

PX222 IR&C - Carnet de Bord - Carouge Nolan & Mignot Lucas

Important : pensez à télécharger le dossier images pour que le Carnet de Bord affiche les images.

Introduction

Le but de ce projet est d'implémenter **AES** en Haskell puis en C. Ceci nous permettra d'appréhender la filière IR&C via l'utilisation de la *cryptographie*, en combinant mathématiques et informatique.

Nous utiliserons la standardisation de l'algorithme AES via le texte du standard **FIPS 97**.

Séance 1 : 21/03/2023

Durant cette séance, nous avons pris en main notre sujet et découvert le principe d'AES. Nous avons pu commencer à implémenter la partie mathématique en Haskell. Tâches réalisées :

- Définition de **classes de types** pour Anneaux, Corps.
- Implémentation du corps à 2 éléments **F2 = GF(2) = Z/2Z**
- Implémentation (types, +,-,x) des **anneaux des polynômes** à coefficients dans un corps.

Note : En fin de séance, nous avons distribué les tâches à faire dans l'entre-séance.

Distribution des tâches pour la prochaine séance

Nolan	Lucas
Tests sur des polynômes à coefficients dans F2	Implémentation de la division euclidienne, modulo et division entière
Implémentation du calcul modulo un polynôme	Implémentation de l'inverse modulo m
Faire un petit résumé de ce qu'on a compris sur AES	Réflexion sur l'implémentation de GF(2^8) à partir des éléments déjà construits

Nous allons réfléchir à l'implémentation de GF(2^8). Ayant plusieurs questions, nous réaliserons ces tâches en séance.

Séance 2 : 05/04/2023

Premièrement, nous avons pu poser nos questions par rapport à la gestion d'AES, qui semble ne pas être un problème pour l'instant. L'objectif est seulement d'implémenter les "blocs" composant AES128.

Nous avons pu également pu prendre en main Gitlab et Markdown.

Cela nous permettras de clarifier notre code et de maintenir une trace écrite. Nous avons divisé dans plusieurs fichiers nos gitations des mathématiques, corps et anneau.

Nous avons réalisé les fonctions pour AES suivantes :

- byteSub (première partie)
- shiftRows

De plus, nous avons réalisé plusieurs fonctions permettant un affichage des données. Ceci nous sera utile pour la vérification des fonctions.

Note : Nous n'avons pas beaucoup avancé lors de cette séance. Cependant, nous avons pu prendre en main les outils comme Gitlab ou Markdown avec la création de fichiers .md.

Distribution des tâches pour la prochaine séance

Pour la prochaine séance, nous allons essayer de réaliser les 4 fonctions complètes nécessaires à AES.

Nolan	Lucas
byteSub	roundKey
addKey	Affichage d'une matrice

Nous réfléchirons ensemble sur la fonction mixColumn, qui semble être la fonction la plus complexe à réaliser.

Travail à la maison : entre-séances

Nous avons finalement pris un petit peu d'avance sur ce que nous voulions faire. Nous avons implémenté toutes les fonctions nécessaires au fonctionnement d'AES (chiffrage). Nous avons aussi implémenté (sauf la fonction inverse de Sub Byte) toutes les fonctions inverses permettant le déchiffrement. Pour ce faire nous avons réalisé les tâches suivantes :

- **Affichage d'une matrice :** Nous considérons que les matrices sont des matrices 4x4 de polynômes de GF(2^8). Nous avons deux styles d'affichage : un affichage comme tableau de polynôme et un comme un tableau de caractère hexadécimal (rendant la lecture plus agréable).
- **Fonction ByteSub et InvByteSub :** Pour la réalisation de ces deux fonctions nous avons implémenté la structure de matrice. Cela nous a permis de construire la multiplication et l'addition. Nous avons aussi implémenté :
 - Des fonctions permettant de convertir des tableaux de ZZZ en Int et inversement.
 - Une fonction permettant de combler un tableau de ZZZ pour obtenir un tableau de taille 8 (indispensable pour réaliser une multiplication avec une matrice 8*8).
 - Le modulo 2 sur tout un tableau.
 - Une fonction de test réalisant ByteSub et InvByteSub et qui compare le résultat pour voir si le résultat est bien l'identité.
- **Fonction ShiftRows et InvShiftRows :** Les fonctions étant définies très simplement, nous avons pu les définir en quelques lignes. Ici aussi, nous avons défini une fonction de test. Réalisant la fonction ShiftRow puis son inverse pour vérifier si cela vaut bien l'identité.

- **Key Expansion :** Avant de réaliser la fonction Key Expansion, nous avons dû définir la fonction Add Key. Il s'agit simplement d'un Xor entre deux chaînes de 128 Bits. Après cette fonction nous avons dû définir Key Expansion, cela a nécessité une compréhension avancée d'AES ainsi que du rôle des autres fonctions. Nous avons ainsi pu définir la fonction nous permettant d'obtenir la nouvelle clé au round que nous désirerions.
- **MixColumn :** Mix Column fut la fonction la plus compliquée. Nous avons modifié notre structure de matrice pour nous permettre de réaliser un produit matriciel entre des matrices 4*4. Une fois cette étape franchie, la fonction était plutôt simple à réaliser. La fonction MixColumn, semble être fonctionnelle sur des tests simples du PDF. Il sera nécessaire de maximiser les tests.

Le 14/04/23 : Une journée de projet

Cette journée, nous avons travaillé ensemble sur la finalisation d'AES. Le but était de finir toutes les fonctions le même jour. Nous n'avons pas réussi à finaliser la fonction inverse de SubByte. Mais nous avons réalisé toutes les autres fonctions.

Tâche	Réalisation
ByteSub	OK
InvByteSub	Presque OK le 17/04/23
ShiftRows	OK
InvShiftRows	OK
MixColumn	OK
InvMixColumn	OK
RoundKey	OK
AddKey	OK
Affichage d'une matrice	Hexadécimal et Polynomial

Pour l'inverse de Sub Byte, nous calculons l'inverse modulaire de notre matrice grâce au site <https://www.dcode.fr/inverse-matrice>, on peut donc réaliser facilement l'inverse de cette fonction.

Le soir avant la séance, nous avons pu corriger les dernières erreurs. Nous avons tous les blocs qui composent AES de fonctionnel.

Note importante : Nous travaillons la majorité du temps en commun sur un même PC grâce à l'outil de travail à distance "Share Live" de VS Code. Ceci nous permet de modifier le code instantanément et d'être plus efficaces.

Séance 3 : 18/04/2023

Au début de séance, nous travaillons sur l'implémentation de l'algorithme de chiffrage.

Une fois toutes les fonctions d'AES implémentées, on peut simplement suivre la procédure d'AES de la manière suivante :

```

aes :: [[GF]] -> [[GF]] -> [[GF]]
aes matrix key = auxAES 0 matrix
  where auxAES 0 matrix = auxAES 1 (addKey matrix key)
        auxAES 10 matrix = auxAES 11 (addKey (shiftRows $ byteSub matrix)
(keyExpansion 10 key))
        auxAES 11 matrix = matrix
        auxAES round matrix = auxAES (round+1) (addKey (mixColumn $
shiftRows $ byteSub matrix) (keyExpansion round key))

```

Cette fonction nous permet ainsi de suivre la procédure d'AES en suivant le numéro du round actuel. Nous pouvons donc ensuite directement implémenter l'inverse d'AES, en nous rappelant que l'inverse de la composition inverse l'ordre : $\$(f \circ g)^{-1} = g^{-1} \circ f^{-1}$

On obtient ainsi la fonction suivante :

```

invAes :: [[GF]] -> [[GF]] -> [[GF]]
invAes matrix key = auxAES 11 matrix
  where auxAES 0 matrix = matrix
        auxAES 1 matrix = auxAES 0 (addKey matrix key)
        auxAES 11 matrix = auxAES 10 (invByteSub $ invShiftRows(addKey
matrix (keyExpansion 10 key)))
        auxAES round matrix = auxAES (round-1) (invByteSub $ invShiftRows
$ invMixColumn (addKey matrix (keyExpansion (round-1) key)))

```

Nous avons fini l'implémentation du Cipher d'AES comme dans l'*Appendix B p. 33* du FIPS fourni d'AES.

On implémente les inputs et la clé cipher :

Input

32	88	31	e0
43	5a	31	37
f6	30	98	07
a8	8d	a2	34

Cipher Key

2b	28	ab	09
7e	ae	f7	cf
15	d2	15	4f
16	a6	88	3c

On obtient alors :

Output

39	02	dc	19
25	dc	11	6a
84	09	85	0b
1d	fb	97	32

Ceci est conforme avec l'exemple donné dans l'Appendix B.

Lorsque nous testons notre fonction d'inverse d'AES avec l'output et la Cipher Key en entrée, nous obtenons bien l'input voulu.

Notre implémentation d'AES semble être fonctionnelle en Haskell.

Note importante : Il semblerait, selon M. Guisse, que notre implémentation pour les fonctions *SubByte* et *MixColumn* n'est pas convaincante. En effet, il semblerait que l'implémentation par produit matriciel n'était pas la solution souhaitée. Nous devrons donc dans l'entre-séance modifier ces fonctions pour nous permettre d'utiliser un simple produit polynomial.

Début de l'implémentation en C

En séance, nous avons donc pu réfléchir à notre future implémentation en C.

Nous allons essayer de rédiger un cahier des charges, avec des consignes à respecter.

À implémenter	Choix d'implémentation	Justification
GF8	int [8]	La mémoire étant statique, nous n'aurons pas besoin d'ajouter ou de retirer le nombre de cases dans le tableau
Fonctions AES	int [8] [8] vers int [8] [8]	Comme pour haskell, les fonctions prendront en paramètre une matrice 4x4 (ou plus) et renverront une matrice 4x4 (ou plus)

Note importante : Il semblerait, toujours selon Mr Guisse, que l'implémentation des fonctions ByteSub et InverseByteSub doit être faite en utilisant la SBOX.

Distribution du travail pour les prochaines séances :

Comme nous avons été prévenus à la fin de la séance que notre implémentation haskell était mauvaise, nous travaillerons sur la correction de notre code.

Nolan	Lucas
ByteSub	Modification du Code
InvByteSub	ShiftRows/Inv

L'objectif est donc de modifier notre code est de commencer l'implémentation d'AES en C. Cette fois-ci de manière plus succincte.

Travail à la maison : entre-séances

Nous avons réussi à modifier la fonction ByteSub pour ne plus traiter des produits matriciels, mais bien de simples fonctions. Nous obtenons la fonction suivante :

```
auxByteSub :: GF -> GF
auxByteSub p = addPoly (polyMod (multPoly [Z1,Z1,Z1,Z1,Z1,Z0,Z0,Z0]
(inverse p)) [Z1,Z0,Z0,Z0,Z0,Z0,Z0,Z1]) [Z1,Z1,Z0,Z0,Z0,Z1,Z1,Z0]

byteSub :: [[GF]] -> [[GF]]
byteSub [l1,l2,l3,l4] = [map auxByteSub l1, map auxByteSub l2, map
auxByteSub l3, map auxByteSub l4]
```

Nous gardons l'utilisation d'une fonction externe pour nous simplifier les tests. Avec l'auxiliaire on peut réaliser les bytesSub indépendamment des tableaux et donc un par un.

Nos tests sont les mêmes et sont validés.

Nous avons également essayé de modifier MixColumn pour nous permettre d'outrepasser l'utilisation de multiplications matricielles. Cependant, nous n'avons pas réussi à l'implanter pour l'instant.

Nous avons modifié la fonction BytesSub selon les dires de M. Guisse. Il nous reste la fonction MixColumn à modifier. En effet, nous n'avons pas réussi à le faire.

Le 2 mai 2022 :

À ce jour, nous avons réalisé les fonctions SubBytes et ShiftRows ainsi que leurs inverses en C.

Nous avons choisi d'implémenter l'octet par un `uint8_t` qui est un entier allant de 0 à 255. Ceci nous permettra de convertir facilement en polynômes pour traiter certaines fonctions. On définit aussi un type `matrix` qui est un tableau de tableau de 4x4 d'éléments du type octets, précédemment cité. On a :

```
#define octet uint8_t
typedef octet matrix [4][4];
```

Ce choix d'implémentation `uint8_t` nous permet également de ne pas avoir à nous soucier des affichages. En effet, on peut facilement afficher une matrice de la manière suivante, en hexadécimal en forçant l'affichage de deux nombres (ex : `0A au lieu de A`) :

```
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++)
        printf("%02x ", matrix[i][j]);
    printf("\n");}
```

SubByte et son inverse

Pour les fonctions *SubBytes* et *InvSubBytes*, nous avons pris le choix de l'implémentation de la Sbox et RSBox, comme mentionné par M. Guisse. Ces choix permettent entre autres de ne pas avoir à coder l'inverse et donc la division euclidienne. Ces boxes nous permettent de définir simplement la fonction ByteSub et InvByteSub :

```
static const octet sbox[256] = {  
  
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab,  
    , 0x76,  
  
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72  
    , 0xc0,  
  
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31  
    , 0x15,  
  
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2  
    , 0x75,  
  
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f  
    , 0x84,  
  
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58  
    , 0xcf,  
  
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f  
    , 0xa8,  
  
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3  
    , 0xd2,  
  
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19  
    , 0x73,  
  
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b  
    , 0xdb,  
  
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4  
    , 0x79,  
  
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae  
    , 0x08,  
  
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b  
    , 0x8a,  
  
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d  
    , 0x9e,  
};
```

```

0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28
,0xdf,
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb
,0x16
};

```

Ainsi, on sait grâce à la sbox que `sbox[0] = 0x63`.

On peut donc facilement trouver la fonction `SubBytes` en récupérant la valeur dans la sbox pour chacune des valeurs du tableau. On a :

```

void sub_bytes (matrix state) {
    for(int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++)
            state[i][j] = sbox[state[i][j]];
}

```

Nous ne détaillerons ici pas l'inverse de `SubByte` qui fonctionne de la même manière avec la RSbox plutôt que la SBox.

ShiftRows et inverse

Pour `ShiftRow` et son inverse, on les réalise de la manière suivante :

```

void shift_rows (matrix state) {
    matrix tmp; copy_matrix(state, tmp);

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            state[i][j] = tmp[i][(j+i)%4];
}

```

Nous passons par une matrice temporaire pour nous permettre de traiter les informations simplement et sans avoir à prendre de risque de pertes de données. Notons que nous aurions pu passer simplement par un octet de réserve et non d'une matrice, mais que nous avons trouvé ça plus simple et plus visuel.

