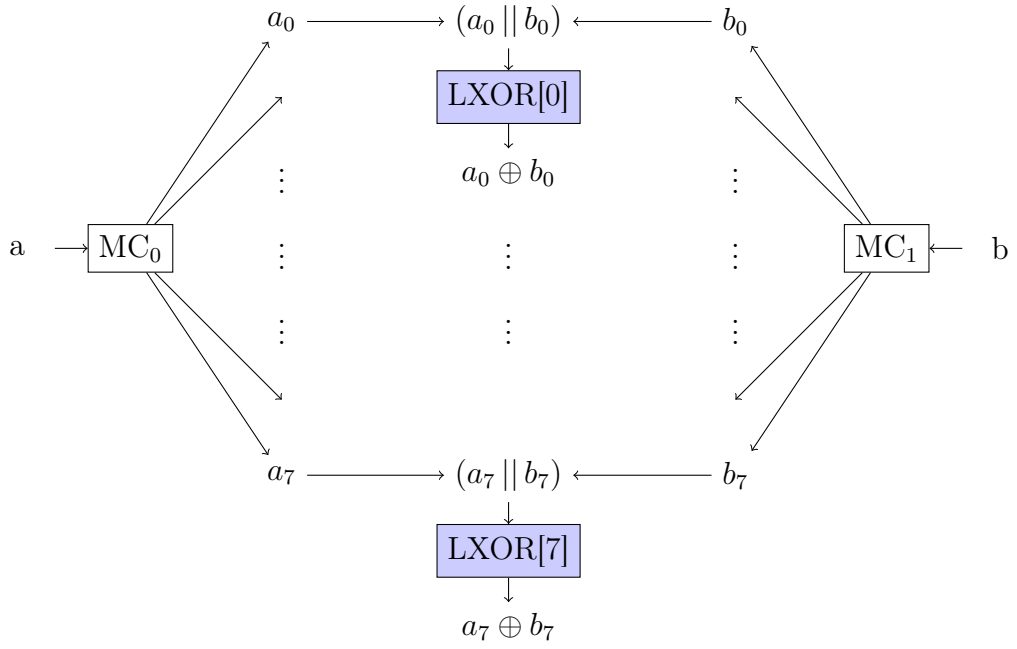


FIGURE 2 – la table A fait correspondre l'octet a au mot $a_0 \,||\, a_1 \,||\, \cdots \,||\, a_7$ et B fait correspondre l'octet b au mot $b_0 \,||\, b_1 \,||\, \cdots \,||\, b_7$

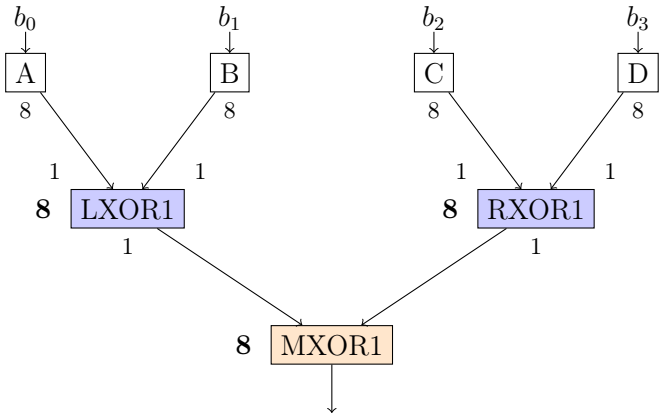


FIGURE 3 – Une étape du circuit principal. Les chiffres au-dessus et en dessous de chaque table indiquent le nombre de nibbles en entrée et en sortie ; les chiffres aux cotés d'une table en indiquent le nombre d'exemplaires. Ainsi, les sorties de A et B sont partagées entre huit tables LXOR1.

Il peut sembler étrange d'utiliser autant de tables XOR identiques. Ce choix sera justifié lors de l'ajout des protections suivantes.

En résumé, pour l'exécution du bloc d'instructions, nous avons besoin de 4 tables de mots et de $3 * 8 = 21$ tables de nibbles. Ceci vaut pour toutes les étapes du premier jusqu'à l'avant dernier tour. On aura donc des tables $A_{r,c}, B_{r,c}, C_{r,c}, D_{r,c}$ définies pour $0 \leq r \leq 8$ et $0 \leq c \leq 3$ et les tables XOR correspondantes. Il y a donc au total $36 * 4$ tables de type A, B, C ou D et $9 * 4 * 24 = 864$ tables XOR.

3.2.2 Découpage du dernier tour :

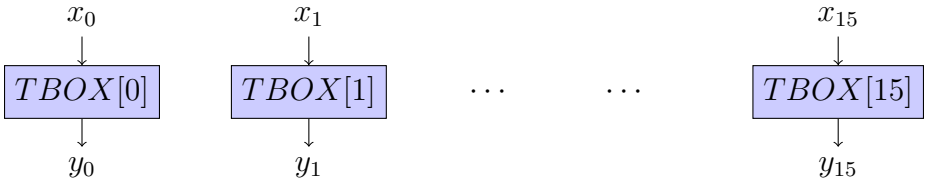
Le pseudo code qui décrit le dernier tour est le suivant :

```
for (elt = 0; elt < 16; elt++)
{
  AddRoundKey (state[elt e], shift_key[bloc 9, elt e]);
  SubBytes (state[elt e]);
  AddRoundKey (state[elt e], key[bloc 10, elt e]);
}
```

On peut alors directement créer 16 tables de correspondance TBOX[0], ... , TBOX[15] pour traiter chacun des octets. Autrement dit, TBOX[e] est telle que pour un byte x :

$$\text{TBOX[e]}[x] = \texttt{SubBytes}\,(x \oplus \texttt{shift\_key[bloc 9, elt e]}) \oplus \texttt{shift\_key[bloc 10, elt e]}$$

On crée 16 tables TBOX supplémentaires et la dernière étape se déroule donc ainsi :



Au final, on obtient les tables de correspondance suivantes :
— tables de mots 36 tables A, 36 tables B, 36 tables C, 36 tables D ;
— tables de bytes : 16 tables TBOX.
— tables de nibbles : 288 tables LXOR1, 288 tables RXOR1, 288 tables MXOR1 ;

Voici le pseudo-code résultant de ces changements.

```
state = plaintext;

for (r = 0; r < 9; r++)
{
  ShiftRows (state);

  for (c = 0; c < 4; c++)
  {
    A_nib = conversion de A[state[col c]] en 8 nibbles;
    B_nib = conversion de B[state[col c]] en 8 nibbles;
    C_nib = conversion de C[state[col c]] en 8 nibbles;
    D_nib =conversion de D[state[col c]] en 8 nibbles;
    for (n = 0; n < 8; n++)
    {
      tmp_l = LXOR1[bloc r, col c, nib n]
      [A_nib[nib n] ||  B_nib[nib n]];
      tmp_r = RXOR1[bloc r, col c, nib n]
      [C_nib[nib n] || D_nib[nib n]];
      result[nib n] = MXOR1[bloc r, col c, nib n]
      [C_nib[nib n] || D_nib[nib n]];
    }
    state[column c] = conversion de result en 4 bytes;
  }
}

ShiftRows (state);

for (elt = 0; elt < 16; elt++)
state[elt e] = TBOX[e][state[elt e]];

ciphertext = state;
```

Correspondance avec notre code :

Dans notre implémentation, les tables figurent dans le fichier *whitebox_ aes/include/tables.h* et sont créées par notre obfuscateur situé dans le dossier *whitebox_ aes_ generator/*.

