INFO0030 : Projet de Programmation
Chiffrement d'Images

B. Donnet, K. Edeline, E. Marechal
Université de Liège

Alerte : Évitez de perdre bêtement des points...

— Nous vous conseillons **vivement** de relire et d'appliquer les guides de style et de langage, mis à votre disposition sur eCampus (INFO0030, Sec. Supports pour le Cours Théorique). Cela vous permettra d'éviter de nombreuses erreurs et de perdre des points inutilement.
— Nous vous conseillons de consulter la grille de cotation utilisée pour la correction des projets afin d'éviter de perdre bêtement des points. Elle est disponible sur eCampus (INFO0030, Sec. Procédures d'Évaluation).
— Votre code sera compilé et testé sur la machine virtuelle qui vous a été fournie et qui sert de référence. Elle est disponible sur eCampus (INFO0030, Sec. Projets). Veillez donc à ce que votre code fonctionne dans cet environnement.

1 Contexte

Les modes de communication digitale ont pris un essort important depuis, maintenant, presque 20 ans. La sécurité de ces moyens de communication a toujours été assurée (entre autre) par la *confidentialité*[1] de l'information qui est échangée entre les différents partenaires de la communication. Une façon d'assurer cette confidentialité est de *chiffer* le contenu de la communication, i.e., rendre illisible l'information sauf si une action spécifique est exécutée pour en autoriser l'accès.

Dans le cadre de ce projet, nous nous intéressons au chiffrement des images `PNM`. En effet, une image peut être décomposée en un tableau de points élémentaires appelés *pixels* (abréviation de *picture element*). Supposons que nous ne manipulons que des images en niveaux de gris (ou *grayscale* en anglais), i.e., des images dont les "couleurs" sont uniquement des nuances de gris. On peut représenter une telle image par une matrice d'entiers, dont la valeur des éléments représente l'intensité lumineuse des pixels de l'image. Par conséquent, une opération de chiffrement peut être réalisée en manipulant la matrice qui représente l'image.

Nous vous demandons, dans ce projet, d'implémenter un système de chiffrement d'images `PNM` basé sur un *registre à décalage à rétroaction linéaire* (*Linear Feedback Shift Register*, ou LFSR, en anglais). Il vous est aussi demandé, dans ce projet, de réutiliser votre librairie permettant de manipuler des images `PNM` (cfr. Projet 1).

2 Linear Feedback Shift Register

Un *Linear Feedback Shift Register* (LFSR) est un registre de bits (i.e., un tableau de bits – le Tableau 1 donne un exemple de registre de bits) sur lequel on réalise des opérations discrètes. A savoir :

— décaler tous les bits d'une position vers la gauche.
— remplacer le bit ainsi libéré par le résultat d'une opération logique (un "ou exclusif") entre le bit perdu (i.e., le bit de poids fort) et celui situé à une position particulière (généralement appelée *tap*) dans le registre. Cette position, le tap, est exprimée à partir du bit de poids faible.

1. De manière générale, on définit la confidentialité comme étant la propriété d'une donnée dont la divulgation doit être limitée aux personnes autorisées (cfr. B. Donnet, "INFO0045 : Introduction to Computer Security").

0										10
0	1	1	0	1	0	0	0	0	1	0

TABLE 1 – Exemple de registre à 11 bits. Le bit à l'indice 0 est le bit de poids fort. Similairement, le bit à l'indice 10 est le bit de poids faible.

Un objet LFSR dispose de trois paramètres qui caractérisent la séquence de bits qu'il produit : le nombre de bits N dans le registre, la graine (*seed* en anglais) initiale (il s'agit de la séquence de bits initiale du registre) et la valeur du tap. La Fig. 1 illustre une opération sur un LFSR de 11 bits initialisé avec la graine "01101000010" et un tap de 8.

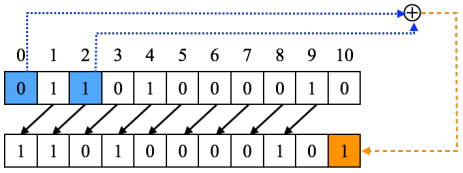


FIGURE 1 – Une opération sur le LFSR composé de 11 bits, initialisé avec la graine "01101000010" et pour un tap de 8. Le symbole $\oplus$ représente le ou exclusif.

Votre premier travail, dans ce projet, est d'écrire une librairie permettant de gérer un LFSR. En particulier, votre librairie devra cacher l'implémentation du LFSR dans un type opaque et permettre les fonctionnalités suivantes :

— initialisation. L'initialisation a pour but de créer un LFSR avec la graine et le tap initiaux. A noter que la taille du registre est donnée par la taille de la graine (qui se présente sous la forme d'une chaîne de caractères). Votre initialisation doit être robuste et résister aux cas où la graine et/ou la valeur du tap ne sont pas valides.