La fonction `copy_matrix` fait partie de plusieurs fonctions utilitaires que nous avons réalisées. Elle permet de copier la matrice `state` dans la matrice `tmp`

Nous avons les fonctions utilitaires suivantes (dont le nom est assez explicite pour ne pas donner de détails) :

```

void print_matrix (matrix state)
int egals_matrix (matrix a, matrix b)

```

```
void copy_matrix (matrix source, matrix dest)
void clean_matrix (matrix state)
```

Tests

Pour réaliser nos tests, on écrit une fonction colorée nous permettant de valider ou non le résultat attendu. Cette fonction est la suivante :

```
#define red "\x1B[31m"
#define green "\x1B[32m"
#define reset "\x1B[0m"

...
void test_print (int test, char* test_name) {
    if (test)
        printf("Test %s %sOK%s\n", test_name, green, reset);
    else
        printf("Test %s %sNON OK%s\n", test_name, red, reset);
}
```

Elle nous permettra, par la suite, de valider nos fonctions précédemment codées assez simplement.

On obtient que nos fonctions *sub_bytes*, *inv_sub_bytes*, *shift_rows* et *inv_shift_rows* sont toutes correctes.

N.B. Toutes les modifications / ajouts ont été réalisées à deux. Toujours en utilisant l'outil Live Share.

Gestion de fichiers

Pour manipuler facilement les octets de nos fichiers, on utilise l'application MacOS **Hex Friends** permettant un affichage et la modification rapide et efficace des octets d'un fichier.

Après plusieurs tests, nous avons conclu qu'il était plus simple de gérer les fichiers de la manière suivante :

- Lire le fichier d'entrée
- Créer une matrice de 16 octets (4x4)
- Manipuler cette matrice
- Écrire les octets dans le fichier de sortie

Ce protocole nous permettra ainsi de manipuler tous nos fichiers, et de pouvoir même tester nos fonctions directement dans le cadre d'un fichier (étant donné que les octets pourront être vus avec l'outil Hex Friend).

La manipulation de la matrice étant variable (du moins pour nos tests), nous avons décidé de partir sur l'utilisation de pointeurs de fonction pour avoir une fonction modulaire nous permettant une manipulation simple.

Après plusieurs essais, on obtient la fonction suivante :

```

int file_operation (char* input_name, char* output_name, void (*p_process)
(matrix state)) {
    FILE* input = fopen(input_name, "r");
    FILE* output = fopen(output_name, "w");

    matrix data_matrix = {};// Initialise la matrice à 0

    if(input == NULL || output == NULL) {
        perror("fopen");
        return 0;
    }

    int counter = 0;
    int c;

    while((c = fgetc(input)) != EOF) {
        data_matrix[counter%4][counter/4] = c;
        if(counter == 15) {
            p_process(data_matrix);
            write_matrix(output, data_matrix);
            clean_matrix(data_matrix);
            counter = 0;
        } else
            counter++;
    }

    // Si on est pas aller jusqu'au bout du fichier par un bloc de 128
    bits (mais moins)
    if (counter != 0) {
        p_process(data_matrix);
        write_matrix(output, data_matrix);
    }

    fclose(output);
    fclose(input);
    return 1;
}

```

Cette fonction sera la base de la lecture, la manipulation et l'écriture des données.

En comptant jusqu'à 16 (0->15), on complète la matrice. Cette fonction est une fonction qui travaille directement sur le type matrix et le modifie (comme c'est le cas avec nos fonctions *mix_columns...*).

Nous avons dû ajouter le code suivant :

```

if (counter != 0) {
    p_process(data_matrix);
    write_matrix(output, data_matrix);
}

```

À la fin du while, étant donné que si le dernier bloc dans le fichier était composé de 2 octets, par exemple, nous n'aurions pas pris en compte ces deux octets. Ceci nous permet donc de vérifier si le dernier bloc d'octet n'a pas été traité et dans ce cas de le faire.

Notons que la matrice de données *data_matrix* est remise à 0 à chaque fin de procédure.

On peut ainsi tester simplement notre code grâce à l'appel suivant :

```
file_operation("files_tests/test_BS1", "files_tests/test_BS1_output",  
&sub_bytes);
```

On obtient en écrivant les octets dans un fichier grâce à Hex Friends :

	test_BS1		test_BS1_output
0	19A09AE9	0	D4E0B81E
4	3DF4C6F8	4	27BFB441
8	E3E28D48	8	11985D52
12	BE2B2A08	12	AEF1E530
16		16	

On obtient bien le résultat attendu, dans le fichier *test_BS1_output* de sortie.

Notons que si nous ajoutons 19 à la fin du fichier `test_BS1`, on a :

0	19A09AE9	0 D4E0B81E
4	3DF4C6F8	4 27BFB441
8	E3E28D48	8 11985D52
12	BE2B2A08	12 AEF1E530
16	19	16 D4636363
		20 63636363
		24 63636363
		28 63636363
		32

Ainsi, nous ne savons pas si ce résultat est censé être juste bien que le résultat est cohérent (car le tableau est complété par des octets ayant tous comme valeur 0).

On peut aussi tester `shift_rows` de la même manière. On obtient :

0	41624364	AbCd
4	61426344	aBcD
8	41426364	ABcd
12	61624344	abCD
		0 41624364 AbCd
		4 42634461 BcDa
		8 63644142 cdAB
		12 44616243 DabC

On peut également voir directement le contenu de notre fichier texte. On observe bien le décalage des octets et du fichier.

Récapitulatif

Voici un tableau récapitulatif de ce que nous avons, pour l'instant, en C :

Fonction	Validation
Structures	OK (octet, matrix)
Utilitaires (affichages, égalités, copies des matrices...)	OK
ByteSub / Inv	OK
ShiftRows / Inv	OK
AddKey	OK
Tests	OK (peut faire plus...)
Opérations sur les fichiers (lecture, process, écriture)	OK

Séance 4 : 03/05/23

Gestion de la clé

Nous allons essayer de travailler avec une clé. Pour ce faire, on choisit que la clé soit stockée dans un fichier. Ce fichier sera mis en paramètre de l'appelle de notre fichier aes. On définit ainsi l'appel de la manière suivante : `./aes <file_key> [input_files]`.

Se basant pour l'instant sur AES128, on va stocker notre clé de 128 bits (ie. 16 octets) dans notre type *matrix*, de la manière suivante. La lecture du fichier se fait donc sur les 16 premiers octets, et ne tient pas compte des suivants. Si la clé n'est pas de taille suffisante, notre fonction renvoie 0 et nous pouvons récupérer cette erreur.

On peut facilement tester notre fonction :

```
if (argc >= 2)
    if(!get_key_128(argv[1], key)) {
        printf("Erreur : la longueur de la clé doit être de minimum 128
bits (16 octets).\n");
        return 0;
    }
print_matrix(key);
```

En écrivant deux clés dans deux fichiers distincts :

key (taille suffisante) lil_key (taille non suffisante)

Ceci est une clé

Ceci e

			key
00	43656369	Ceci	
04	20657374	est	
08	20756E65	une	
0C	20436C65	Cle	
10	20		

			lil_key
00	43656369	Ceci	
04	2065		e
08			
0C			
10			

On obtient :

```
./aes key
43 20 20 20
65 65 75 43
63 73 6e 6c
69 74 65 65

./aes lil_key
Erreur : la longueur de la clé doit être de minimum 128 bits (16 octets).
```

On peut donc récupérer notre clé.

On teste d'écrire la clé *Cipher Key* de la doc, dans un fichier binaire **2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C**.

On a :

```
./aes cipher_key
2b 28 ab 09
7e ae f7 cf
15 d2 15 4f
16 a6 88 3c
```

Ce qui est cohérent avec la doc.

ShiftRows

Nous avons commencé par relire la définition de la fonction. Nous avons ensuite décidé de réaliser une fonction permettant de calculer la clé en fonction d'un round.

```
void key_expansion_round_n(matrix key, int n);
```

Nous avons par la suite déduit que d'appeler cette fonction serait très couteux. Nous avons définit une fonction qui permet de calculer toutes les clés d'un coup. Cela utilise la clé précédemment calculer, cela permet d'économiser des ressources. La fonction est de la forme :

```
void key_expansion(matrix key, matrix all_key[10]);
```

Nous passons en paramètre la variable all_key, cela permet de la modifier directement est donc de ne pas renvoyer un pointeur de pointeur de pointeur. En effet all_key est un tableau de tableau de tableau.

Mix Column

Pour cette fonction, nous devons implémenter le calcul modulo le polynôme irréductible. Ce polynôme est 283 en base 10.

On réalise le code suivant :

```
int dozens (unsigned int value) {
    int counter = 0;
    while (value) {
        counter++;
        value >= 1;
    }
    return counter;
}

octet mod_irr_gf (unsigned int value) {
    unsigned int reduct = value;
    while (reduct > 255) {
        reduct ^= 283 << (dozens(reduct) - 9);
    }
    return reduct;
}
```

Ce code permet, grâce aux décalages de bits (ie multiplications et divisions par 2), de trouver le modulo simplement.

On a :

```
printf("%d\n", mod_irr_gf(35095));
```

Et on a :

251

On obtiendrait donc que **35095 % pirr = 251**.

Vérifions ce modulo en Haskell.

```
|ghci> p = [Z1,Z1,Z1,Z0,Z1,Z0,Z0,Z0,Z1,Z0,Z0,Z1,Z0,Z0,Z1]
|ghci> gftoInt p
35095
|ghci> gftoInt $ polyMod p irrGF
251
```

Ceci correspond à nos attentes.

Nous avons donc un moyen d'implémenter tous les calculs modulaires entre les polynômes, vu pour nous comme des octets.

Nous avons presque réussi à modifier la fonction MixColumn en haskell pour tenir aux spécifications de M. Guisse. Il manque encore quelques détails, mais nous avançons.

Distribution du travail pour les prochaines séances :

La prochaine séance étant la dernière. Il serait préférable d'avoir corrigé le code en haskell. De plus comme il nous manque une seule fonction en C. Nous pourrons l'implémenter pour compléter AES.

L'objectif est donc de finir le C, ainsi que de finir le Haskell.

Nous avons donc beaucoup d'avance. Cette avance nous permettra lors de la semaine de projet de finaliser le tout correctement. Et de pouvoir proposer un compte rendu solide.

Travail à la maison : entre-séances

AES-128

Étant donné que nous allons pouvoir soit chiffrer, soit déchiffrer un fichier, on prend le prototype d'appel suivant : **./aes <crypt|decrypt> <file_key> [input_files]**

On a le code suivant :

```
if (argc >= 4) {
    if(!get_key_128(argv[2], key)) {
        printf("Erreur : la longueur de la clé doit être de minimum 128
bits (16 octets).\n");
        return 0;
    } else
        for (int i = 3; i < argc; i++) {
            double start = (double) clock() / CLOCKS_PER_SEC;
            if (argv[1][0] == 'd' || argv[1][0] == 'D') { // decrypte
                char* output = combine_strings(argv[i], "_decrypted");
                if (file_operation(argv[i], output, &aes_inverse))
                    printf("Fichier %s déchiffré dans le fichier %s avec
succès en %fs.\n", argv[i], output, (((double)clock()) / (double)
CLOCKS_PER_SEC) - start);
                else
                    printf("Erreur durant le déchiffrage de %s\n", argv[i]);
            }
        }
}
```

```

        free(output);
    } else { //chiffre
        char* output = combine_strings(argv[i], "_encrypted");
        if (file_operation (argv[i], output, &aes))
            printf("Fichier %s chiffré dans le fichier %s avec succès
en %fs.\n", argv[i], output, (((double)clock()) / (double) CLOCKS_PER_SEC)
- start);
        else
            printf("Erreur durant le chiffrage de %s\n", argv[i]);
        free(output);
    }
}
} else
    printf("Usage : ./aes <crypt|decrypt> <file_key> [input_files]\n");

```

Ce code nous permet de facilement encoder plusieurs fichiers à la fois avec une clé. Nous allons tester ce code pour savoir si notre algorithme est juste et performant. Pour mesurer sa performance, on mesure le temps process grâce à la fonction `clock`.

Maintenant que tous nos blocs ont été réalisés, nous pouvons tester notre implémentation d'AES-128.

Étant donné que nous n'avons eu aucune information sur la gestion de la clé et des fichiers, on part du principe que notre clé est dans un fichier entré en paramètre de notre fonction.

Nous allons chiffrer, puis déchiffrer le fichier `paroles.txt` suivant avec la clé `Ceci est une Cle`:

Tourner le temps à l'orage
 Revenir à l'état sauvage
 Forcer les portes, les barrages
 Sortir le loup de sa cage
 Sentir le vent qui se déchaîne
 Battre le sang dans nos veines
 Monter le son des guitares
 Et le bruit des motos qui démarrent
 Il suffira d'une étincelle
 D'un rien, d'un geste
 Il suffira d'une étincelle
 Et d'un mot d'amour
 Pour
 Allumer le feu
 Allumer le feu
 Et faire danser les diables et les dieux
 Allumer le feu
 Allumer le feu
 Et voir grandir la flamme dans vos yeux
 Allumer le feu
 Laisser derrière toutes nos peines
 Nos haches de guerre, nos problèmes
 Se libérer de nos chaînes
 Lâcher le lion dans l'arène
 Je veux la foudre et l'éclair

L'odeur de poudre, le tonnerre
 Je veux la fête et les rires
 Je veux la foule en délire
 Il suffira d'une étincelle
 D'un rien, d'un contact
 Il suffira d'une étincelle
 D'un peu de jour
 Pour
 Allumer le feu
 Allumer le feu
 Et faire danser les diables et les dieux
 Allumer le feu
 Allumer le feu
 Et voir grandir la flamme dans vos yeux
 Allumer le feu
 Il suffira d'une étincelle
 D'un rien, d'un geste
 Il suffira d'une étincelle
 D'un mot d'amour pour
 Allumer le feu
 Allumer le feu
 Et faire danser les diables et les dieux
 Allumer le feu
 Allumer le feu
 Et voir grandir la flamme dans vos yeux
 Oh, allumer le feu
 Allumer le feu
 Et faire danser, les diables et les dieux
 Allumer le feu (allumer le feu)
 Allumer le feu (allumer le feu)

On lance (fichier .txt de 1390 octets) : `./aes c key aes_tests/paroles.txt`

Fichier aes_tests/paroles.txt chiffré dans le fichier aes_tests/paroles.txt_crypted avec succès en 0.00213s.

On obtient :



The screenshot shows three hex editors side-by-side. The first editor on the left contains the lyrics in French. The second editor in the middle contains the encrypted binary data. The third editor on the right shows the file size as 0 bytes out of 1.4 kilobytes. Each editor has controls for selecting data (Unsigned Int, Signed Int), and the bottom of each editor window displays the file size.

File	Content (Hex View)	Size
paroles.txt	French lyrics in ASCII	1.4 kilobytes
paroles.txt_crypted	Encrypted binary data (mostly non-printable characters)	0 bytes

qui est un fichier de 1392 octets. Ceci est logique étant donné que 1390 octets permettent de faire 86 blocs de 16 octets et il reste 14 octets. Il faut donc compléter avec deux octets 00 00 la fin du fichier de départ.