3.3 Première attaque

Jusqu'à présent, notre code contient toujours de façon simple la clé. En effet, on doit se rappeler qu'elle est égale au premier bloc de la clé étendue. Notant ici `key` la clé (non étendue), nous savons alors que :

$$
\begin{aligned}
A_{0,i}(0) &= \texttt{SubBytes}\left(\texttt{ShiftRows(key)[col i, row 0]}\right) \cdot C_0 \\
B_{0,i}(0) &= \texttt{SubBytes}\left(\texttt{ShiftRows(key)[col i, row 1]}\right) \cdot C_1 \\
C_{0,i}(0) &= \texttt{SubBytes}\left(\texttt{ShiftRows(key)[col i, row 2]}\right) \cdot C_2 \\
D_{0,i}(0) &= \texttt{SubBytes}\left(\texttt{ShiftRows(key)[col i, row 3]}\right) \cdot C_3
\end{aligned}
$$

ce qui montre que la clé se retrouve aisément car les fonctions appliquées sont inversibles. Il faut donc protéger les tables.

4 Encodages internes

Dans cette section, nous parlerons simplement d'encodages, le qualificatif « interne » n'étant utile que pour les distinguer des encodages externes définis plus tard.

4.1 Une protection ajoutant de la confusion

La deuxième protection proposée dans l'article va permettre de protéger le contenu des tables. En faisant une analogie avec les notions de diffusion et de confusion de Shannon, on peut dire que cette protection va ajouter de la confusion.

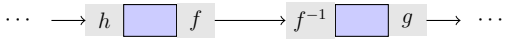
Elle consiste à ajouter en début et en fin de chaque table une bijection choisie aléatoirement. Autrement dit, pour toute table T, on prend 2 bijections aléatoires f et g et on remplace l'ancienne table par :

$$g \circ \mathrm{T} \circ f$$

f est alors appelé l'encodage d'entrée de T et g son encodage de sortie. Le contenu de T n'est alors plus visible.

Évidemment, si on fait cela pour chaque table sans autre précaution, notre code ne marchera plus. Il faut donc choisir les encodages de façon à ce que si une table T est directement suivie par une table T', leurs encodages s'annulent. Par exemple, dans le cas simple :

il faut mettre :



Parfois, la valeur x obtenue par un encodage de sortie ne sera pas directement composée à un encodage d'entrée mais sera décomposée en une concaténation de nibbles $x_0 \,||\, \cdots \,||\, x_n$ dont chaque élément pourra aller vers des encodages d'entrée distincts.

Pour cette raison, on va imposer une restriction sur les encodages possibles :

Définition 2. *un encodage est une concaténation de bijections entre nibbles. Par exemple, un encodage entre bytes est une fonction $f = f_0 \,||\, f_1$ où f_0 et f_1 sont des bijections entre nibbles.*

Ainsi, l'enchaînement de tables suivant :

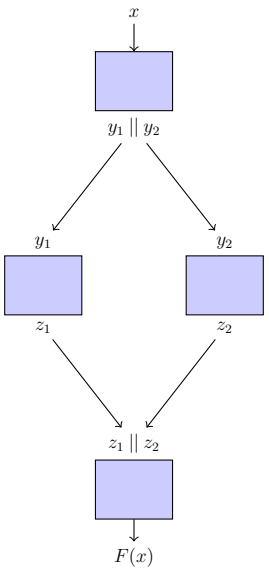
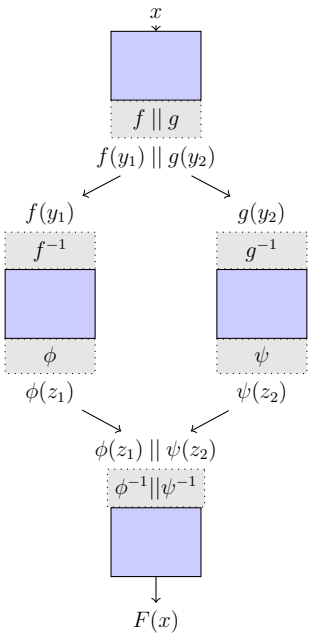


FIGURE 4 – Exemple simplifié, tous les variables indiquées sont des nibbles, ainsi, la sortie de la première table est un octet.

sera protégé ainsi par des encodages :



Une autre remarque qu'on peut tirer de cet exemple est qu'il semble logique de ne mettre d'encodage ni à l'entrée de la première table ni à la sortie de la dernière table car ils ne pourraient pas être annulés. En fait, ceci peut se faire et sera expliqué dans une future section.

Maintenant que nous avons vu la notion sur un exemple, examinons là sur notre obfuscation d'AES. Un cas typique concerne les tables A et B : comme le montre la figure 2 page 12, l'encodage de sortie de la table A doit nécessairement être la concaténation de 8 bijections entre nibbles.

En application de la remarque précédente, on ne donne pas d'encodages d'entrée aux 16 tables A, B, C, et D utilisées au premier tour. La situation est similaire pour les 16 TBOX utilisées au dernier tour à qui on n'associe pas d'encodages de sortie.

Nous représentons de la façon suivante une étape d'un tour protégée par les encodages :

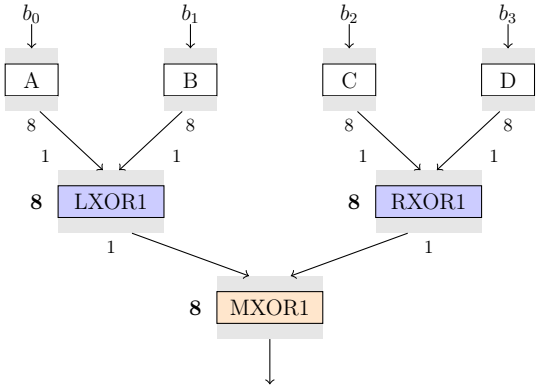


FIGURE 5 – Une étape d'un tour hormis le premier

et une étape du premier tour :

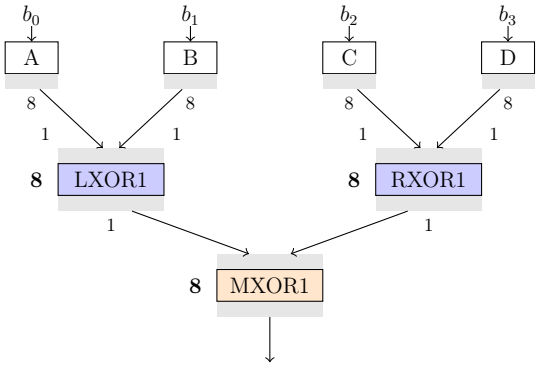


FIGURE 6 – Une étape du premier tour

Correspondance avec notre code :

Dans notre code, nous notons IT et OT pour signifier les encodages d'entrée (input) et de sortie (output) d'une table T. Dans le fichier *table_ writing.h*, nous avons créé 3 types qui couvrent tous les cas possibles :

```
typedef uint8_t nibble_t;

typedef nibble_t function_t[16];

typedef function_t word_plug_t[8];
typedef function_t byte_plug_t[2];
typedef function_t nibble_plug_t;
```

On voit donc que toutes les fonctions considérées seront des concaténations de fonctions entre nibbles.

Nous avons ensuite créé des fonctions pour associer une bijection et son inverse à des couples d'encodages :

```
void
branch_byte (byte_plug_t out, byte_plug_t in,
bool activated)
void
branch_nibble (nibble_plug_t out, nibble_plug_t in,
bool activated)
void
branch_word (word_plug_t out, word_plug_t in,
bool activated)
```

Si le booléen est vrai, chacune de ces fonctions va générer aléatoirement un couple de bijections inverses l'une de l'autre puis attribuer l'une des fonctions à `in` et l'autre à `out`.

Toute la difficulté revient maintenant à correctement « brancher » les encodages d'entrée et de sortie. Cela est fait dans le code et des indications plus précises à ce propos peuvent aussi être trouvées dans [6].

4.2 Deuxième attaque

Les tables sont maintenant mieux protégées et l'attaque de la section 3.3 ne donnerait que des résultats aléatoires. Il va donc falloir être un peu plus fin pour obtenir la clé.

Pour cela, il sera seulement nécessaire d'attaquer les A,B, C et D [2] utilisées au premier tour, soit 16 tables. Il s'agit en effet exactement des tables contenant un morceau de la clé, la clé étant le premier bloc de la clé étendue.