— transformation du contenu du registre en une chaîne de caractères. Cette fonctionnalité retourne une chaîne de caractères représentant l'état actuel du registre. Par exemple, si on applique cette fonctionnalité au résultat de l'opération illustrée à la Fig. 1, on obtient la chaîne de caractères "11010000101".

— réalisation d'une opération. Il s'agit de simuler une opération sur le registre (telle qu'illustrée à la Fig. 1). En outre, cette opération retourne la valeur du nouveau bit de poids faible (1 sur la Fig. 1). Le code suivant illustre l'opération sur le registre (le type `LFSR` est un type opaque représentant un LFSR) :

```
LFSR *lfsr = initialisation("01101000010", 8);
for(int i=0; i<10; i++){
  int bit = operation(lfsr);
  //affichage du registre résultat (avec la représentation en chaîne de
  //caractères du LFSR -- module string()) et du bit retourné
  printf("%s %d\n", string(lfsr), bit);
}//fin for - i
```

et devrait afficher à l'écran le résultat suivant :

```
11010000101 1
10100001011 1
01000010110 0
10000101100 0
00001011001 1
00010110010 0
00101100100 0
01011001001 1
10110010010 0
01100100100 0
```

— génération. Il s'agit de simuler k opérations sur le registre et retourner une valeur entière représentant les k bits générés par chacune des étapes. Obtenir cette valeur est assez simple : il suffit d'avoir une variable, initialisée à zéro et, pour chaque bit extrait à chaque opération, on double la valeur de la variable et on lui ajoute la valeur du bit retournée par l'opération. Par exemple,

si on effectue 5 opérations sur le registre et que les valeurs retournées sont (dans l'ordre) 1 1 0 0 1, la variable va prendre les valeurs 1, 3, 6, 12, 25 (cette dernière étant celle retournée par votre fonctionnalité). Par exemple :

```
LFSR *lfsr = initialisation("01101000010", 8);
for(int i = 0; i < 10; i++){
  int r = generation(lsfr, 5);
  printf("%s %d\n", string(lsfr), r);
}//fin for - i
```

devrait afficher à l'écran le résultat suivant :

```
00001011001 25
01100100100 4
10010011110 30
01111011011 27
01101110010 18
11001011010 26
01101011100 28
01110011000 24
01100010111 23
01011111101 29
```

En outre, nous vous demandons de valider votre librairie LFSR à l'aide de tests unitaires. Pour cela, vous utiliserez l'outil `seatest` vu au cours.[2] N'oubliez pas de joindre, dans votre archive, les fichiers `seatest.c` et `seatest.h`.

3 Chiffrement Basique d'Images

Votre tâche principale, dans ce projet, est d'écrire un module qui utilise vos librairies `PNM` (définie dans le projet 1) et LFSR (cfr. Sec. 2). Ce module doit permettre de chiffrer (i.e., rendre illisible) une image `PNM` à l'aide d'un registre LFSR.

Votre module devra donc offrir une fonctionnalité de *transformation* d'une image `PNM` sur base d'une certaine graine et d'une valeur de tap. Pour chaque pixel (x, y) de votre image[3], il suffit de réaliser un "ou exclusif" entre la valeur de ce pixel et la valeur obtenue après 32 générations du LFSR.

Attention, vous veillerez à parcourir votre image dans l'ordre de la trame (*raster-scan order* en anglais), à savoir $(0, 0)$, $(0, 1)$, $(0, 2)$, etc.[4]. Il est aussi important de noter que les images chiffrées ne respectent plus nécessairement à la lettre le format PGM, c'est à dire que la valeur maximale que peut prendre un pixel (255 pour le format PGM) peut être dépassée. Vous veillerez donc à relaxer votre lecture de fichier pour ne pas rejeter les images chiffrées.

Une fois compilé, votre module devra pouvoir être exécuté de la manière suivante :

```
$>./basic_cipher -i monalisa.pgm -o xmonalisa.pgm -s 01101000010100010000 -t 16
```

où l'option `-i` indique le fichier source (ici à chiffrer) l'option `-o` donne le nom du fichier résultat, l'option `-s` donne la graine et, enfin, `-t` donne la valeur du tap.

La Fig. 2 illustre le résultat du chiffrement (Fig. 2(b) – fichier `xmonalisa.pgm`) sur la figure `monalisa.pgm` (Fig. 2(a)) avec une graine "01101000010100010000" et une valeur de tap de 16.