On réalise le déchiffrage du fichier obtenu : `./aes c key aes_tests/paroles.txt_crypted`

Fichier `aes_tests/paroles.txt_crypted` déchiffré dans le fichier `aes_tests/paroles.txt_crypted_decrypted` avec succès en 0.00349.

On obtient :

paroles.txt_crypted_decrypted			
000	546F7572	6E657220	Tourner
008	6C652074	656D7073	le temps
010	20C3A020	6C276F72	.. l'or
018	6167650A	52657665	age Reve
020	6E697220	C3A0206C	nir .. l
028	27C3A974	61742073	'.?tat s
030	61757661	67650A46	auvage F

(select some data)
 (select some data)

0 out of 1392 bytes

On observe également qu'en fin de fichier, on a bien un rajout de deux octets nuls :

The screenshot shows a hex editor window with the title "paroles.txt_crypted_decrypted". The main pane displays a grid of hex values and their corresponding ASCII representation. The text visible includes "r le", "feu", and ")". Below the main pane, there are two sets of dropdown menus for data types: "Unsigned Int" and "Signed Int", both set to "le, dec". To the right of these menus are four buttons: two minus signs and two plus signs. At the bottom of the window, a status bar displays "0 out of 1392 bytes".

Et on retrouve bien le fichier :

```
Turner le temps à l'orage
Revenir à l'état sauvage
Forcer les portes, les barrages
Sortir le loup de sa cage
Sentir le vent qui se déchaîne
Battre le sang dans nos veines
Monter le son des guitares
Et le bruit des motos qui démarrent
Il suffira d'une étincelle
D'un rien, d'un geste
Il suffira d'une étincelle
Et d'un mot d'amour
Pour
Allumer le feu
Allumer le feu
Et faire danser les diables et les dieux
Allumer le feu
Allumer le feu
Et voir grandir la flamme dans vos yeux
Allumer le feu
Laisser derrière toutes nos peines
Nos haches de guerre, nos problèmes
Se libérer de nos chaînes
```

Lâcher le lion dans l'arène
Je veux la foudre et l'éclair
L'odeur de poudre, le tonnerre
Je veux la fête et les rires
Je veux la foule en délire
Il suffira d'une étincelle
D'un rien, d'un contact
Il suffira d'une étincelle
D'un peu de jour
Pour
Allumer le feu
Allumer le feu
Et faire danser les diables et les dieux
Allumer le feu
Allumer le feu
Et voir grandir la flamme dans vos yeux
Allumer le feu
Il suffira d'une étincelle
D'un rien, d'un geste
Il suffira d'une étincelle
D'un mot d'amour pour
Allumer le feu
Allumer le feu
Et faire danser les diables et les dieux
Allumer le feu
Allumer le feu
Et voir grandir la flamme dans vos yeux
Oh, allumer le feu
Allumer le feu
Et faire danser, les diables et les dieux
Allumer le feu (allumer le feu)
Allumer le feu (allumer le feu)

??

Nous pouvons nous consacrer à optimiser notre code. Pendant la semaine de PX, nous essayerons d'implémenter AES-256.

Notons qu'il faudra essayer de trouver un code chiffré et de le déchiffrer pour voir si notre code est bien juste et si le processus de crypt/decrypt ne compense pas leurs erreurs.

Modifications des entrées

Utiliser un fichier comme entrée de clé nous pose problème (sécurité...). Nous avons donc implémenter un processus afin de demander à l'utilisateur une clé de chiffrement. Cette clé sera demandée et pourra être soit affichée soit cachée. L'implémentation actuelle est pour une clé de 16 octets.

Afin de récupérer le texte tappé par l'utilisateur sans l'afficher, on utilise une fonction trouvée sur [ce site](#) :

```
int getch () {
    int ch;
    struct termios oldt, newt;
```

```

        tcgetattr(STDIN_FILENO, &oldt);
        newt = oldt;
        newt.c_lflag = newt.c_lflag & ~(ICANON|ECHO);
        tcsetattr(STDIN_FILENO, TCSANOW, &newt);
        ch = getchar();
        tcsetattr(STDIN_FILENO, TCSANOW, &oldt);

        return ch;
    }
}

```

Puis, on peut implémenter directement notre fonction pour récupérer la clé :

```

printf("Entrez la clé de chiffrement (sans accents) : ");
char c, i = 0;
while (i < 16) {
    c = getch();
    if (c == 127 && i > 0) {
        printf("\nEntrez la clé de chiffrement (sans accents) : ");
        i--;
        for (int j = 0; j < i; j++)
            printf("%c", key[j%4][j/4]);
    } else if (c >= ' ' && c != 127) {
        if (show_key) // On n'affiche pas les caractères
            printf("%c", c);
        else
            printf("*");
        key[i%4][i/4] = c;
        i++;
    }
}

```

Pour tester nos fonctions, on va encoder puis décoder de deux manières différentes :

```

luka.mig@MBP-de-Lucas AES C % ./aes
Usage : ./aes <crypt|decrypt> <show|hide|file_key> [input_files]

```

Raccourcis :

- c = crypt
- d = decrypt
- s = show
- h = hide

```

luka.mig@MBP-de-Lucas AES C % ./aes c s aes_tests/paroles.txt
aes_tests/esisar.jpeg

```

Entrez la clé de chiffrement (sans accents) : Ceci estt

Entrez la clé de chiffrement (sans accents) : Ceci est une Cle

Fichier aes_tests/paroles.txt chiffré dans le fichier

aes_tests/paroles.txt_crypted avec succès en 0.00186s.

Fichier aes_tests/esisar.jpeg chiffré dans le fichier

```
aes_tests/esisar.jpeg_crypted avec succès en 0.01831s.
```

```
luka.mig@MBP-de-Lucas AES C % ./aes d key aes_tests/paroles.txt_crypted  
Fichier aes_tests/paroles.txt_crypted déchiffré dans le fichier  
aes_tests/paroles.txt_crypted_decrypted avec succès en 0.00427.
```

```
luka.mig@MBP-de-Lucas AES C % ./aes d h aes_tests/esisar.jpeg_crypted  
Entrez la clé de chiffrement (sans accents) : *****  
Fichier aes_tests/esisar.jpeg_crypted déchiffré dans le fichier  
aes_tests/esisar.jpeg_crypted_decrypted avec succès en 0.04473.
```

On retrouve bien notre image de départ et nos paroles en utilisant les différents appels de clé.



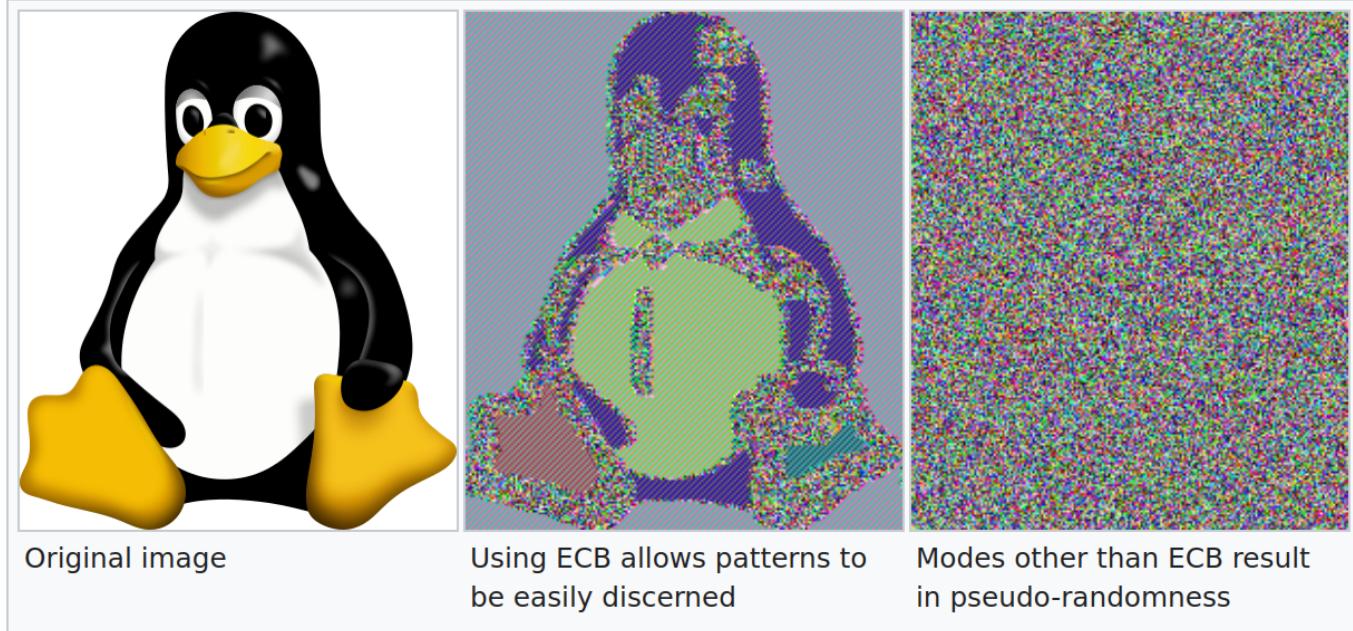
On observe également que notre boucle pour chiffrer tout les fichiers entrés est fonctionnelle.

Séance 5 : 17/05/23

Nous avons trouvé d'où vient le problème de Mix Column en Haskell pour ne pas utiliser une multiplication matricielle. Celle-ci vient de notre division euclidienne qui ne sait pas faire de divisions de listes de GF256. Nous avons essayé de corriger ce problème, mais nous n'avons pour l'instant toujours pas réussi.

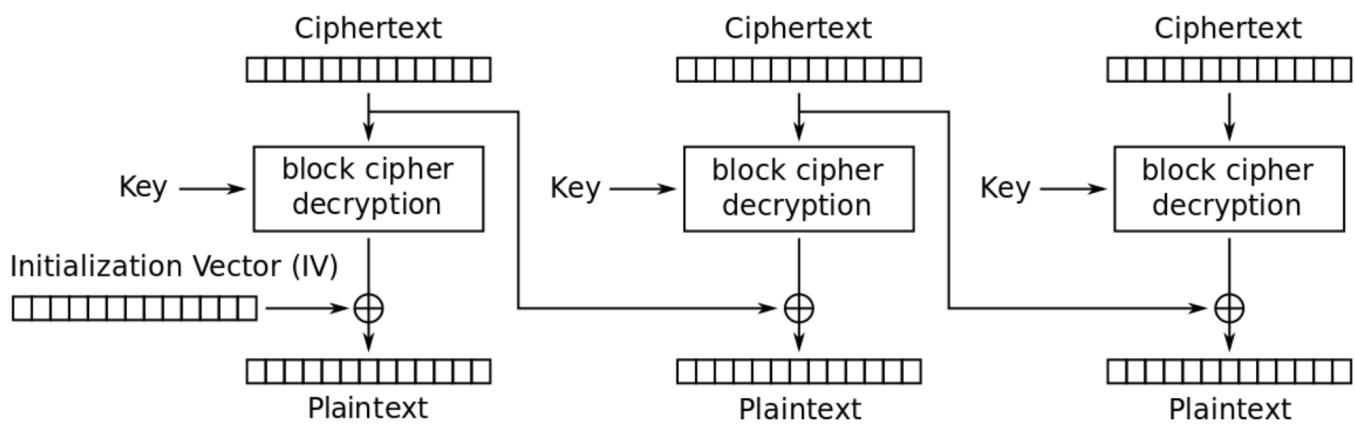
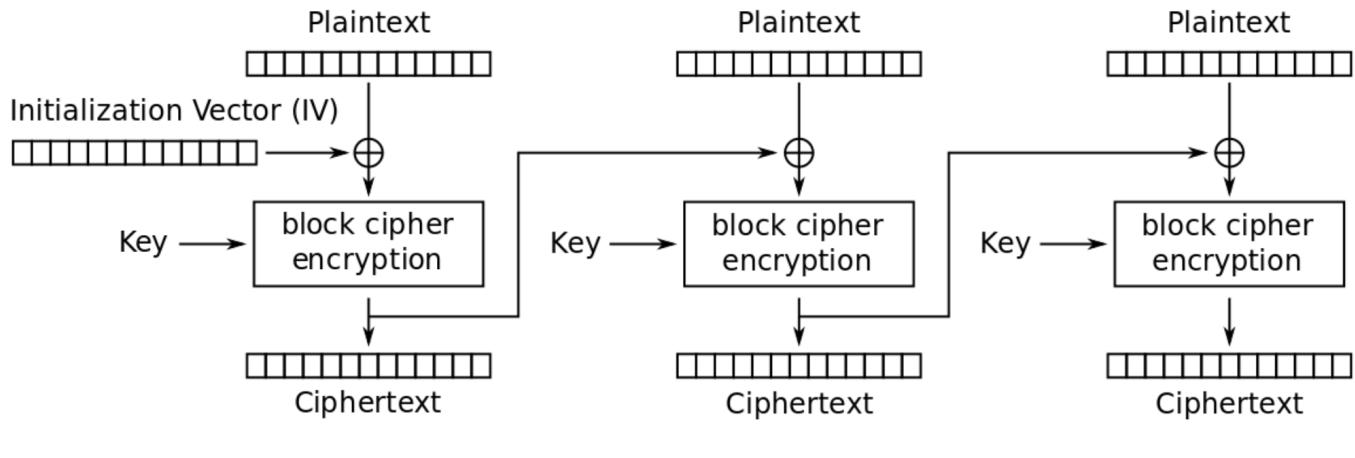
Quant-à AES128 en C, nous avons pu montrer à l'enseignant comment fonctionnait notre système. Étant donné que nous réalisons simplement les blocs d'AES Nous allons maintenant pouvoir implémenter un protocole de chiffrage complet, en utilisant le protocole *Cipher Block Chaining* (CBC). Ceci permettra de ne pas chiffrer de la même manière deux blocs identiques dans le fichier.

Exemple :



On observe bien que l'utilisation d'un protocole CBC par rapport à ECB (notre protocole actuel) que les données sont complètement cachées et on ne reconnaît plus rien de l'image de départ.

CBC fonctionne de cette manière :



Cipher Block Chaining (CBC) mode decryption

Pour l'instant, on ignore le vecteur d'initialisation et considérons qu'il est à 0.

On programme donc notre CBC de la manière suivante :

```
extern matrix key;
extern matrix mem_cbc;

void aes_cbc (FILE* output, matrix state) {
    xor_state(state, mem_cbc); // xor
    cipher(state, key); // crypt
    write_matrix(output, state); // écriture
    copy_matrix(state, mem_cbc); // prochain vecteur
    clean_matrix(state);
}

void aes_cbc_inverse (FILE* output, matrix state) {
    static matrix tmp_cbc_inv;
    copy_matrix(state, tmp_cbc_inv); // prochain vecteur
    cipher_inverse(state, key); // crypt inverse
    xor_state(state, mem_cbc); // xor avec vecteur
    write_matrix(output, state); // écriture
    copy_matrix(tmp_cbc_inv, mem_cbc); // prochain vecteur
    clean_matrix(state);
}
```

Ceci nous permet de facilement réaliser le protocole CBC. Ces fonctions seront appelées directement par la gestion de fichier. Nous avons donc du modifier notre fonction de gestion des fichiers afin de prendre en paramètre le fichier de sortie.