Il suffit donc de récupérer les morceaux de clé situés dans ces 16 tables puis de les mettre dans le bon ordre en appliquant l'inverse de `ShiftRows` .

Nous supposons donc maintenant être dans une étape du premier tour, décrite par la figure 6. De plus, nous allons seulement extraire le morceau de clé de la table A, l'attaque étant similaire pour les autres.

Convention importante : nous allons voir tout tableau T de 16^2 entrées comme un tableau à 2 dimensions (ou bien une matrice) avec la convention suivante :

$$T(i \,||\, j) = T[i][j] \;\; (= T_{i,j})$$

4.2.1 Signature de fréquence d'un tableau de bytes

Considérons un tableau de nibbles de taille $N \times N$. Pour une ligne (resp colonne) i fixée, on fait un tableau comptant le nombre d'apparitions de chaque nibble :

élément	0	1	2	...	15
nombre d'apparitions	n_0	n_1	n_2	...	n_{15}

On trie ensuite les n_i du plus grand au plus petit, la suite obtenue est appelée signature de fréquence de la ligne (resp colonne) i du tableau. Notez qu'on peut omettre les 0 en fin de suite qui n'apportent aucune information supplémentaire.

Si on considère un tableau de bytes (resp. de mots) on peut aussi lui attribuer des signatures de fréquence. Il suffit pour cela de constater qu'un byte est la concaténation de 2 nibbles (resp. 8 nibbles). On obtient ainsi non plus une mais 2 signatures (resp. 8 signatures) pour chaque ligne et pour chaque colonne.

Prenons l'exemple du tableau 4×4 d'octets de la figure 7.

2. Par abus de language, on appelle encore A la table A composée avec des encodages

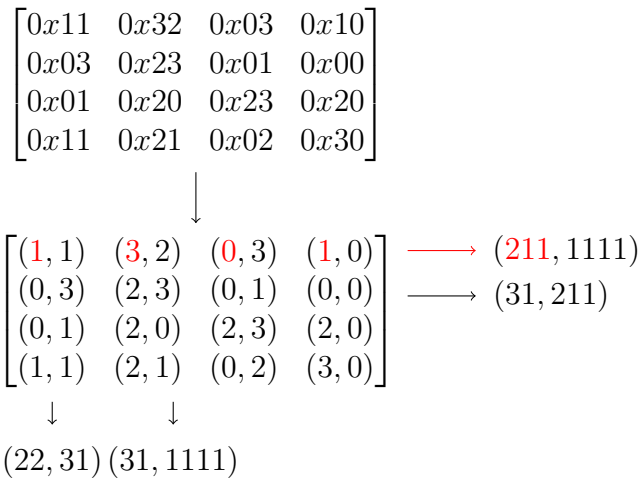


FIGURE 7 – On sépare d'abord chacune des valeurs du tableau en couples de nibbles puis on applique la définition.

Si on compte les nibbles de poids fort de la première ligne, on obtient :

élément	0	1	2	4	5	$\cdots$	15
nombre d'apparitions	1	2	0	1	0	$\cdots$	0

La signature de fréquence de cette ligne est donc $(2, 1, 1)$, notée 211 sur la figure. Justifions maintenant l'intérêt de cette notion.

Proposition 1. *Pour un tableau de nibbles (resp. bytes, mots) T et un encodage f entre nibbles (resp. bytes, mots), notant $f \circ T$ le tableau défini par*

$$(f \circ T)(x) = f(T[x])$$

les signatures de fréquence de T sont égales aux signatures de fréquence correspondantes de $f \circ T$. Autrement dit, la signature de fréquence est un invariant par composition avec un encodage sortant.

Démonstration. Pour un tableau de nibbles, c'est évident par construction. Pour un tableau de bytes ou de mots, c'est tout aussi évident en se rappelant qu'un encodage est par construction une concaténation de bijections entre nibbles. □

4.2.2 Utilisation de la signature de fréquence pour une attaque

Rappelons nous que par construction, A est la composition du tableau de mots T de format 16×16 qui à $i \,||\, j$ associe

$$T(i \oplus k_0 \,||\, j \oplus k_1) = \begin{pmatrix} 02 * \text{SubBytes} \\ 01 * \text{SubBytes} \\ 01 * \text{SubBytes} \\ 03 * \text{SubBytes} \end{pmatrix} (i \oplus k_0 \,||\, j \oplus k_1)$$
$$= \begin{pmatrix} 02 * \text{SubBytes}(i \oplus k_0 \,||\, j \oplus k_1) \\ 01 * \text{SubBytes}(i \oplus k_0 \,||\, j \oplus k_1) \\ 01 * \text{SubBytes}(i \oplus k_0 \,||\, j \oplus k_1) \\ 03 * \text{SubBytes}(i \oplus k_0 \,||\, j \oplus k_1) \end{pmatrix}$$

et d'un encodage de sortie F.

Pour faire l'attaque, on constate que grâce à la proposition précédente, la signature de fréquence de T en une certaine ligne (resp. colonne) est la même que celle de $F \circ T$. Ainsi, il est possible de calculer la signature de T à la ligne k_0 (en prenant $i = 0$ et en faisant varier j) et à la colonne k_1 (en prenant $j = 0$ et en faisant varier i).

Il reste à savoir si la connaissance de la signature de T en la ligne k_0 (resp. colonne k_1) permet de retrouver k_0 (resp. k_1).

Par chance, on peut montrer que les signatures de lignes et de colonnes de T sont toutes distinctes. On peut donc retrouver de façon non ambigüe l'indice de colonne ou de ligne à partir des signatures qu'il a produit. Ceci nous permet donc de retrouver k_0 et k_1.

Si on ne considère pas A mais plutôt B, C où D, la situation est identique, le tableau T considéré étant similaire. On peut ainsi retrouver la clé.

Implémentation :

Remarquons que connaitre les signatures de T est équivalent à connaitre les signatures de S, $02 * S$ et $03 * S$. C'est donc elles qui sont stockées dans notre code. Plus précisément : les signatures de lignes sont stockées dans un tableau `ROW` de 16 éléments qui à l'indice i contient les signatures de ligne i de S, $02 * S$ et $03 * S$ (donc six nombres par entrée) tandis que celles de colonnes sont stockées dans un tableau `COL`. Ces tableaux sont définis dans un header *include/COL_ROW.h* et ont été créés par l'exécutable *src/make_COL_ROW* dont c'est la seule utilité.

Nous avons implémenté cette attaque et elle s'avère très efficace : la clé est trouvée en moins d'une demi seconde.

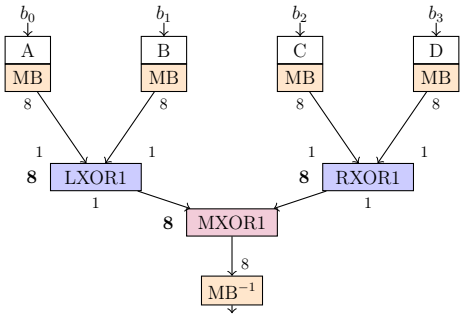
5 Mixing bijections

5.1 Une protection ajoutant de la diffusion

Comme le montre l'attaque précédente, les encodages ne constituent pas une protection suffisante et cela est dû à un manque de diffusion. En effet, la signature de fréquence est un invariant car les encodages sont scindés en bijections entre nibbles, ce qui implique que la transformation d'un nibble n'est pas en rapport avec celle de ses voisins. Nous allons maintenant ajouter des fonctions qui vont « mélanger » les nibbles entre eux et rendre caduque l'attaque précédente. Cela peut notamment se faire avec des isomorphismes linéaires, que les auteurs de l'obfuscation nomment « mixing bijections ».