La magie du système est la suivante. Si on exécute votre module de la façon suivante (opération de déchiffrement, donc) :

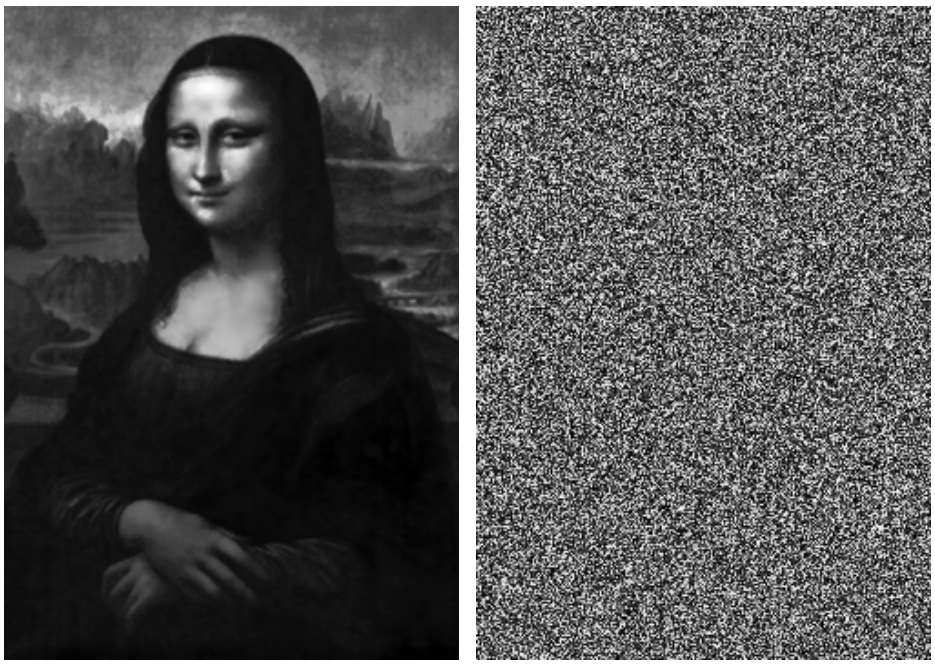
```
$>./basic_cipher -i xmonalisa.pgm -o monalisa.pgm -s 01101000010100010000 -t 16
```

on obtient la figure d'origine (soit la Fig. 2(a)).

Cela signifie donc que quiconque possède la graine et la valeur du tap peut retrouver l'image d'origine.

La page web du cours[5] contient plusieurs fichier `PGM` ayant déjà été chiffré pour vous. Le nom de chaque fichier indique la graine et la valeur du tap.

2. Le code source de `seatest` est disponible sur eCampus.
3. (x, y) sont les coordonnées du pixel. Par exemple $(0, 0)$ est le pixel en haut à gauche de l'image, $(0, 1)$ est le pixel sur la deuxième colonne, première ligne, de l'image, etc.
4. Pour en savoir plus sur le raster-scan order : https://en.wikipedia.org/wiki/Raster_scan
5. Voir eCampus, INFO0030, Sec. Projets/Projet 2: Chiffrement d'Images



(a) Image originale (`monalisa.pgm`) (b) Image chiffrée (`xmonalisa.pgm`)

FIGURE 2 – Chiffrement et déchiffrement d'une image.

Caractère	Valeur Décimale	Valeur Binaire
'A'	0	000000
'B'	1	000001
	...	
'W'	22	010110
	...	

TABLE 2 – Encodage des caractères en Base 64.

4 Chiffrement Avancé d'Images

Considérer un mot de passe binaire très court est une protection très faible. [6] D'un autre côté, un mot de passe binaire très long est assez pénible dans une utilisation quotidienne (mémorisation et encodage difficiles).

Dans cette troisième partie, nous allons améliorer notre système de chiffrement d'images en utilisant un mot de passe alpha-numérique (i.e., composé de lettres et chiffres). De façon à simplifier un peu les choses, le mot de passe sera encodé sous la forme d'une chaîne de caractères tirée d'un alphabet de 64 caractères :

```
1 char *base64 =
2 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
```

Chaque caractère du mot de passe alpha-numérique est encodé dans sa représentation binaire limitée à 6 bits. C'est la position du caractère dans l'alphabet qui permet la représentation binaire (illustrée dans le Tableau 2). Ainsi, par exemple, le caractère 'W' est en $22^{\text{ème}}$ position dans l'alphabet en base 64. La valeur binaire du chiffre 22, limitée à 6 bits, est "010110".

L'idée du chiffrement avancé est donc de construire la graine binaire, nécessaire au LFSR, à l'aide du mot de passe alpha-numérique. Chaque caractère sera transformé en sa chaîne binaire et concaténé à la

6. Voir le cours "Introduction to Computer Security" (INFO0045), Part 3, Chapter 1.

graine. Ainsi, par exemple, le mot de passe “MaitreGims” (10 caractères) sera transformé en la chaîne binaire suivante (60 caractères) : “001100011010100010101101101011011110000110100010100110101100”.