Notre vecteur d'initialisation sera donc directement `mem_cbc`. Nous ignorons, pour l'instant, la gestion de ce vecteur et considérons qu'il vaut 0.

On modifie l'appel de fonction de la manière suivante, pour préciser si nous voulons encrypter par la manière ECB ou CBC : `./aes <crypt|decrypt> <ecb|cbc> <show|hide|file_key> [input_files]`.

Pour simplifier notre code d'appel, on modifie notre gestion des entrées dans le main de la manière suivante :

```
int succes;
for (int i = 4; i < argc; i++) {
    double time = (double) clock() / CLOCKS_PER_SEC;

    int decrypt = argv[1][0] == 'd';
    char* output = combine_strings(argv[i],
    decrypt?"_decrypted":"_crypté");

    if (decrypt) // decrypte
        succes = file_operation (argv[i], output, (argv[2][0] == 'e')?
(&aes_ecb_inverse):(&aes_cbc_inverse));
```

```

    else //chiffre
        succes = file_operation (argv[i], output, (argv[2][0] == 'e')?
(&ampaes_ecb):(&ampaes_cbc));

        time = (((double)clock()) / (double) CLOCKS_PER_SEC) - time;

        if (succes)
            printf("Fichier %s %s (%s) dans %s avec succès (%.5fs).\n",
argv[i], decrypt?"déchiffré":"chiffré", (argv[2][0] == 'e')?"ECB":"CBC",
output, time);
        else
            printf("Erreur durant le %s de %s\n",
decrypt?"déchiffrage":"chiffrage", argv[i]);

        free(output);
    }
}

```

On obtient (méthode CBC) :

	test	test_crypted	test_crypted_decrypted
0	20202020	0 67A6B7B5 g...	0 20202020
4	20202020	4 B2DAE7CA	4 20202020
8	20202020	8 88F38E52 ...R	8 20202020
12	20202020	12 0F2A8A4B *.K	12 20202020
16	20202020	16 9825E63D .%.=	16 20202020
20	20202020	20 D8B1D541 ...A	20 20202020
24	20202020	24 C11FB775 . .u	24 20202020
28	20202020	28 ED15222B . "+	28 20202020
32		32	32

Unsigned Int ↴ [le, dec] 42702100946941297375 [- +]
Signed Int ↴ [le, dec] 42702100946941297375 [- +]

16 bytes selected at offset 16 out of 32 bytes

Unsigned Int ↴ [le, dec] 57333786656048888772 [- +]
Signed Int ↴ [le, dec] 57333786656048888772 [- +]

16 bytes selected at offset 16 out of 32 bytes

Unsigned Int ↴ [le, dec] 42702100946941297375 [- +]
Signed Int ↴ [le, dec] 42702100946941297375 [- +]

16 bytes selected at offset 16 out of 32 bytes

On voit que deux blocs identiques ne sont donc plus chiffrer de la même manière. En effet, avec la méthode ECB, on aurait obtenu :

	test	test_crypted	test_crypted_decrypted
0	20202020	0 67A6B7B5	0 20202020
4	20202020	4 B2DAE7CA	4 20202020
8	20202020	8 88F38E52	8 20202020
12	20202020	12 0F2A8A4B	12 20202020
16	20202020	16 67A6B7B5	16 20202020
20	20202020	20 B2DAE7CA	20 20202020
24	20202020	24 88F38E52	24 20202020
28	20202020	28 0F2A8A4B	28 20202020
32		32	32
	Unsigned Int ↴ le, dec 42702100946 Signed Int ↴ le, dec 42702100946	Unsigned Int ↴ le, dec 10040948972! Signed Int ↴ le, dec 10040948972!	Unsigned Int ↴ le, dec 42702100946941297375 Signed Int ↴ le, dec 42702100946941297375

On observe bien que deux blocs identiques sont cryptés de la même manière.

Il serait bien de créer une fonction permettant de ne pas modifier l'entête fichier d'un fichier .jpeg, permettant ainsi de voir l'efficacité de notre code CBC.

Nous essayerons d'implémenter AES en version 256 bits durant la semaine de PX. Nous devons également essayer de modifier notre code en Haskell avant ces séances, pour implémenter Mix Column de la bonne manière.

Pour réduire le temps de chiffrage et de déchiffrage, on choisit de calculer et stocker les 256*256 multiplications possibles pour GF. On stocke le résultat dans un tableau de tableau permettant de récupérer facilement les multiplications.

```
void calculate_mult() {
    for (int i = 0; i < 256; i++)
        for (int j = 0; j < 256; j++)
            mult_calculated[i][j] = mult_gf(i, j);
}

octet mult_calc_gf (octet a, octet b) {
    return mult_calculated[a][b];
}
```

La fonction *calculate_mult* est appelée une fois au début du main.

Sans utiliser notre fonction qui pré-calcule les 65655 multiplications possibles, on chiffre un fichier de 1 go en 23 minutes. Ce temps est long.

Avec l'utilisation de *mult_calc_gf*, ce temps passe à 12 minutes 40 en utilisant la méthode CBC.

Le fichier faisant 1073741824 octets, on passe d'un temps de 778 Ko/s à 1,4 Mo/s. Notons que ce temps est un temps d'instruction processeur et n'est pas forcément représentatif du temps réel.

Notons également que nous calculons ici les 65655 multiplications modulo notre irréductible mais nous n'utilisons seulement que quelques valeurs multiplicatives, soit les multiplications par : 0x0e, 0x0b, 0x0d, 0x09, 0x02, 0x03, 0x01. Nous pouvons donc nous contenter de calculer seulement 6 tables de multiplications, soit 1536 opérations. Ceci n'affectera pas notre temps de chiffrage et décryptage mais permettra d'utiliser moins de mémoire avec une table plus légère.

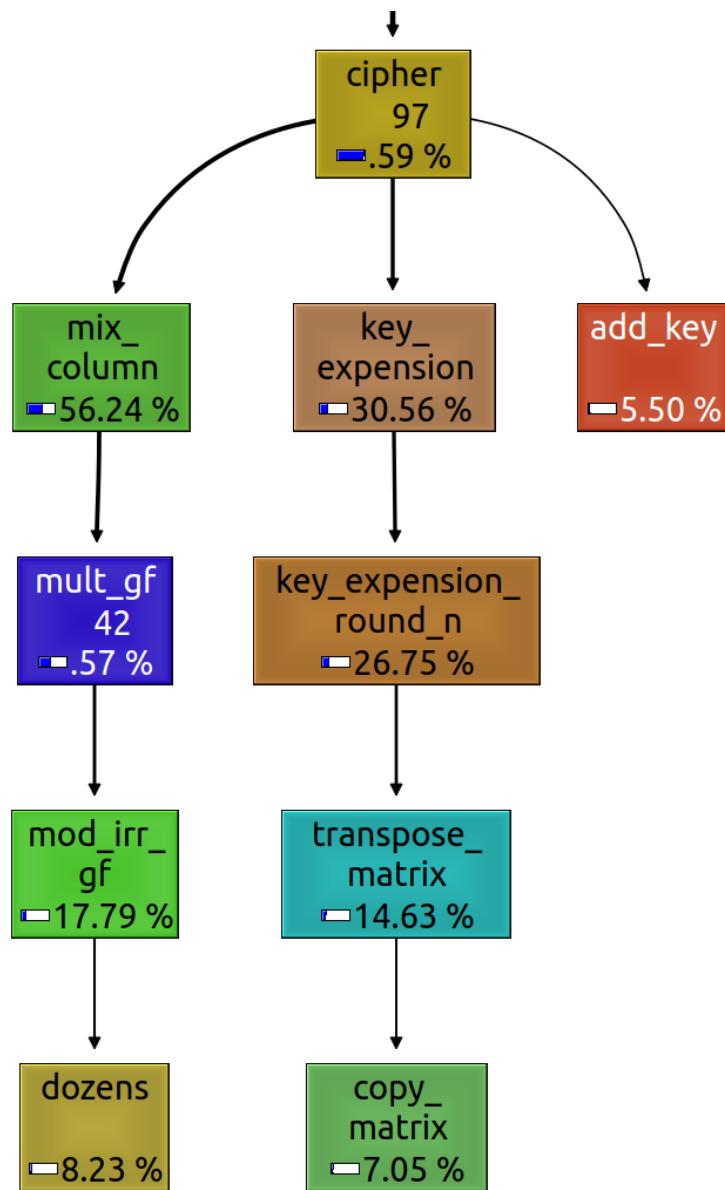
Semaine de PX

Lundi 5 juin 2023

Nous avons commencé la semaine de projet par écouter les nouvelles directives données par M. Guisse. Comme nous avons terminé AES en C, nous devons maintenant optimisé notre code. Pour ce faire nous allons avoir recours à l'outil Valgrind. En début de matinée nous avons essayé de comprendre pourquoi notre code était lent comparé aux benchmarks que nous avons pu trouver sur internet. Nous avons eu plusieurs idées :

- L'utilisation de matrice temporaire.
- Les boucles / affectations inutiles.
- ..

En début d'apres midi nous avons eu une explication sur l'outil Valgrind, nous avons pu déterminer quelles étaient les fonctions en tord.



Nous pouvons voir que `key_expansion` représente une partie importante de l'utilisation du processeur alors que les clés ne changent pas au cours du temps.

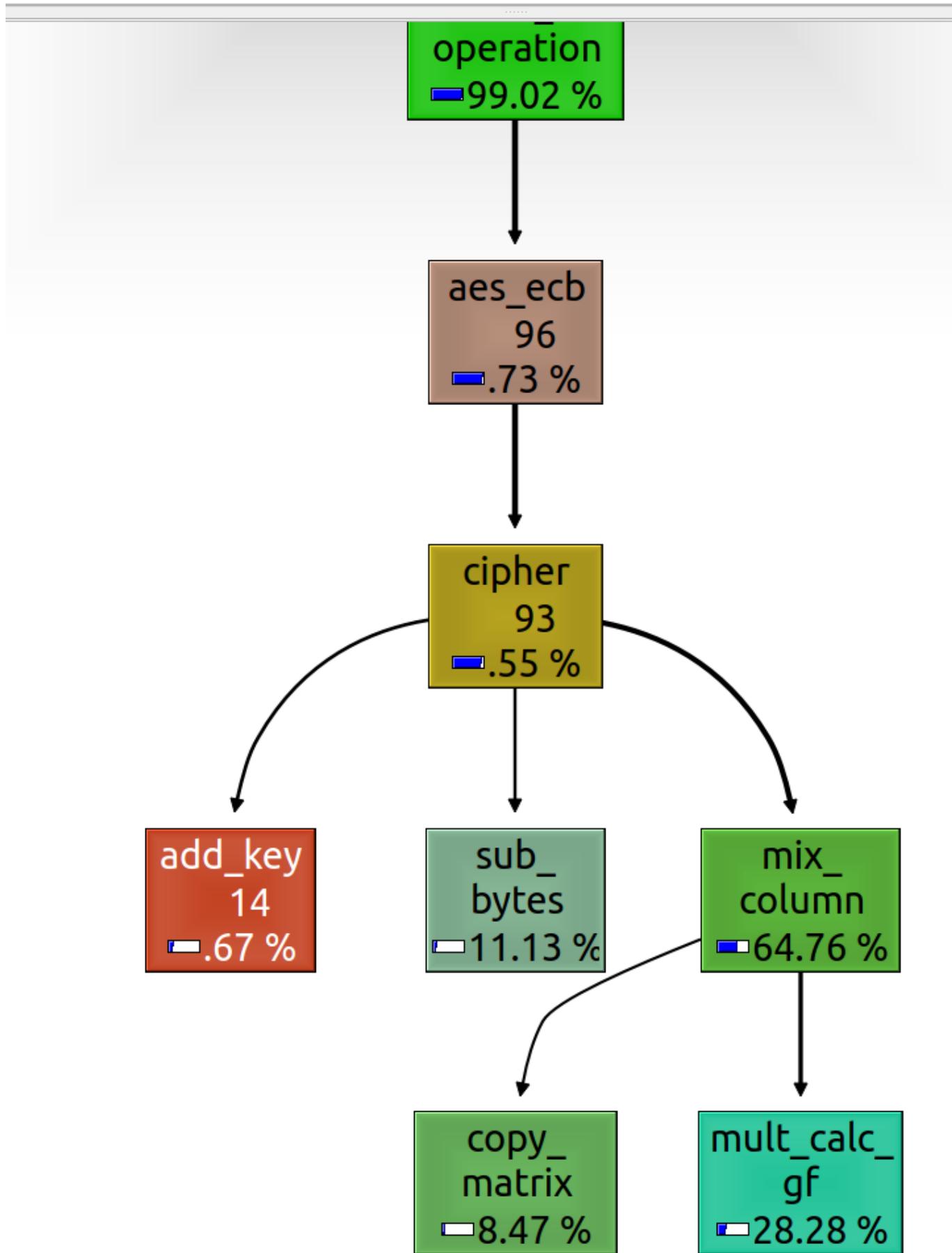
Ceci s'explique par le fait que les 10 clés (pour 128 bits) sont calculés **à chaque** appel pour chiffrer un bloc de 16 octets. Ainsi, si nous avons un fichier de 1073741824 octets, comme précédamment, nous allons calculer les clés 67 108 864 fois. Ce calcul de clé sera identique à chaque appel. En modifiant notre code permettant l'usage d'une variable stockant les 10 clés, le calcul n'est réalisé plus qu'une seule fois.

La clé est donc maintenant calculé avant l'appel d'AES et remplit le tableau de clés permettant de chiffrer ou déchiffrer nos données.