Définition 3. *Nous appellerons mixing bijection entre nibbles (resp. bytes, mots) un isomorphisme entre $\mathbb{F}_2$-espaces-vectoriels de dimension 4 (resp. 8, 32).* [3]

Prenons une mixing bijection MB entre mots et composons-la avec les tables A, B, C et D pour une étape donnée. Comme MB est linéaire, on peut placer son inverse après les tables de xor. On pourrait alors simplement composer MB^{-1} à la suite de MXOR1 comme montré ici :



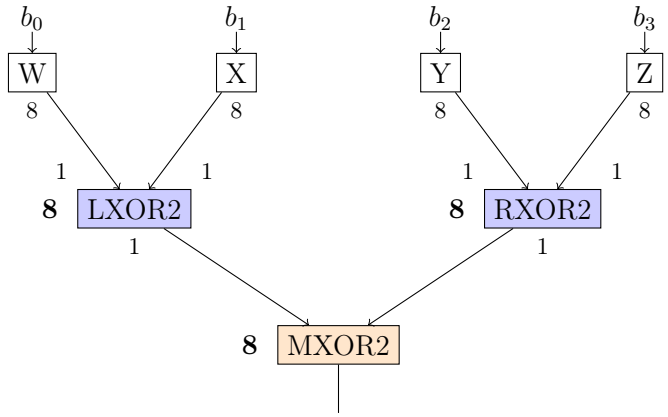
Toutefois, cela nécessiterait un tableau ayant 2^{32} entrées : nous faisons face au même problème que pour le découpage de `MixColumns` .

Nous allons donc utiliser la même solution et considérer les colonnes W, X, Y, Z de MB^{-1} afin d'avoir :

$$\mathrm{MB}^{-1}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathrm{MB}^{-1}\begin{pmatrix} x_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \oplus \mathrm{MB}^{-1}\begin{pmatrix} 0 \\ x_1 \\ 0 \\ 0 \end{pmatrix} \oplus \mathrm{MB}^{-1}\begin{pmatrix} 0 \\ 0 \\ x_2 \\ 0 \end{pmatrix} \oplus \mathrm{MB}^{-1}\begin{pmatrix} 0 \\ 0 \\ 0 \\ x_3 \end{pmatrix}$$
$$= x_0 \cdot W \oplus x_1 \cdot X \oplus x_2 \cdot Y \oplus x_3 \cdot Z$$

De la même façon que pour `MixColumns` , on va alors placer de nouvelles tables xors :

3. Notons que [9] demande certaines conditions supplémentaires pour augmenter encore la diffusion.



$$\big[W(x_0) \oplus X(x_1)\big] \oplus \big[Y(x_2) \oplus Z(x_3)\big]$$

Une étape générique ressemble maintenant à ce que montre la figure 8.

Toutefois, si on estime que les encodages ne fournissent pas une protection suffisante, il faut aller plus loin. En effet, les 4 tables W, X, Y et Z ne sont protégées que par des encodages alors qu'elles permettent de retrouver MB^{-1}. Les laisser ainsi revient donc à ne pas avoir ajouté de protection étant donné que la protection d'un système se mesure à celle de sa partie la plus fragile.

Nous allons donc aussi protéger W, X, Y et Z avec des mixing bijections pour pallier à ce problème.

Pour chaque $0 \leq r \leq 8$, nous prenons 16 isomorphismes entre bytes :

$$L_0^{r+1}, \cdots, L_{15}^{r+1}$$

et on crée aussi 4 isomorphismes :

$$\begin{aligned}
\mathrm{LC}_0^r &= L^{r+1^{-1}}_{\sigma^{-1}(0)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(1)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(2)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(3)} \\
\mathrm{LC}_1^r &= L^{r+1^{-1}}_{\sigma^{-1}(4)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(5)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(6)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(7)} \\
\mathrm{LC}_2^r &= L^{r+1^{-1}}_{\sigma^{-1}(8)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(9)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(10)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(11)} \\
\mathrm{LC}_3^r &= L^{r+1^{-1}}_{\sigma^{-1}(12)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(13)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(14)} \,||\, L^{r+1^{-1}}_{\sigma^{-1}(15)}
\end{aligned}$$

où on note σ la permutation sur 16 éléments faite par `ShiftRows` .

Chaque L^i sera pré-composé à une des tables A, B, C ou D d'une étape[4] tandis que LC^r sera composé plusieurs fois : avec W, X, Y et Z. Cela peut se voir dans la figure 9.

Comme le montre la définition, les LC_i^r, placés au tour r, permettront d'inverser les tables L_i^{r+1} du tour suivant.

Avec cette construction, on constate que l'inverse de LC_i^r peut simplement se découper en 4 tables de correspondance (contrairement à un isomorphisme linéaire

4. ou bien au TBOX si nous sommes dans le dernier tour

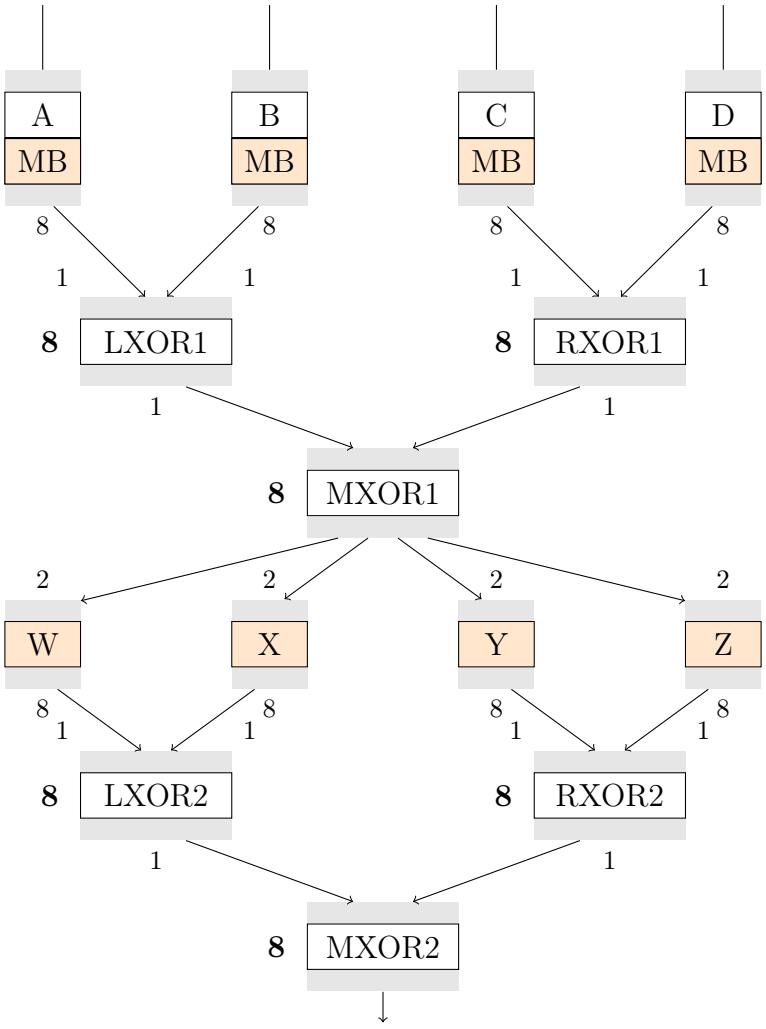


FIGURE 8 – une étape du circuit principal avec MB et son inverse scindé en W, X, Y et Z.

quelconque, comme par exemple MB^{-1}, qui demande plus de travail et d'introduire des tables xor) formées par certains LC_i.

Notons que σ intervient ici car LC et les L_k interagissent sur des tours différents et qu'alors la fonction `ShiftRows` vient mettre la pagaille. La même chose se fait aussi avec les encodages de fin et de début d'une étape mais nous avions caché la chose sous le tapis.

Remarquons enfin qu'il n'y a donc pas de L_i dans le premier tour ; cela peut se voir dans la figure 10.