Il vous est demandé d'écrire un module permettant d'implémenter ce mode de chiffrement (et par extension, déchiffrement) avancé. Attention, inutile de faire du copier/coller de code. Vous penserez à invoquer les procédures/fonctions que vous avez déjà définies pour la gestion du LFSR et pour le chiffrement basique.

Une fois compilé, votre module devra pouvoir être exécuté de la manière suivante :

```
$>./advanced_cipher -i monalisa.pgm -o x-advancedmonalisa.pgm -p MaitreGims -t 16
```

où l'option `-i` indique le fichier source (ici à chiffrer), l'option `-o` donne le nom du fichier résultat, l'option `-p` donne le mot de passe alpha-numérique et, enfin, `-t` donne la valeur du tap.

Il est évident que l'exécution suivante :

```
$>./advanced_cipher -i x-advancedmonalisa.pgm -o monalisa.pgm -p MaitreGims -t 16
```

permet de déchiffrer l'image précédemment encodée avec le mot de passe “MaitreGims”.

5 Enoncé du Projet

Il vous est demandé d'écrire du code C permettant de chiffrer et déchiffrer des images `PGM` tel que cela a été décrit dans ce document.

Votre projet devra :

- être soumis dans une archive `tar.gz`, appelée `chiffrement.tar.gz`, via la Plateforme de Soumission. La décompression de votre archive devra fournir tous les fichiers nécessaires **dans le répertoire courant où se situe l'archive**. Vous penserez à joindre tous les codes sources nécessaires.
- réutiliser la librairie `PNM` que vous avez réalisé dans le précédent projet. Il est évident que l'équipe pédagogique s'attend à ce que vous ayez apporté des modifications à votre librairie `PNM` en fonction du feedback donné. A cette fin, vous nous fournirez le code source nécessaire de votre librairie `PNM`.
- être modulaire, i.e., nous nous attendons à trouver un (ou plusieurs) header(s) et un (ou plusieurs) module(s). Il est évident que votre projet doit contenir un programme utilisant le code écrit.
- appliquer les principes de la programmation défensive (vérification des préconditions, vérification des `malloc()`, ...). Pensez à libérer la mémoire allouée en cours de programmation afin d'éviter les fuites de mémoire.
- être parfaitement documenté. Vous veillerez à documenter correctement chaque header/fonction/-procédure/structure de données que vous définirez. Votre document suivra les principes de l'outil `doxygen`.[7]
- être validé par une librairie de tests unitaires. Pour cela, vous utiliserez l'outil `seatest` vu au cours.[8] N'oubliez pas de joindre, dans votre archive, les fichiers `seatest.c` et `seatest.h`. Ces tests unitaires porteront sur votre librairie `PNM` et la librairie relative au LFSR.
- implémenter une librairie de gestion d'un LFSR (Sec. 2), une librairie de chiffrement basique (Sec. 3) et de chiffrement avancé (Sec. 4).
- comprendre un `Makefile` permettant au moins de

  1. compiler vos tests unitaires pour votre librairie `PNM`. L'exécution de la commande

     ```
     $>make pnm_tests
     ```

     doit produire un fichier binaire, appelé `pnm_tests`, permettant de tester votre librairie `PNM`. Ce fichier binaire devra pouvoir être exécuté de la façon suivante :

     ```
     $>./pnm_tests
     ```

  2. compiler vos tests unitaires pour votre librarie LFSR. L'exécution de la commande

7. Vous veillerez, aussi, à documenter en `doxygen` votre librairie `PNM`.
8. Le code source de `seatest` est disponible sur la page Web du cours.

```
$>make lfsr_tests
```

doit produire un fichier binaire, appelé `lfsr_tests`, permettant de tester votre librairie implémentant un LFSR (Sec. 2). Ce fichier binaire devra pouvoir être exécuté de la façon suivante :

```
$>./lfsr_tests
```

3. de compiler et exécuter un chiffrement basique (Sec. 3). La commande

```
$>make basic_cipher
```

devra produire un fichier binaire `basic_cipher` pouvant être exécuté de la façon suivante (pour le chiffrement et le déchiffrement) :

```
$>./basic_cipher -i <input> -o <output> -s <seed> -t <tap>
```

Cette ligne de commande est expliquée à la Sec. 3.

4. de compiler et exécuter un chiffrement avancé (Sec. 4). La commande

```
$>make advanced_cipher
```

devra produire un fichier binaire `advanced_cipher` pouvant être exécuté de la façon suivante (pour le chiffrement et le déchiffrement) :

```
$>./advanced_cipher -i <input> -o <output> -p <password> -t <tap>
```

Cette ligne de commande est expliquée à la Sec. 4.

5. générer de la documentation. L'exécution de la commande

```
$>make doc
```

doit produire une documentation, au format HTML, dans le sous-répertoire `doc/`.

Il est impératif de respecter **scrupuleusement** les consignes sous peine de se voir attribuer une note de 0/20 pour non respect de l'énoncé.