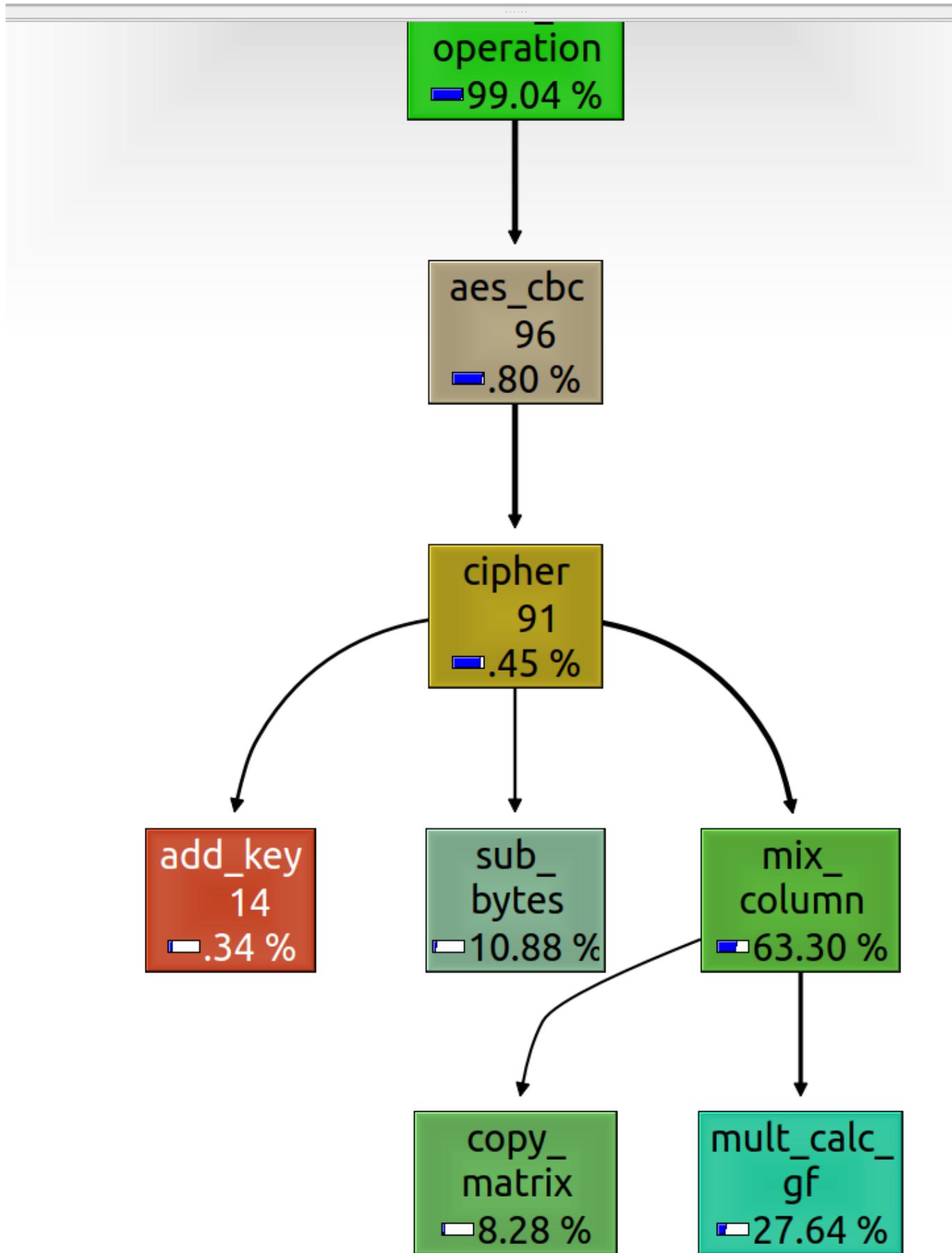
On obtient que notre fichier de 1 Go précédemment utilisé est chiffré par la méthode CBC en 8 minutes. Ceci représente une vitesse de plus de 2,2 Mo/s.

Nous avons donc réussi à optimiser, à ce jour, notre protocole de chiffrage/decryptage en passant de 778 Ko/s à 2,2 Mo/s. Ceci représente une augmentation de 187 %, soit un fois 2,87.

Sur cette image on peut voir la répartition en pourcentage avec ECB après l'optimisation.



Sur cette image il s'agit cette fois-ci de CBC.



Nous voyons ici la performance de notre programme en nombre d'opérations. Nous pouvons aussi voir l'aspect de mémoire du programme. Pour des fonctions utilisant des boucles, nous utilisons des compteurs par entier. `for(int i = 0; i < 4; i++)`. Pour réduire ces opérations et le temps, nous pouvons utiliser des variables statiques dans les fonctions ne s'appelant pas récursivement. Nous utilisons *static* pour les variables temporaires, et les indices de boucles. Ainsi, l'attribution de mémoire sera faite en début de programme seulement et ne sera pas réalisé après. Ceci ne réduit pas la quantité d'utilisation mémoire mais le nombre d'appels d'allocation et de libération des espaces mémoires.

Une telle modification de notre programme nous permet de chiffrer par la méthode CBC notre fichier de 1073741824 octets en 3 minutes et 51 secondes. Ceci nous permet d'atteindre une vitesse de plus de 4,6 Mo/s.

Notons que toutes les tests ont tous été réalisés sur la même machine, dans les mêmes conditions.

Mardi 6 juin 2023

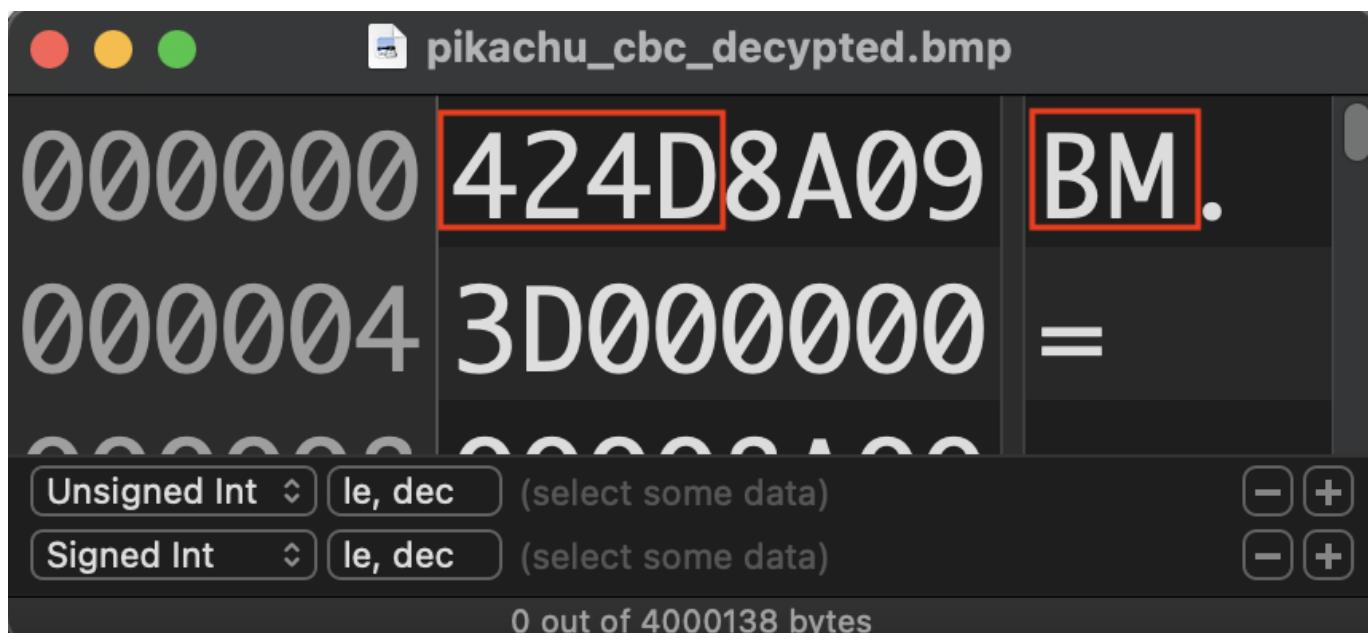
Nous avons travaillé lors de cette séance en commun grâce à l'outil *Live Share* de VSCode.

Dans un premier temps, nous avons continué de travailler sur l'optimisation de notre code. En passant par des variables temporaires plutôt que de copier/coller des matrices, on arrive à une vitesse d'environ 7 Mo/s.

Lecture des fichiers BMP

Par la suite, nous voulions mettre un visuel sur la différence entre ECB et CBC. Pour cela, nous utilisons le travail réalisé durant le TP 2 de CS211. Ce TP concernait la manipulation de fichiers *bitmap*, avec la gestion d'entête et de données.

Les fichiers *bitmap* possèdent une signature. Cette signature est constituée par les deux premiers octets du fichier. On peut voir ci-dessous que la signature est **BM** ou *0x424D* en hexadécimal.



Nous pouvons donc facilement savoir si le fichier d'entrée est un fichier au format bitmap ou non. On peut le faire de cette manière :

```

int is_bmp(char* input_name) {
    FILE* input = fopen(input_name, "r");

    if(input == NULL) return 0;

    unsigned short signature = 0;

    fread(&signature, sizeof(unsigned short), 1, input);

    fclose(input);
    return signature == 0x4D42;
}

```

Notre fonction `bitmap_process` nous permet ainsi d'écrire les métadonnées (entête fichier, entête image et palette couleur) dans le fichier de sortie avant de chiffrer les données de l'image.

Notre protocole de prise d'information et du processus de chiffrage ou déchiffrage par ECB ou CBC se traduit donc de la manière suivante :

```

int decrypt = argv[1][0] == 'd';
int bmp = is_bmp(argv[i]);
int ecb = bmp?argv[2][1] == 'e':argv[2][0] == 'e';
char* output = combine_strings(argv[i], decrypt?"_decrypted":"_crypté");

double time = (double) clock() / CLOCKS_PER_SEC;

key_expansion(key); // permet de calculer les clés du roundkey

if (bmp) // on s'occupe d'une image bmp
    if (decrypt) // décrypte
        succès = bitmap_process (argv[i], output, ecb?(&aes_ecb_inverse):
(&aes_cbc_inverse));
    else // chiffre
        succès = bitmap_process (argv[i], output, ecb?(&aes_ecb):
(&aes_cbc));
else
    if (decrypt) // décrypte
        succès = file_operation (argv[i], output, ecb?(&aes_ecb_inverse):
(&aes_cbc_inverse));
    else // chiffre
        succès = file_operation (argv[i], output, ecb?(&aes_ecb):
(&aes_cbc));

time = (((double) clock()) / (double) CLOCKS_PER_SEC) - time;

if (succès)
    printf("Fichier %s %s (%s) dans %s avec succès %.5fs.\n", argv[i],
decrypt?"déchiffré":"chiffré", (argv[2][0] == 'e')?"ECB":"CBC", output,
time);
else

```

```
printf("Erreur durant le %s de %s\n",
decrypt?"déchiffrage":"chiffrage", argv[i]);
```

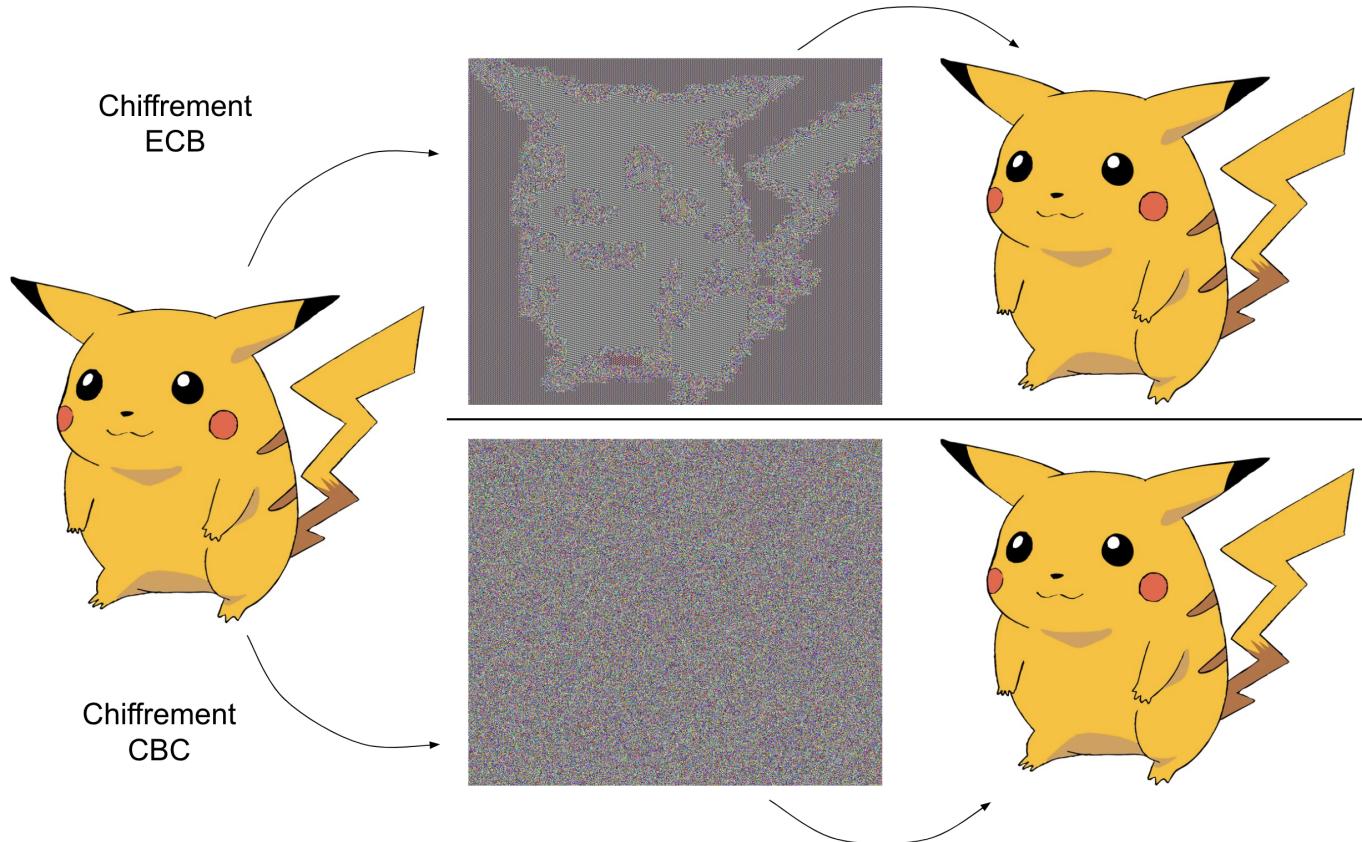
Ainsi, si notre fichier est un fichier bitmap, le processus sera différent et nous permettra de voir l'image cryptée. Nous pouvons donc maintenant mettre en lumière la différence entre ECB et CBC.

Exemple :

```
Début du chiffrement de aes_tests/pika.bmp (ECB).
Fichier aes_tests/pika.bmp chiffré (ECB) dans
aes_tests/pika_ecb_crypted.bmp avec succès (0.37928s).
./aes d e key aes_tests/pika_ecb_crypted.bmp
Début du déchiffrement de aes_tests/pika_ecb_crypted.bmp (ECB).
Fichier aes_tests/pika_ecb_crypted.bmp déchiffré (ECB) dans
aes_tests/pika_ecb_crypted_decrypted.bmp avec succès (0.35048s).
```

On réalise la même chose avec le protocole CBC.

On obtient les images suivantes :



Comme nous avions vu précédemment, nous pouvons voir que le chiffrement par ECB représente une faille dans les données, en reproduisant à l'identique deux blocs de données. Le chiffrement par CBC ne représente pas ce problème.

Mesure de l'entropie

L'entropie est le concepte correspondant au degré de chaos d'un système.