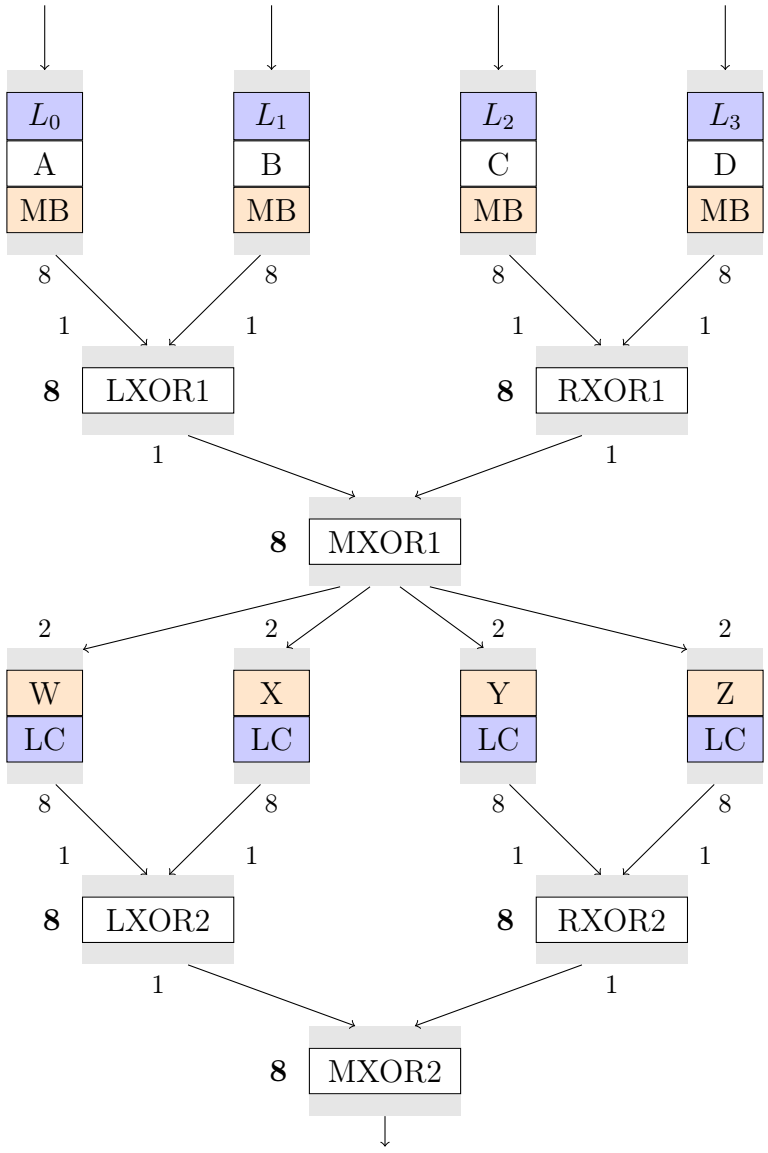


FIGURE 9 – une étape (hormis pour le premier tour) avec les mixing bijections .

Si on fait le décompte du nombre de tables nécessaires, on a donc :
— tables de mots $36 * 8 = 288$ tables ;

— tables de bytes : 16 tables ;
— tables de nibbles : $6 * 288 = 1728$ tables.

Il y a donc au final 2032 tables et elles occupent une mémoire de $256 * (288 * 4 + 16 + 1728/2) = 520192$ octets soit environ 520 Ko.

Correspondance avec notre code :

Dans notre code, les tables MB, L et LC se nomment respectivement `mixing_bijection`, `mixing_bijection_2` et `mixing_bijection_2_concat`.

5.2 Troisième attaque

Remarque 1. *Pour des raisons de lisibilité, nous noterons souvent dans cette section* `Mix` *pour* `MixColumns` *et* `Sub` *pour* `SubBytes` .

L'attaque que nous allons maintenant évoquer est succinctement décrite dans [9] et va nous permettre de réduire l'espace des clés entre environ 2^{44} (cas le plus probable) et 2^{60} (très improbable).

Nous allons pour cela nous focaliser sur une des étapes du premier tour et nous notons $k = (k_0, \cdots, k_3)$ le mot tiré de la clé qui y est utilisé (k_0 est caché dans la table A et k_3 dans la table D).

La situation lors d'une de ces étapes est montrée dans la figure 10. Considérons un mot $w = (w_0, \cdots, w_3)$. Si on lui applique les tables de l'étape, on pourra alors observer que sa valeur après la table MXOR2 est :

$$\tilde{w_i} = F \circ \mathrm{LC}\left[(\texttt{Mix}\ (\texttt{Sub}\ (k_i \oplus w)))_i\right]$$

où F est l'encodage sortant de MXOR2.

Comme $F \circ \mathrm{LC}$ est bijective, on a la proposition suivante :

Proposition 2. *Pour deux mots x et w et $0 \leq i \leq 3$:*

$$\tilde{x_i} = \tilde{w_i} \implies (\texttt{Mix}\ (\texttt{Sub}\ (k \oplus x)))_i = (\texttt{Mix}\ (\texttt{Sub}\ (k \oplus w)))_i$$

Pour réaliser l'attaque, on va commencer par chercher deux mots $x = (x_i)$ et $w = (w_i)$ tels que

— pour $0 \leq i \leq 3$, les bytes x_i et w_i sont distincts ;
— $\tilde{x}$ et $\tilde{w}$ sont égaux sur 3 bytes ;

Posant :

$$y = (y_i)_{0 \leq i \leq 3} = \ (\texttt{Sub}\ (k_i \oplus x_i) \oplus \texttt{Sub}\ (k_i \oplus w_i))_{0 \leq i \leq 3}$$

on remarque alors que :

$$\begin{aligned} \texttt{Mix}\ (y) &= \texttt{Mix}\ [\texttt{Sub}\ (k \oplus x) \oplus \texttt{Sub}\ (k \oplus w)] \\ &= \texttt{Mix}\ [\texttt{Sub}\ (k \oplus x)] \oplus \texttt{Mix}\ [\texttt{Sub}\ (k \oplus w)] \end{aligned}$$

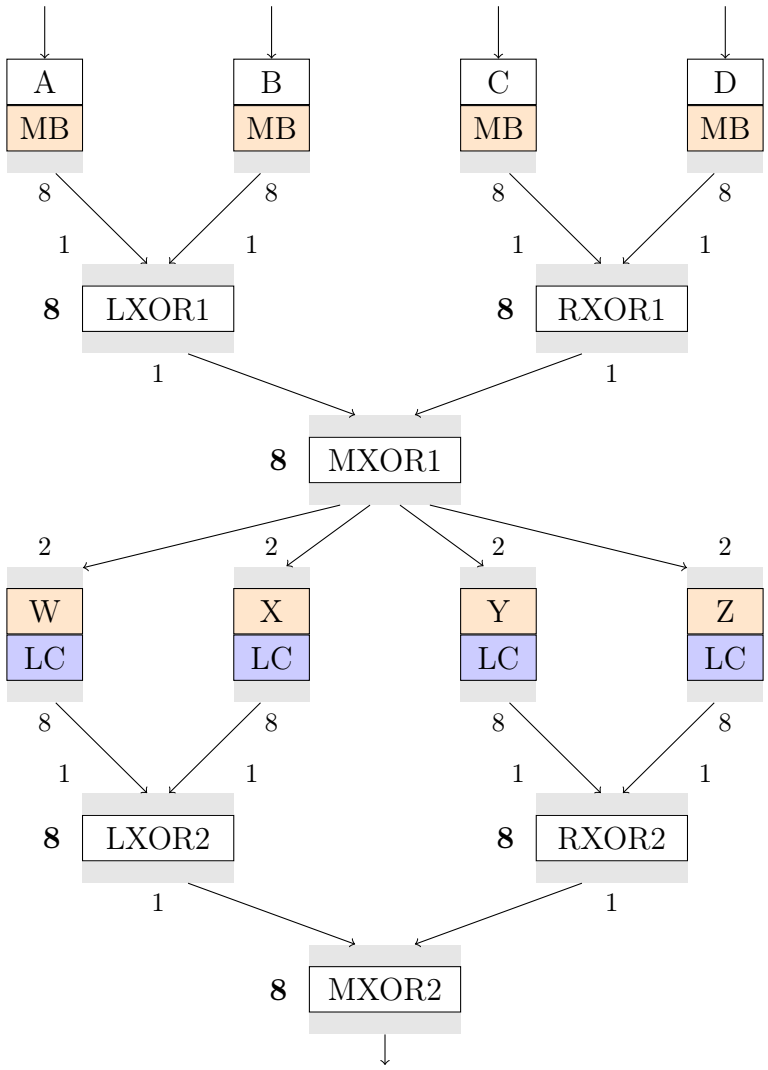


FIGURE 10 – La situation du premier tour : notez qu'il n'y a pas d'encodage d'entrée sur les premières tables.