Dans notre cas, l'entropie d'un fichier correspond aux aléas du fichier. Plus l'entropie est importante, plus le fichier est varié d'informations (d'octets). L'entropie se mesure, pour nous, en **bits par octets** et prend des valeurs de 0 à 8 bits par octet, 0 b/o étant un fichier dont tout les octets sont identiques et 8 b/o un fichier le plus aléatoire possible.

Pour mesurer l'entropie de nos différents fichiers, on crée un executable *entropie*, prenant en entrée les fichiers dont l'entropie doit être calculée : Usage : `./entropie <fichiers>`.

L'entropie se calcule de la façon suivante, où P représente la répartition de l'octet i , n représente le nombre d'octet i dans le fichier et N le nombre d'octets total du fichier :
$$\text{Entropie} = -\sum_{i=0}^{255} P_i \log_2(P_i)$$
 avec $P_i = \frac{n_i}{N}$

Afin de calculer simplement la répartition des octets dans le fichier, nous pouvons parcourir notre fichier et incrémenter de 1 chaque octet rencontré dans un tableau, de la façon suivante :

```
void calculate_occurrence(FILE* file, long occurrences[max]) {
    int c;
    fseek(file, 0, SEEK_SET);
    while((c = fgetc(file)) != EOF) {
        occurrences[c] += 1;
    }
}
```

Nous pouvons facilement déterminer la taille en octets du fichier de la manière suivante :

```
fseek(file, 0, SEEK_END);
long size = ftell(file);
```

Ce qui nous permet de déterminer l'entropie par le calcul de la somme :

```
for (int o = 0; o < 256; o++) {
    if (occurrences[o] != 0) { // Pour ne pas calculer log2(0)...
        double P = occurrences[o] / ((double) size);
        entrop -= P * log2(P);
    }
}
```

On trouve ainsi :

```
./entropie pika.bmp pika_ecb_crypted.bmp pika_cbc_crypted.bmp
Entropie du fichier aes_tests/pika.bmp : 2.933884 bits par octet.
Entropie du fichier aes_tests/pika_ecb_crypted.bmp : 6.894644 bits par
octet.
Entropie du fichier aes_tests/pika_cbc_crypted.bmp : 7.999853 bits par
octet.
```

	Pikachu de base	Pikachu chiffré ECB	Pikachu chiffré CBC
Silhouette	X	Reconnaissable	Brouillée
Entropie	2,93 bits/octet	6,89 bits/octet	7,9998 bits/octet

On peut donc voir que l'entropie du pikachu chiffré par CBC à une entropie quasiment égale à 8 bits/octet, ce qui signifie que la suite obtenue est quasiment aléatoire et donc qu'il n'y a que très peu voir aucune redondance entre les mots. La méthode ECB permet également une valeur haute d'environ 6,9 bits/octet. Cette valeur reste inférieure de 1,1 bits/octet, soit plus de 13,8 % de cette valeur.

Essayons maintenant d'analyser l'entropie d'un fichier crypté par ECB ou CBC comportant 256 et 2048 octets identiques :

	Bloc de 256 octets identiques	Bloc chiffré ECB	Bloc chiffré CBC
Entropie	0 bits/octet	4 bits/octet	7,17 bits/octet
	Bloc de 2048 octets identiques	Bloc chiffré ECB	Bloc chiffré CBC
Entropie	0 bits/octet	4 bits/octet	7,92 bits/octet

Notons que dans tous les cas, la clé de chiffrement est identique pour ne pas créer de facteur externe.

Nous pouvons voir que l'entropie d'un fichier comportant des octets identiques est bien de 0. Quant au chiffrement ECB, nous pouvons voir que l'entropie est toujours de 4 bits/octets. En effet, nous avons une répétition d'un bloc de 16 octets. Ainsi, la manière dont le chiffrement est fait n'étant pas variable, l'entropie est constante (dépend de la clé et des données mais pas du nombre de répétitions du bloc). Le bloc est : 06 66 89 52 DB 51 ED F1 C6 16 80 A5 6C 7E 7F 4A. On voit bien que chaque octet étant répété une seule fois, on a : $H = 16 * (-\frac{1}{16} * \log_2(\frac{1}{16})) = \log_2(16) = 4$ bits/octet

Pour CBC, on voit que plus il y a de données, plus l'entropie augmente. Ceci est cohérent avec le principe de CBC.

Pour conclure sur l'entropie des fichiers issus par les méthodes ECB ou CBC, nous pouvons dire que nous avons vu plusieurs manières d'affirmer que AES CBC est une méthode de chiffrement plus sécurisée que la méthode de chiffrement AES ECB.

Durant la prochaine séance, nous nous attarderons sur les comportements de notre programme et des fuites de mémoires potentielles.

Mercredi 7 juin 2023

Blabla mémoire...

Nous pouvons également voir en utilisant l'outil valgrind que la multiplication dans *mix_column* nous prend un temps important. Étant donné que nous n'avons pas à nous soucier des failles de sécurité de notre executable, nous allons implémenter les tables de multiplications en clair pour les multiplications par : 2, 3, 9, 0xB, 0xD et 0xE.

Pour facilement clarifier ces tables, on réalise la fonction suivante, qui nous permet de directement afficher sous la forme d'un tableau la table :

```
void get_table (octet a) {
    printf("octet[256] mult_%X =\t{", a);
    for (int i = 0; i < 256; i++) {
        printf("0x%X", mult_calc_gf(a, i));
        if (i != 255)
            printf(",");
        if (i%16 == 0 && i != 0)
            printf("\n\t\t\t");
    }
    printf("};\n");
}
```

Puis, on récupère facilement les tables :

```
get_table(2);
get_table(3);
get_table(9);
get_table(0xb);
get_table(0xd);
get_table(0xe);
```

Ceci nous permet de modifier notre *mix_column* en ne faisant plus d'appels à des fonctions mais seulement en manipulant les tableaux.

Nous devons modifier notre code pour nous permettre de facilement faire les multiplications. Étant donné que nous n'arrivons pas à manipuler les tableaux de tableaux des tables de multiplications, on réalise un code nous permettant d'adapter ce que nous avions au préalable avec l'utilisation de tableaux au lieu de fonctions.

On obtient le code grâce à la fonction suivante :

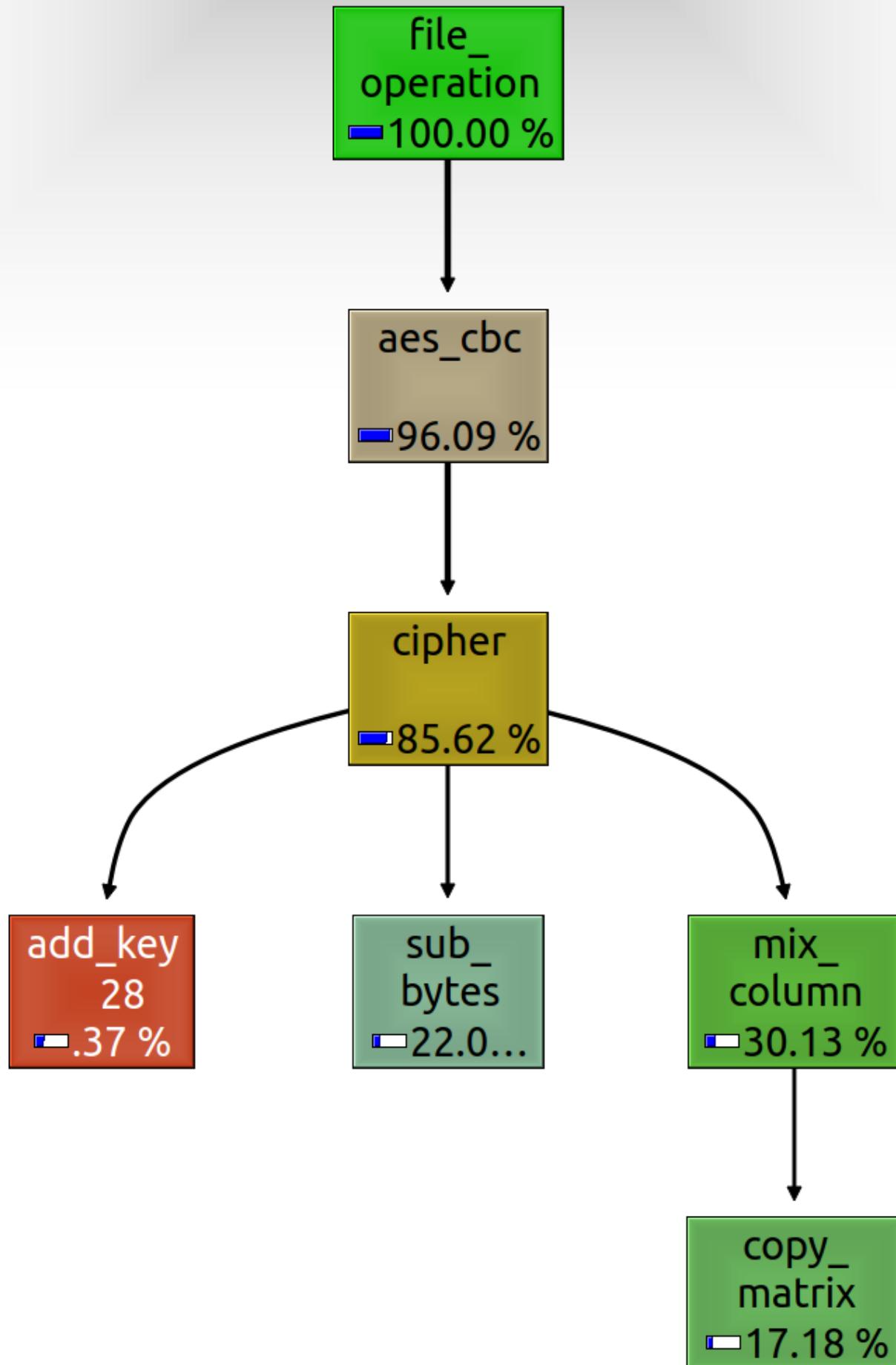
```
void print_mix_col() {
    const matrix M = {{0x0e, 0x0b, 0x0d, 0x09},
                      {0x09, 0x0e, 0x0b, 0x0d},
                      {0x0d, 0x09, 0x0e, 0x0b},
                      {0x0b, 0x0d, 0x09, 0x0e}};

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            printf("state[%d][%d] = mult_%X[tmp[0][%d]] ^ mult_%X[tmp[1]
[%d]] ^ mult_%X[tmp[2][%d]] ^ mult_%X[tmp[3][%d]];\n", i, j, M[i][0], j,
M[i][1], j, M[i][2], j, M[i][3], j);
}
```

Qui nous permet d'obtenir :

```
state[0][0] = mult_E[tmp[0][0]] ^ mult_B[tmp[1][0]] ^ mult_D[tmp[2][0]] ^  
mult_9[tmp[3][0]];  
state[0][1] = mult_E[tmp[0][1]] ^ mult_B[tmp[1][1]] ^ mult_D[tmp[2][1]] ^  
mult_9[tmp[3][1]];  
state[0][2] = mult_E[tmp[0][2]] ^ mult_B[tmp[1][2]] ^ mult_D[tmp[2][2]] ^  
mult_9[tmp[3][2]];  
state[0][3] = mult_E[tmp[0][3]] ^ mult_B[tmp[1][3]] ^ mult_D[tmp[2][3]] ^  
mult_9[tmp[3][3]];  
state[1][0] = mult_9[tmp[0][0]] ^ mult_E[tmp[1][0]] ^ mult_B[tmp[2][0]] ^  
mult_D[tmp[3][0]];  
state[1][1] = mult_9[tmp[0][1]] ^ mult_E[tmp[1][1]] ^ mult_B[tmp[2][1]] ^  
mult_D[tmp[3][1]];  
state[1][2] = mult_9[tmp[0][2]] ^ mult_E[tmp[1][2]] ^ mult_B[tmp[2][2]] ^  
mult_D[tmp[3][2]];  
state[1][3] = mult_9[tmp[0][3]] ^ mult_E[tmp[1][3]] ^ mult_B[tmp[2][3]] ^  
mult_D[tmp[3][3]];  
state[2][0] = mult_D[tmp[0][0]] ^ mult_9[tmp[1][0]] ^ mult_E[tmp[2][0]] ^  
mult_B[tmp[3][0]];  
state[2][1] = mult_D[tmp[0][1]] ^ mult_9[tmp[1][1]] ^ mult_E[tmp[2][1]] ^  
mult_B[tmp[3][1]];  
state[2][2] = mult_D[tmp[0][2]] ^ mult_9[tmp[1][2]] ^ mult_E[tmp[2][2]] ^  
mult_B[tmp[3][2]];  
state[2][3] = mult_D[tmp[0][3]] ^ mult_9[tmp[1][3]] ^ mult_E[tmp[2][3]] ^  
mult_B[tmp[3][3]];  
state[3][0] = mult_B[tmp[0][0]] ^ mult_D[tmp[1][0]] ^ mult_9[tmp[2][0]] ^  
mult_E[tmp[3][0]];  
state[3][1] = mult_B[tmp[0][1]] ^ mult_D[tmp[1][1]] ^ mult_9[tmp[2][1]] ^  
mult_E[tmp[3][1]];  
state[3][2] = mult_B[tmp[0][2]] ^ mult_D[tmp[1][2]] ^ mult_9[tmp[2][2]] ^  
mult_E[tmp[3][2]];  
state[3][3] = mult_B[tmp[0][3]] ^ mult_D[tmp[1][3]] ^ mult_9[tmp[2][3]] ^  
mult_E[tmp[3][3]];
```

Grâce à l'outil valgrind, on trouve :



Ainsi, notre temps utilisé pour *mix_column* est bien réduit. On trouve tout de même que le temps de copie des matrices est important. Nous allons essayer de ne pas avoir à copier des matrices pour nous permettre de gagner du temps. Nous pouvons, au lieu de travailler sur les matrices appelées par la fonction directement travailler par un retour de matrice.

Pour l'après midi nous avons décidé de nous diviser les tâches. Lucas s'occupera de réaliser la gestion des fichiers et nolan s'occupera d'AES 192 et 256.

Nous avons rencontré quelques difficultés concernant AES 192 et 256. En effet le programme que nous avions réalisé était spécifique pour l'utilisation d'une clé de 128 bits.

```
void key_expansion_round_n(matrix key, int n) {
    transpose_matrix(key);

    static matrix tmp; copy_matrix(key, tmp);
    g_function(tmp[3]);

    xor_word(key[0], key[0], tmp[3]);
    xor_word(key[1], key[0], tmp[1]);
    xor_word(key[2], key[1], tmp[2]);
    xor_word(key[3], key[2], key[3]);

    transpose_matrix(key);

    static int i;
    for(i = 0; i < 4; i++)
        key[0][i] = key[0][i] ^ mod_irr_gf(1 << n);
}
```

Pour corriger ce problème nous avons du réfléchir à l'utilisation d'une algorithme pouvant réaliser AES128,192 et 256. Le choix se faisant en fonction d'un paramètre ou d'une variable globale. Nous avons aussi essayé de ne pas utiliser la transposition de matrice pour gagner du temps. Après avoir étudier la documentation nous avons écrit l'algorithme suivant :

```
void all_key_expansion(octet key[4*Nk], octet w[Nb*(Nr+1)][4]){
    static int i = 0;
    word temp;
    while (i < Nk){
        for(int j = 0; j<4; j++){
            w[i][j] = key[4*i+j];
        }
        i += 1;
    }
    word r_con = {0x0,0x0,0x0,0x0};
    i = Nk;

    while (i < Nb * (Nr+1)){
        for(int j = 0; j<4; j++){
            temp[j] = w[i-1][j];
        }
    }
}
```

```

    }
    if ((i % Nk) == 0){
        r_con[0] = mod_irr_gf((0x01)<<((i/Nk)-1));
        g_function(temp);
        xor_word(temp,temp, r_con);
    }
    else if ((Nk > 6) && (i % Nk == 4)){
        SubWord(temp);
    }
    xor_word(w[i],w[i-Nk], temp);
    i += 1;
}
}

```

Nk = Nombre de words dans la clé, Nr = Nombre de rounds, Nb = Taille des bloc en words.

Avec ces trois variables que nous initialisons au début du programme, nous pouvons réaliser l'extension de la clé, quelle soit de 128, 192 et 256 octets.

Gestion de la taille des fichiers

Un problème peut être rencontré sur les tailles de fichiers, c'est lorsque un fichier ne contient pas un nombre d'octet multiple de 16. Avant cette séance, Mr. Guisse nous avait dit de combler avec des 0x00 les fichiers et de ne pas se soucier du fait qu'il y aura des octets supplémentaires dans le fichier déchiffré.

Cependant, nous devons maintenant prendre en compte ce problème et donc implémenter un outil permettant de savoir le nombre exacts d'octets dans le fichier originel. La fonction *f tell* permet d'obtenir, en la combinant avec *fseek*, la taille du fichier. Nous pouvons faire cela et l'enregistrer dans un *long*, qui est son type de retour. Travaillant sur des machines 64 bits, le type *long* est codé dans 8 octets. Ceci nous permet de travailler jusqu'à un fichier de plus de 9,2 exaoctets, soit 9,2 giga de gigaoctets. Ceci est suffisant.

Les 8 premiers octets du fichier chiffré correspondent donc à la taille du fichier, et seront stockés de la manière suivante : 0xABCD -> 0xAB | 0xCD.

Notre astuce consiste à forcer l'encodage de notre taille (*long*) en un octet tout en réalisant un décallage de multiple de 8 octets. On le réalise de la manière suivante :

```

size = file_size(input);

data_matrix[0][0] = size >> (8*7);
data_matrix[1][0] = size >> (8*6);
data_matrix[2][0] = size >> (8*5);
data_matrix[3][0] = size >> (8*4);
data_matrix[0][1] = size >> (8*3);
data_matrix[1][1] = size >> (8*2);
data_matrix[2][1] = size >> (8*1);
data_matrix[3][1] = size;

```

On peut ainsi facilement, selon les cas de chiffrement ou déchiffrement, procéder à l'écriture et la lecture des données ainsi que définir ou déterminer la taille du fichier originel afin de ne pas écrire les derniers octets ajoutés lors du complément à 0x00 pour le chiffrement des données.

On réalise un script *BASH* permettant de vérifier que les fichiers d'entrées et de sorties sont identiques. On réalise pour cela 17 fichiers d'une longueur de 60 octets à 67 octets, permettant ainsi de traiter tout les cas de reste d'octets par blocs de 16.

En premier lieu, ce script chiffre puis déchiffre les 17 fichiers par la méthode ECB et CBC, en supprimant les fichiers chiffrés :

```
#!/bin/bash
rm test/*_*
for t in ecb cbc do
  ./aes c "$t" fips_key test/*x
  ./aes d "$t" fips_key test/*"$t"*
done
rm test/*_cryptd
```

Puis, ce script vérifie que chaque fichier sont bien identiques. Un message apparaît en cas de différences.

```
#!/bin/bash
for t in ecb cbc do
  for i in {1..16} do
    diff test/"$i"x test/"$i"x_"$t"_cryptd_"$t"_decrypted
  done
done
```

On obtient ainsi que dans les deux cas, notre fichier est identique à 100% à l'originel.

On trouve en effet bien que le fichier source et déchiffré est identique, dans le cas du chiffrement du bloc d'exemple d'AES de la doc. FIPS :

fips_test		fips_test_cbc_crypted		fips_test_cbc_crypted_cbc_decrypted	
0	3243F6A8	0	8DB6C6D8	0	3243F6A8 2C..
4	885A308D	4	BA0A7C4E	4	885A308D .Z0.
8	313198A2	8	F18CA913	8	313198A2 11..
12	E0370734	12	EACD3117	12	E0370734 .7 4
16	01020304	16	9898C7A0	16	01020304
20	05	20	57108E53	20	05
		24	24052BBD		
		28	DCF0AC18		

Afin de chiffrer correctement AES 128, 196 et 256 et sans modifier radicalement nos fonctions, on modifie notre code pour ne plus travailler sur des matrices mais sur des tableaux. Ainsi, un tableau *matrix[4][4]* devient *tab[16]*.

Nous devons travailler, dans notre ultime séance, sur la création des fichiers AES en version 128, 192 et 256 bits ainsi que le code permettant l'utilisation correcte d'AES.

Mercredi 7 juin 2023

Nous avons distribué les tâches simplement, après avoir vérifié simplement que notre code d'AES 192 et 256 est fonctionnel :

Nolan	Lucas
Vérification des blocs avec la doc	Modifier la fonction d'extension de clé
Création de tests pour aes 192 et 256	Modification du Makefile permettant de compiler les 3 versions d'AES

L'extension de clé est maintenant réalisée :

```
octet w[Nb*(Nr+1)][4] = {{}};  
  
all_key_expansion(key, w);  
  
int select = 0, counter = 0;  
  
for(int i = 0; i < (Nb*(Nr+1)); i++){  
    for(int j = 0; j<4; j++){  
        round_keys[select][counter] = w[i][j];  
        counter++;  
        if (counter == Matrix_size) {
```

```
        counter = 0;  
        select++;  
    }  
}
```

Nous gardons le même système d'extendre les clés au début et de les stocker dans un tableau de clés. La taille de ce tableau dépend du type d'AES utilisé géré automatiquement par des appels avec un Flag dans le *all.h* :

```
#ifdef AES192
    #define Nk 6
    #define Nr 12
#elif AES256
    #define Nk 8
    #define Nr 14
#else
    #define Nk 4
    #define Nr 10
#endif
```

Ces appels seront fait dans la compilation en objet .o de la manière suivante :

```
gcc -Wall -g -c $< -DAES128 -o $@
```

Ainsi, nous pouvons facilement créer notre Makefile générique de la façon suivante, pour gérer automatiquement les versions 128, 192 et 256 bits sans intervention :

```
aes:
    make aes128
    make aes192
    make aes256
    rm -rf *.o

aes128: main128.o utile128.o functions128.o tests128.o files128.o
p bmp128.o entropie128.o
    gcc -Wall -g -o $@ $^ -O3 -lm

aes192: main192.o utile192.o functions192.o tests192.o files192.o
p bmp192.o entropie192.o
    gcc -Wall -g -o $@ $^ -O3 -lm

aes256: main256.o utile256.o functions256.o tests256.o files256.o
p bmp256.o entropie256.o
    gcc -Wall -g -o $@ $^ -O3 -lm

%128.o: %.c
```

```
gcc -Wall -g -c $< -DAES128 -o $@

%192.o: %.c
gcc -Wall -g -c $< -DAES192 -o $@

%256.o: %.c
gcc -Wall -g -c $< -DAES256 -o $@
```

Nous pouvons donc soit compiler nos versions 128, 192 et 256 ensemble en utilisant `make` ou `make all` et les versions indépendamment en utilisant `make aes128`, `make aes192` ou `make aes256`. Lorsque l'on lance le `make`, on a :

```
luka.mig@MBP-de-Lucas AES C % make
make aes128
gcc -Wall -g -c main.c -DAES128 -o main128.o
gcc -Wall -g -c utile.c -DAES128 -o utile128.o
gcc -Wall -g -c functions.c -DAES128 -o functions128.o
gcc -Wall -g -c tests.c -DAES128 -o tests128.o
gcc -Wall -g -c files.c -DAES128 -o files128.o
gcc -Wall -g -c p bmp.c -DAES128 -o p bmp128.o
gcc -Wall -g -c entropie.c -DAES128 -o entropie128.o
gcc -Wall -g -o aes128 main128.o utile128.o functions128.o tests128.o p bmp128.o entropie128.o -O3 -lm
make aes192
gcc -Wall -g -c main.c -DAES192 -o main192.o
gcc -Wall -g -c utile.c -DAES192 -o utile192.o
gcc -Wall -g -c functions.c -DAES192 -o functions192.o
gcc -Wall -g -c tests.c -DAES192 -o tests192.o
gcc -Wall -g -c files.c -DAES192 -o files192.o
gcc -Wall -g -c p bmp.c -DAES192 -o p bmp192.o
gcc -Wall -g -c entropie.c -DAES192 -o entropie192.o
gcc -Wall -g -o aes192 main192.o utile192.o functions192.o tests192.o files192.o p bmp192.o entropie192.o -O3 -lm
make aes256
gcc -Wall -g -c main.c -DAES256 -o main256.o
gcc -Wall -g -c utile.c -DAES256 -o utile256.o
gcc -Wall -g -c functions.c -DAES256 -o functions256.o
gcc -Wall -g -c tests.c -DAES256 -o tests256.o
gcc -Wall -g -c files.c -DAES256 -o files256.o
gcc -Wall -g -c p bmp.c -DAES256 -o p bmp256.o
gcc -Wall -g -c entropie.c -DAES256 -o entropie256.o
gcc -Wall -g -o aes256 main256.o utile256.o functions256.o tests256.o p bmp256.o entropie256.o -O3 -lm
rm -rf *.o
luka.mig@MBP-de-Lucas AES C % ls aes*
aes128  aes192  aes256
```

Ceci nous génère bien nos 3 executables d'AES, pour les versions 128, 192 et 256 bits.

Pour vérifier que nos algorithmes de chiffrement fonctionnent nous avons utilisé la documentation. Nous avons réalisé les tests de chiffrement et déchiffrement pour les versions 128, 192 et 256 bits. Pour pouvoir utiliser chiffrer et déchiffrer simplement un bloc d'octet sans avoir à se soucier de la taille des fichiers inscrite au début de chaque chiffrement, on désactive cette option par l'utilisation simple d'un `define`.

Nous avions déjà les tests pour la version 128 bits, nous avons juste implémenté les tests pour la version 192 et 256 bits. Nous obtenons en exécutant les différents exécutables :

```
nolancarouge@nolancarouge:~/Desktop/PX222 IRC/binome19/AES C$ ./aes128
Test sub_bytes OK
Test inv_sub_bytes OK
Test shift_rows OK
Test inv_shift_rows OK
Test add_key OK
Test key_expansion_128 OK
Test mix_columns OK
Test inverse_mix_columns OK
Test cipher OK
Test cipher_inverse OK
nolancarouge@nolancarouge:~/Desktop/PX222 IRC/binome19/AES C$ ./aes192
Test sub_bytes OK
Test inv_sub_bytes OK
Test shift_rows OK
Test inv_shift_rows OK
Test add_key OK
Test key_expansion_192 OK
Test mix_columns OK
Test inverse_mix_columns OK
Test cipher OK
Test cipher_inverse OK
nolancarouge@nolancarouge:~/Desktop/PX222 IRC/binome19/AES C$ ./aes256
Test sub_bytes OK
Test inv_sub_bytes OK
Test shift_rows OK
Test inv_shift_rows OK
Test add_key OK
Test key_expansion_256 OK
Test mix_columns OK
Test inverse_mix_columns OK
Test cipher OK
Test cipher_inverse OK
```

Nous pouvons conclure que nos différentes versions d'AES sont fonctionnelles.

En réintégrant la fonctionnalité du chiffrement de la taille du fichier en octet, nous pouvons adapter le test automatique des 17 fichiers que nous avions réalisé au préalable pour chacune des versions d'AES. On a pour la création des fichiers :

```
#!/bin/bash
rm test/*_*
for a in 128 192 256 do
for t in ecb cbc do
./aes"$a" c "$t" key_"$a" test/*
./aes"$a" d "$t" key_"$a" test/*"$t"*$a"*
done
done
rm test/*_cryptd_128
rm test/*_cryptd_192
rm test/*_cryptd_256
```

Pour la vérification :

```
#!/bin/bash
for a in 128 192 256 do
for t in ecb cbc do
for i in {1..16} do
diff test/"$i"x test/"$i"x_"$t"_crypted_"$a"_"$t"_decrypted
done
done
done
```

On modifie également notre accès aux tests de la manière suivante pour facilement montrer leurs utilisations lors du passage à l'oral :

- `./aes128 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]`
- `./aes192 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]`
- `./aes256 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]`

Tests finaux : AES128 - AES192 - AES 256 : ECB et CBC

Nous allons maintenant procéder aux derniers tests de nos différentes méthodes de chiffrement et dechiffrement d'AES. Pour cela, nous allons chiffrer et dechiffrer un fichier texte de 69 577 octets, qui n'est pas un multiple de 16. Nous pourrons donc bien essayer de façon efficace notre option d'encodage de taille du fichier.