La proposition précédente implique que `Mix` (y) est nul sur 3 bytes (alors que la valeur du dernier est inconnue). Cette relation va nous permettre de trouver un lien entre les différents y_i comme le montre la proposition suivante :

Proposition 3. *Si* `Mix` (y) *est nul sur trois de ses bytes, il existe alors un entier* $0 \leq i \leq 3$ *et, pour chaque* $0 \leq l \leq 3$ *différent de* i*, un* $a_l \in \mathbb{F}_{2^8}$ *tel que :*

$$y_l = a_l \, y_i$$

Démonstration. `Mix` (y) n'est rien d'autre que :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

où on nomme MC la matrice carrée. Pour $0 \leq i \leq 3$, on note MC_i la matrice obtenue en supprimant la $i$ème ligne. La propriété est alors vérifiée si pour tout $0 \leq i \leq 3$, la matrice MC_i est de rang 3.

Par exemple, si `Mix`(y) est nul sur ses trois premiers bytes, on va essayer de résoudre le système :

$$MC_3 \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

et le fait que MC_3 soit de rang 3 permettra effectivement d'exprimer les y_i en fonction d'un y_j fixé, dépendant de quelle sous matrice 3×3 de MC_3 est inversible.

On donc doit montrer que pour tout i, MC_i est de rang 3. Pour cela, on voit qu'on peut extraire de M_0 la matrice :

$$\begin{pmatrix} 01 & 02 & 03 \\ 01 & 01 & 02 \\ 03 & 01 & 01 \end{pmatrix}$$

qui est inversible, en effet :

$$\begin{vmatrix} 01 & 02 & 03 \\ 01 & 01 & 02 \\ 03 & 01 & 01 \end{vmatrix} = \begin{vmatrix} 01 & 03 & 03 \\ 01 & 02 & 02 \\ 03 & 04 & 01 \end{vmatrix} \quad \text{en ajoutant la première colonne à la deuxième}$$

$$= \begin{vmatrix} 01 & 00 & 03 \\ 01 & 00 & 02 \\ 03 & 03 & 01 \end{vmatrix} \quad \text{en soustrayant la troisième colonne à la deuxième}$$

$$= -03 * \begin{vmatrix} 01 & 03 \\ 01 & 02 \end{vmatrix} = -03 * (-01) = 03$$

De plus, on peut extraire de chacune des autres M_i une matrice ayant au signe près le même déterminant :

de MC_1 :

$$\begin{pmatrix} 03 & 01 & 01 \\ 01 & 02 & 03 \\ 01 & 01 & 02 \end{pmatrix}$$

de MC_2 :

$$\begin{pmatrix} 02 & 01 & 01 \\ 01 & 03 & 01 \\ 03 & 01 & 02 \end{pmatrix}$$

de MC_3 :

$$\begin{pmatrix} 02 & 03 & 01 \\ 01 & 02 & 01 \\ 01 & 01 & 03 \end{pmatrix}$$

La proposition est donc démontrée. □

Par exemple, si c'est la première coordonnée qui est nulle, on a (comme noté dans [9]) :

$$\begin{aligned} y_0 &= ec * y_3 \\ y_1 &= 9a * y_3 \\ y_2 &= b7 * y_3 \end{aligned}$$

Ainsi, pour chaque valeur possible de y_3, y est entièrement déterminé ; or

$$y = (y_i) = \ (\mathsf{Sub}\ (k_i \oplus x_i) \oplus \mathsf{Sub}\ (k_i \oplus w_i))$$

et nous allons maintenant voir que la connaissance de y_i permet de réduire le nombre de k_i possibles.

Proposition 4. *Soient deux bytes u et v distincts et considérons la fonction qui à un byte x fait correspondre :*

$$\mathsf{Sub}\ (x \oplus u) \oplus \mathsf{Sub}\ (x \oplus v)$$

Alors, il existe un byte dont la préimage par cette fonction contient 4 éléments et les préimages des autres bytes sont ou bien vides, ou bien constituées de 2 éléments. De plus, l'image de de cette fonction contient 127 éléments.

Démonstration. Le fait que l'image de la fonction contienne 127 éléments découle directement du reste de la proposition. Montrons le reste. Posons i la fonction qui envoie 0 sur 0 et tout byte non nul sur son inverse. On sait alors qu'il existe une matrice $A \in \mathrm{GL}_{8*8}(\mathbb{F}_2)$ et un vecteur $B \in (\mathbb{F}_2)^8$ tels que pour tout byte x :

$$\mathsf{Sub}\ (x) = A \cdot i(x) \oplus B$$

On en déduit que :

$$\mathsf{Sub}\ (x \oplus u) \oplus \mathsf{Sub}\ (x \oplus v) = A(i(x \oplus u) \oplus i(x \oplus v))$$

Comme nous nous intéressons uniquement à la taille des préimages et que A est inversible ; on voit qu'il suffit de s'intéresser à la fonction qui à x fait correspondre :

$$i(x \oplus u) \oplus i(x \oplus v)$$

On peut encore simplifier cela en faisant un changement de variable : quitte à remplacer x par $x \oplus u$, on voit qu'il suffit d'étudier la fonction

$$\phi(x) = i(x) \oplus i(x \oplus w)$$

où w est non nul (w correspond à $u \oplus v$)

Nous allons donc maintenant seulement supposer un w non nul et étudier les préimages de la fonction ϕ.

Pour tout byte x, on a : $\phi(x) = \phi(x \oplus w)$, ce qui fait que chaque préimage non vide contient au moins 2 éléments.

Montrons alors que tout byte a ne contient pas plus de deux x distincts de 0 et w dans sa préimage. En effet, pour un tel x, on a :

$$\begin{aligned}
&\phi(x) = a \\
&\Leftrightarrow x^{-1} \oplus (x \oplus w)^{-1} = a \quad \text{car } x \text{ et } x \oplus w \text{ sont non nuls} \\
&\Leftrightarrow w = a(x \oplus w)x \\
&\Leftrightarrow ax^2 + awx - w = 0
\end{aligned}$$

or ce polynôme n'a pas plus de 2 racines dans $\mathbb{F}_{2^8}$, ce qui démontre ce que l'on souhaitait.

Enfin, occupons-nous du cas de w et 0. Leur image par ϕ est w^{-1} et nous allons voir que cette valeur est aussi atteinte par 2 autres bytes. En effet, pour x un byte distinct de 0 et w :

$$\begin{aligned}
&\phi(x) = w^{-1} \\
&\Leftrightarrow x^{-1} \oplus (x \oplus w)^{-1} = w^{-1} \quad \text{car } x \text{ et } x \oplus w \text{ sont non nuls} \\
&\Leftrightarrow w^2 = (x \oplus w)x \\
&\Leftrightarrow x^2 \oplus wx \oplus w^2 = 0
\end{aligned}$$

On constate que ce polynôme ne peut pas avoir comme racines w ou 0 car cela impliquerait alors $w^2 = 0$, ce qui est absurde. Ainsi, on a ce que l'on souhaite si on montre que ses racines sont dans $\mathbb{F}_{2^8}$ Nous devons à M. Cerri les arguments qui suivent.