AES 128

```
luka.mig@MBP-de-Lucas AES C % ./aes128
Usage : ./aes128 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]
Raccourcis :
    - c = crypt
    - d = decrypt
    - t = test
    - s = show
    - h = hide
    - iecb et icbc permettent de chiffrer des fichiers bmp avec l'entête en clair.
luka.mig@MBP-de-Lucas AES C % ./aes128 crypt ecb show aes_tests/interstellar.txt
Entrez la clé de chiffrement (sans accents) : Ceci est une Clé
Début du chiffrement de aes_tests/interstellar.txt (ECB).
Fichier aes_tests/interstellar.txt chiffré (ECB - 128 bits) dans aes_tests/interstellar.txt_ecb_crypted_128 avec succès (0.01399s).
Entropie : 7.997269 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes128 decrypt ecb show aes_tests/interstellar.txt_ecb_crypted_128
Entrez la clé de chiffrement (sans accents) : Ceci est une Clé
Début du déchiffrement de aes_tests/interstellar.txt_ecb_crypted_128 (ECB).
Fichier aes_tests/interstellar.txt_ecb_crypted_128 déchiffré (ECB - 128 bits) dans aes_tests/interstellar.txt_ecb_decrypted avec succès (0.01464s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % diff aes_tests/interstellar.txt aes_tests/interstellar.txt_ecb_crypted_128_ecb_decrypted
luka.mig@MBP-de-Lucas AES C % ./aes128 crypt cbc key aes_tests/interstellar.txt
Début du chiffrement de aes_tests/interstellar.txt (CBC).

Fichier aes_tests/interstellar.txt_cbc_crypted_128 chiffré (CBC - 128 bits) dans aes_tests/interstellar.txt_cbc_crypted_128 avec succès (0.01496s).
Entropie : 7.997148 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes128 decrypt cbc key aes_tests/interstellar.txt_cbc_crypted_128
Début du déchiffrement de aes_tests/interstellar.txt_cbc_crypted_128 (CBC).

Fichier aes_tests/interstellar.txt_cbc_crypted_128 déchiffré (CBC - 128 bits) dans aes_tests/interstellar.txt_cbc_decrypted avec succès (0.01427s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % diff aes_tests/interstellar.txt aes_tests/interstellar.txt_cbc_crypted_128_cbc_decrypted
luka.mig@MBP-de-Lucas AES C |
```

AES 192

```

luka.mig@MBP-de-Lucas AES C % ./aes192
Usage : ./aes192 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]
Raccourcis :
- c = crypt
- d = decrypt
- t = test
- s = show
- h = hide
- iecb et icbc permettent de chiffrer des fichiers bmp avec l'entête en clair.
luka.mig@MBP-de-Lucas AES C % ./aes192 crypt ecb show aes_tests/interstellar.txt
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle longuee
Début du chiffrement de aes_tests/interstellar.txt (ECB).

Fichier aes_tests/interstellar.txt chiffré (ECB - 192 bits) dans aes_tests/interstellar.txt_ecb_crypted_192 avec succès (0.01584s).
Entropie : 7.997228 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes192 decrypt ecb show aes_tests/interstellar.txt_ecb_crypted_192
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle longuee
Début du déchiffrement de aes_tests/interstellar.txt_ecb_crypted_192 (ECB).

Fichier aes_tests/interstellar.txt_ecb_crypted_192 déchiffré (ECB - 192 bits) dans aes_tests/interstellar.txt_ecb_crypted_192_ecb_decrypted avec succès (0.01516s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes192 crypt cbc show aes_tests/interstellar.txt_ecb_crypted_192_ecb_decrypted
luka.mig@MBP-de-Lucas AES C % ./aes192 crypt cbc show aes_tests/interstellar.txt
Entrez la clé de chiffrement (sans accents) : Cecie
Entrez la clé de chiffrement (sans accents) : Cecies
Entrez la clé de chiffrement (sans accents) : Cecie
Entrez la clé de chiffrement (sans accents) : Ceci est une cle eeeeeee
Début du chiffrement de aes_tests/interstellar.txt (CBC).

Fichier aes_tests/interstellar.txt chiffré (CBC - 192 bits) dans aes_tests/interstellar.txt_cbc_crypted_192 avec succès (0.01538s).
Entropie : 7.997079 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes192 decrypt cbc show aes_tests/interstellar.txt_cbc_crypted_192
Entrez la clé de chiffrement (sans accents) : Ceci est une cle c
Entrez la clé de chiffrement (sans accents) : Ceci est une cle eeeeeee
Début du déchiffrement de aes_tests/interstellar.txt_cbc_crypted_192 (CBC).

Fichier aes_tests/interstellar.txt_cbc_crypted_192 déchiffré (CBC - 192 bits) dans aes_tests/interstellar.txt_cbc_crypted_192_cbc_decrypted avec succès (0.01550s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % diff aes_tests/interstellar.txt aes_tests/interstellar.txt_cbc_crypted_192_cbc_decrypted
luka.mig@MBP-de-Lucas AES C %

```

AES 256

```

luka.mig@MBP-de-Lucas AES C % ./aes256
Usage : ./aes256 <crypt|decrypt|test> <ecb|cbc|iecb|icbc> <show|hide|file_key> [input_files]
Raccourcis :
- c = crypt
- d = decrypt
- t = test
- s = show
- h = hide
- iecb et icbc permettent de chiffrer des fichiers bmp avec l'entête en clair.
luka.mig@MBP-de-Lucas AES C % ./aes256 crypt ecb show aes_tests/interstellar.txt
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle tres longue :))
Début du chiffrement de aes_tests/interstellar.txt (ECB).

Fichier aes_tests/interstellar.txt chiffré (ECB - 256 bits) dans aes_tests/interstellar.txt_ecb_crypted_256 avec succès (0.01619s).
Entropie : 7.997365 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes256 decrypt ecb show aes_tests/interstellar.txt_ecb_crypted_256
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle t res
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle t re
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle t r
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle t
Entrez la clé de chiffrement (sans accents) : Ceci est une Cle tres longue :))
Début du déchiffrement de aes_tests/interstellar.txt_ecb_crypted_256 (ECB).

Fichier aes_tests/interstellar.txt_ecb_crypted_256 déchiffré (ECB - 256 bits) dans aes_tests/interstellar.txt_ecb_crypted_256_ecb_decrypted avec succès (0.01678s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % diff aes_tests/interstellar.txt aes_tests/interstellar.txt_ecb_crypted_256_ecb_decrypted
luka.mig@MBP-de-Lucas AES C % ./aes256 crypt cbc hide aes_tests/interstellar.txt
Entrez la clé de chiffrement (sans accents) : ****
Début du chiffrement de aes_tests/interstellar.txt (CBC).

Fichier aes_tests/interstellar.txt chiffré (CBC - 256 bits) dans aes_tests/interstellar.txt_cbc_crypted_256 avec succès (0.01701s).
Entropie : 7.997246 bits par octet.

luka.mig@MBP-de-Lucas AES C % ./aes256 decrypt cbc hide aes_tests/interstellar.txt_cbc_crypted_256
Entrez la clé de chiffrement (sans accents) : ****
Début du déchiffrement de aes_tests/interstellar.txt_cbc_crypted_256 (CBC).

Fichier aes_tests/interstellar.txt_cbc_crypted_256 déchiffré (CBC - 256 bits) dans aes_tests/interstellar.txt_cbc_crypted_256_cbc_decrypted avec succès (0.01746s).
Entropie : 4.696689 bits par octet.

luka.mig@MBP-de-Lucas AES C % diff aes_tests/interstellar.txt aes_tests/interstellar.txt_cbc_crypted_256_cbc_decrypted
luka.mig@MBP-de-Lucas AES C %

```

Récapitulatif

On peut en tableau suivant :

AES	Durée de chiffrement	Débit de chiffrement	Entropie	Durée de déchiffrement	Débit de déchiffrement	Fichiers égaux
128 - ECB	0.01399 s	4.97 Mo/s	7.997269 b/o	0.01464 s	4.75 Mo/s	Oui
128 - CBC	0.01496 s	4.65 Mo/s	7.997148 b/o	0.01427 s	4.87 Mo/s	Oui

AES	Durée de chiffrement	Débit de chiffrement	Entropie	Durée de déchiffrement	Débit de déchiffrement	Fichiers égaux
192 - ECB	0.01584 s	4.39 Mo/s	7.997228 b/o	0.01516 s	4.58 Mo/s	Oui
192 - CBC	0.01538 s	4.52 Mo/s	7.799708 b/o	0.01550 s	4.48 Mo/s	Oui
256 - ECB	0.01619 s	4.29 Mo/s	7.997365 b/o	0.01678 s	4.14 Mo/s	Oui
256 - CBC	0.01701 s	4.09 Mo/s	7.997246 b/o	0.01746 s	3.98 Mo/s	Oui

Nous pouvons ainsi voir les performances de nos différents AES. Il est logique que le chiffrement et déchiffrement en 256 bits ait une plus faible performance par rapport à aes 128 et 192.

CBC

Pour notre version de CBC, nous n'utilisons pas de vecteur d'initialisation étant donné que ceci n'était pas demandé. Cependant, nous avons appris lors de la dernière séance qu'un vecteur pouvait être demandé lors des tests.

Nous devons donc intégrer cette fonctionnalité pour rendre notre aes CBC complet.

Pour intégrer cette fonctionnalité, nous devons ajouter une fonction permettant d'entrer le vecteur en hexadécimal. Nous élargissons ceci pour permettre à l'utilisateur d'entrer une clé également en hexadécimal.

Pour cette dernière fonction, on utilise simplement une comparaison de l'octet écrit par l'utilisateur. On a :

```

while (i < 2*4*Nk) {
    c = getch();
    number = c >= '0' && c <= '9';
    if (number || (c >= 'a' && c <= 'f')) {
        printf("%c",c);
        i++;
        if ((i+1)%2 && i != 0) {
            oct_vect += c - (number?'0':87);
            key[oct_write] = oct_vect;
            oct_write++;
            printf(" ");
        } else
            oct_vect = (c - (number?'0':87)) << 4;
    }
}

```

En testant notre code sur aes 256 bits (le plus contraignant), on a :

Lorsque l'on affiche notre clé sur 128 bits, la clé est bien entré correctement.

Nous pouvons légèrement adapter cette fonction pour la demande du vecteur d'initialisation :

```

void ask_init_vect() {
    printf("Entrez le vecteur d'initialisation (en hexadécimal) : ");
    octet c, i = 0, oct_vect, oct_write = 0;
    int number;

    while (i < 32) {
        c = getch();
        number = c >= '0' && c <= '9';
        if (number || (c >= 'a' && c <= 'f')) {
            printf("%c", c);
            i++;
            if ((i+1)%2 && i != 0) {
                oct_vect += c - (number?'0':87);
                mem_cbc[oct_write] = oct_vect;
                oct_write++;
                printf(" ");
            } else
                oct_vect = (c - (number?'0':87)) << 4;
        }
    }
    printf("\n");
}

```

Ce code est bel est bien fonctionnel. Nous pouvons stocker le vecteur dans notre type matrix temporaire *mem_tmp*. Au vu de l'implémentation de notre code pour CBC, il n'y a rien d'autre à faire étant donné que le XOR se fera automatiquement pour le chiffrement et le déchiffrement.

Notons que le nom de ce type ne correspond plus à l'état actuel de notre code, étant donné que nous ne manipulons plus de matrice. Pour éviter tout problèmes, nous préférons ne pas modifier nos noms de types et de fonctions à ce stade.

Nous pouvons voir que le chiffrement par un vecteur d'initialisation nul (00 .. 00) avec CBC ou un chiffrement par ECB nous donne le même bloc.

	test_cbc_crypted_128	test_ecb_crypted_128
0	F71045E4 .	0 F71045E4 . E .
4	7568733E u	4 7568733E uhs>
8	F7337B27 .	8 F7337B27 .3{ '
12	1E099C3F .?	12 1E099C3F .?
16	8022E0F1 g . .	16 67A6B7B5 g . .

Lorsque l'on utilise un vecteur d'initialisation (ici 00 .. 00 01), on trouve bien un résultat différent :

	test_cbc_crypted_128	test_ecb_crypted_128
0	4BE697B2	0 F71045E4
4	4E58997E	4 7568733E
8	24D08AF1	8 F7337B27
12	17B8FAE8	12 1E099C3F
16	540BEC0B	16 67A6B7B5

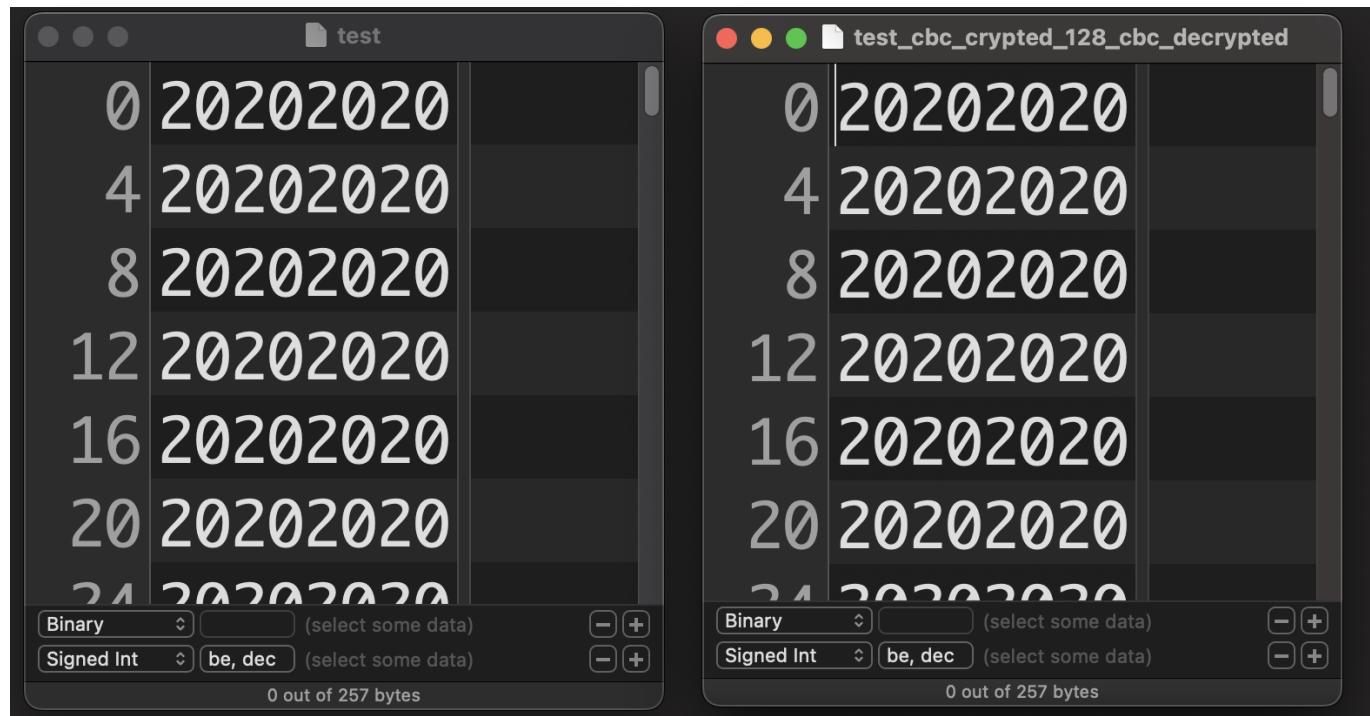
Ceci est cohérent.

Lorsque l'on déchiffre ce que l'on a obtenu, on a :

```
./aes128 d c key aes_tests/test_cbc_crypted_128
Entrez le vecteur d'initialisation (en hexadécimal) : 00 00 00 00 00 00 00
00 00 00 00 00 00 00 01
Début du déchiffrement de aes_tests/test_cbc_crypted_128 (CBC).
```

Fichier aes_tests/test_cbc_crypted_128 déchiffré (CBC – 128 bits) dans
aes_tests/test_cbc_crypted_128_cbc_decrypted avec succès (0.00036s).
Entropie : 0.000000 bits par octet.

Et on retrouve bien notre fichier de départ, avec le bon nombre d'octets comme l'atteste la capture (en bas de la capture) :



Nous n'avons pas besoin de revoir les autres implémentations d'AES étant donné que ce que nous avons modifié est indépendant de la taille de la clé.