Quitte à changer x par x/w, on voit qu'on peut s'intéresser à trouver les racines de x^2+x+1. On sait qu'une racine r existe dans une extension de $\mathbb{F}_{2^8}$. Pour montrer

qu'elle est dans $\mathbb{F}_{2^8}$, il suffit de prouver qu'on a $r^{2^8} + r = 0$. Or :

$$\begin{aligned} r^{2^8} + r &= \sum_{k=1}^{8} (r^{2^k} + r^{2^{k-1}}) \\ &= \sum_{k=1}^{8} (r^2 + r)^{2^{k-1}} \quad \text{car on est en caractéristique 2} \\ &= \sum_{k=1}^{8} 1 = 0 \quad \text{par hypothèse sur } r \end{aligned}$$

Ainsi, toutes les racines de ce polynôme sont bien dans $\mathbb{F}_{2^8}$: il y a donc bien deux éléments distincts de w et 0 dont l'image par ϕ est w^{-1}.

En rassemblant tout ce que nous avons dit, on constate que la préimage de w^{-1} contient 4 éléments et que toutes les autres préimages sont ou bien vides, ou bien constituées de 2 éléments, ce qu'il fallait démontrer. □

Remarque 2. *Notons que c'est pour avoir cette propriété qu'on souhaite que les deux mots x et w choisis aient des bytes deux à deux distincts.*

Revenant à notre attaque, on voit qu'il y a 127 valeurs possibles pour y_3 et pour une valeur de y_3 fixée, on a :

— dans le meilleur des cas : 2^4 mots de clé k possibles car il y alors 2 k_i possibles pour chaque $0 \leq i \leq 3$;

— dans le pire des cas (très improbable) : $4^4 = 2^8$ mots de clés k possibles car 4 k_i possibles pour chaque $0 \leq i \leq 3$.

En faisant varier y_3 dans les 127 valeurs possibles, on obtient donc au mieux $127 * 2^4$ et au pire $127 * 2^8$ choix possibles pour le mot de clé k. Comme la clé est constituée de 4 mots, on réussit donc à réduire l'espace des clés à $(127 * 2^4)^4 \simeq (2^{11})^4 = 2^{44}$ éléments au mieux et à $(127 * 2^8)^4 \simeq 2^{60}$ éléments au pire.

En fait, si on n'est pas dans le cas optimal, il semble judicieux de chercher de nouvelles collisions ; cela est en effet assez rapide comme nous avons pu le constater avec notre implémentation.

Implémentation :

Nous n'avons pas implémenté l'attaque complète, notamment car nous trouvions que le nombre de clés restantes restait trop grand. Nous avons toutefois écrit un programme qui cherche les collisions décrites et il met généralement 1 minute pour trouver les 4 collisions nécessaires.

6 Encodages externes

Nous noterons dans cette section E_k la fonction de chiffrement AES associée à une clé k.

Pour que l'attaque précédente fonctionne, il est évidemment nécessaire de pouvoir « savoir » que la clé k testée est ou non la bonne. Cela se fait simplement en comparant les sorties du programme et de E_k.

Supposons alors qu'on ait choisi deux bijections F et G entre blocs et que le programme whitebox ne calcule non pas E_{k_0} mais $F \circ E_{k_0} \circ G$ pour une clé k_0 fixée. Il n'est alors plus possible pour un attaquant ne connaissant ni F ni G de savoir si la clé k qu'il essaie est la bonne et l'attaque précédente ne peut donc plus fonctionner. Une autre raison pour laquelle l'attaque précédente ne fonctionne pas est que la recherche de collisions qu'elle nécessite ne peut plus se faire.

Bien sûr, pour que le programme ait encore un intérêt, il faut qu'un programme ou un composant externe, non accessible/analysable par l'attaquant, permette d'annuler les effets de F et G. Notons que pour intégrer F et G aux circuits de tables déjà établis, le plus simple est de faire comme les encodages externes et de demander qu'elles soient des concaténations de bijections entre octets afin de les intégrer aux tables A, B, C, D au départ et aux TBOX à l'arrivée. Toutefois, Chow et al. conseillent plutôt de prendre pour F et G des mixing bijections, ce qui implique la création de tables supplémentaires [5].

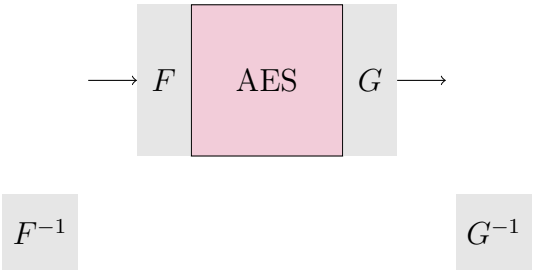


FIGURE 11 – Encodages externes.

Dans ces cas là, des attaques sont malgré tout toujours possibles comme par exemple l'attaque BGE due à O. Billet, H. Gilbert et C. Ech-Chatbi, expliquée dans [3] et étudiée par d'autres personnes notamment dans [7]. Étant bien plus complexe et longue que les précédentes, nous n'avons pas eu le temps de l'étudier.

Correspondance avec notre code :

Nous n'avons pas codé cette protection.

7 Code source

Dans cette dernière partie, nous détaillons comment utiliser les fichiers que nous avons écrits. Nous avons utilisé quelques fonctions mathématiques offertes par la GSL [6]. Le code requiert donc l'installation de cette librairie.

5. Nous n'entrerons pas dans le détail mais la décomposition est similaire à celle de MB^{-1} vue dans la section précédente

6. GNU Scientific Library

L'archive contient sept répertoires dont voici une brève description (voir les parties correspondantes pour plus de détails).

— Le dossier *rapport/* contient les sources LaTeX du présent rapport.
— Le dossier *bibliography/* contient les documents sur lequels nous nous sommes appuyés.
— Le dossier *basic_aes/* contient une implémentation de l'algorithme AES standard.
— Le dossier *whitebox_aes_generator/* contient le code source du programme qui à partir d'une clé, génère les tables de correspondance utilisées dans l'algorithme AES whitebox.
— A partir d'un fichier de tables obtenu avec le générateur, le code du répertoire *whitebox_aes/* compile un AES whitebox .
— Le répertoire *attacks/* possède trois sous répertoires, un pour chacune des trois attaques implémentées [7].
— Le dossier *tests/* contient les tests effectués sur l'AES standard ainsi que le générateur d'AES whitebox.

7.1 AES standard

Nous avons implémenté un AES 128 bits classique dans le répertoire *basic_aes/*. Le binaire s'obtient par un simple `make` et s'exécute avec deux arguments. Toutes les fonctionnalités implémentées sont accessibles via l'option `--help`.

L'opération par défaut est le chiffrement. L'option `-i` (inverse cipher) permet de procéder au déchiffrement. L'option `-o` permet de spécifier un nom de fichier de sortie. Sans cette option, le nom par défaut est celui du fichier traité accolé d'un suffixe `.aes`. Deux modes opératoires sont supportés : ECB (`--mode=ecb`) et CBC (`--mode=cbc`). Dans le mode Cipher Block Chaining, nous avons opté pour la norme de padding la plus courante : PKCS#7.

7.2 Générateur AES whitebox

Les répertoires *whitebox_aes_generator/* et *whitebox_aes/* contiennent à eux deux le code source du générateur AES whitebox.

Dans le dossier *whitebox_aes_generator/* se trouve la partie liée à l'obfuscation. Le *Makefile* compile le binaire *generate_tables*. Ce dernier obfusque une clé en la transformant en un ensemble de tables. Ces tables sont rassemblées dans le fichier *tables.h* dans le répertoire *whitebox_aes/include/*. Les protections désirées s'obtiennent en exécutant le binaire avec des options. `generate_tables --help` en donne les détails.

Une fois les tables générées, Le *Makefile* du répertoire *whitebox_aes/* peut compiler un binaire AES whitebox. Un AES whitebox ainsi créé ne propose que le chiffrement. On peut consulter les fonctionnalités supportées via l'option `--help`.

7. L'attaque avec mixing bijections n'ayant pas été entièrement implémentée, seule la recherche de collisions est faite.

Pour faciliter l'utilisation, le répertoire *whitebox_aes_generator/* contient le script python *generate_aes*. Son utilisation nécessite au préalable la compilation de l'exécutable *generate_tables*. Le script *generate_aes* supporte les mêmes options et arguments que le binaire *generate_tables*. Ainsi le script exécute *generate_tables* (avec mêmes arguments et options), puis lance la compilation de l'AES whitebox (dans le répertoire *whitebox_aes/*) et enfin copie le binaire résultant dans le répertoire courant. Le script supporte en plus une option de clé aléatoire (il faut passer un nom pour le fichier clé qui sera généré). Une aide est également disponible via : `python generate_aes --help`.

7.3 Tests

Le répertoire *tests/* est décliné en deux sous répertoires *basic_aes/* et *whitebox_aes_generator/* qui contiennent les tests pour respectivement l'AES standard et le générateur AES whitebox. Il suffit pour chacun d'exécuter le fichier *test* après l'avoir compilé avec `make`. `--help` détaillent les options disponibles.

Pour l'AES standard nous avons effectué deux sortes de tests. D'abord nous avons récupéré des jeux de données du NIST et vérifié pour ces entrées que les sorties de notre algorithme étaient bien celles attendues. Ensuite nous avons procédé successivement au chiffrement puis au déchiffrement de fichiers en s'assurant qu'on retrouvait à chaque fois le fichier de départ. Les fichiers testés ont été choisis de façon à couvrir une majorité de cas (fichier vide, fichier dont la taille est multiple du nombre de blocs, ...). Ces fichiers se trouvent dans les sous répertoires *ecb_inputs* et *cbc_inputs*. Chaque fichier se trouve dans les deux répertoires pour tester les deux modes opératoires. À l'issue des tests, les fichiers chiffrés (intermédiaires) et les fichiers de sortie se trouvent dans les mêmes répertoires que les originaux. Les noms sont aussi les mêmes à suffixe près (`.aes` pour les chiffrés et `.cpy` pour les déchiffrés).

Pour tester notre générateur AES whitebox, le binaire *test* procède comme suit. Il génère aléatoirement trois clés auxquelles sont ajoutées deux autres clés fixées. Il place les cinq clés dans le sous répertoire *keys* et pour chacune d'elles :

- il construit les tables de correspondance (en utilisant des protections différentes d'une clé à l'autre) ;
- compile l'AES whitebox correspondant ;
- l'utilise pour chiffrer tous les fichiers du sous répertoire *inputs* ;
- chiffre par ailleurs ces mêmes fichiers avec l'AES standard (en utilisant évidemment la clé en question) ;
- compare enfin les chiffrés.

Remarques (sur les tests whitebox) :

- il est facile d'ajouter d'autres fichiers à tester. Il suffit de les mettre dans le sous répertoire *inputs*. Ils seront alors automatiquement inclus dans les tests.
- comme l'ensemble des tests peut prendre plusieurs secondes et générer beaucoup d'affichage, il peut être utile de ne lancer que quelques tests. Ceci est faisable en passant leurs numéros. Pour avoir les numéros : `test -l`.

Attention toutefois aux dépendances (chiffrer échoue nécessairement si l'AES whitebox n'existe pas !).
— l'option `-v` affiche les étapes intermédiaires du chiffrement (à combiner de préférence avec un numéro de test).
— comme d'habitude `--help` récapitule tout ça.

7.4 Attaques

Chaque attaque présuppose que l'exécutable *whitebox_ aes_ generator/generate_ tables* ait été utilisé pour générer les tables de correspondance (dans *whitebox_ aes/include/tables.h*) avec les protections correspondantes.

7.4.1 Sur l'implémentation sans protection
(répertoire *attacks/attack_ basic/)*

On suppose ici qu'un algorithme AES a été créé sans aucune protection (option `-z`). Il suffit alors de compiler l'exécutable *attack* avec la commande `make` et de l'exécuter sans options pour qu'apparaîsse la clé.

Notons que si l'algorithme AES attaqué est protégé par des encodages ou bien des mixing bijections, la clé affichée ne sera pas la bonne (et sera différente à chaque lancement de l'exécutable).

7.4.2 Sur les tables protégées par des encodages
(répertoire *attacks/attack_ encodings/)*

On suppose ici qu'un algorithme AES a été créé sans mixing bijections (option `-e`). Il suffit alors de compiler l'exécutable *attack* avec la commande `make` et de l'exécuter sans options pour qu'apparaîsse la clé.

Notons que si l'algorithme AES attaqué est protégé par des mixing bijections, l'attaque ne fonctionnera pas et un message d'erreur apparaîtra.

7.4.3 Sur les tables protégées par encodages et mixing bijections
(répertoire *attacks/attack_ mixing/)*

Ici, ce n'est pas l'attaque mais la recherche de collisions telle que décrite dans la section 5.2 qui est implémentée. Il suffit de compiler l'exécutable *collision* avec `make` puis de l'exécuter sans options pour que 4 collisions (une par colonne de l'état) soient cherchées et trouvées. Les images des mots en collision sont données ainsi que le nombre de tentatives qu'il a fallu pour les trouver.

8 Conclusion : et aujourd'hui ?

L'implémentation étudiée date de 2002 et a posé les bases de la cryptographie whitebox ; il n'est donc pas étonnant qu'on sache maintenant l'attaquer. Toutefois, il semble qu'aujourd'hui encore, les attaquants ont un coup d'avance.

Comme exposé dans [5], un *catch the flag challenge* fut organisé au CHES 2017 (voir [1]). Les participants étaient invités à proposer des whitebox AES 128 bits, puis à les attaquer. Au final, 94 implémentations furent recueillies. 13 d'entre elles seulement résistèrent plus d'un jour. La plus solide tint 28 jours ; son attaque est détaillée dans [5].

La plupart des implémentations furent défaites par l'une de ces deux attaques :

— Differential Computation Analysis (DCA [4]), qui s'inspire des attaques par analyse de consommation faites notamment sur des cartes à puce. Ici, au lieu d'analyser le courant consommé durant l'exécution de l'algorithme, on regarde les adresses accédées ainsi que leurs contenus et - notamment par de l'analyse de patterns - on réussit à en extraire des informations ;

— Differential Fault Analysis (DFA [10]). L'idée est d'injecter à certains endroits de l'algorithme des valeurs afin d'obtenir des relations impliquant des fragments de clés. On peut donc la voir comme une variante d'attaque différentielle qui détourne l'algorithme afin d'obtenir des relations.

Toutefois, ces échecs n'empêchent pas aujourd'hui l'utilisation de cryptographie whitebox, notamment pour les DRM, car elle reste dissuasive et permet au moins de ralentir l'attaquant.

Enfin, du côté de la recherche, l'existence d'une obfuscation sécurisée d'un algorithme tel qu'AES reste une question ouverte. La discipline a donc encore de beaux jours devant elle.

Références

[1] CHES 2017. Capture the flag challenge - the whibox contest, an ECRYPT white box cryptography competition.

[2] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs, 2010.

[3] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white-box AES implementation, 2005.

[4] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis, hiding your whitebox designs is not enough, 2016.

[5] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation, 2018.

[6] James A. Muir. A tutorial on white-box AES, February 2013.

[7] Yoni De Mulder, Peter Roelse, and Bart Preneel. Revisiting the BGE attack on a white-box AES implementation, 2013.

[8] National Institute of Standards and Technology (NIST). Advanced encryption standard (AES), November 2001.

[9] S. Chow P, Eisen H.Johnson, and P.C. van Oorschot. White-box cryptography and an AES implementation, August 2002.

[10] Eloi Sanfelix, Cristofaro Mune, and Job de Hass. Unboxing the whitebox, pratical attacks against obfuscated ciphers, 2017.