# RSX112 : TP chiffrement



## Implémentation de l'algorithme RC4

A l'aide du pseudo-code vu en cours, réalisez une implémentation logicielle de l'algorithme RC4, sur la plate-forme (Windows, Unix…) et avec le langage de programmation de votre choix.

Une invocation du programme nécessite 3 paramètres :

- Le nom du fichier contenant la clé
- Le nom du fichier contenant les données source à traiter
- Le nom du fichier où le résultat sera stocké

Les deux premiers fichiers seront ouverts en lecture, le dernier en écriture.

De plus, un paramètre optionnel (« `-s` » ou « `--skipbytes` » ci-dessous) permettra de ne pas traiter les N premiers octets du fichier source, mais de les recopier tels quels dans le fichier destination – cette fonctionnalité sera utile pour la suite.

A-t-on besoin de prévoir un paramètre supplémentaire pour indiquer si l'on souhaite chiffrer ou déchiffrer les données ?

Exemple de syntaxe pour un utilitaire en ligne de commande :

```
% ./modecrypt.py -h
usage: modecrypt.py [-h] [-d] [-s SKIPBYTES] -c {rc4,aes-ecb,aes-cbc,aes-ctr}
                    [--decrypt] -k KEYFILE [--iv IV] -i INFILE -o OUTFILE

Encrypt or decrypt a file

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Enable debug diagnostics
  -s SKIPBYTES, --skipbytes SKIPBYTES
                        Number of bytes to copy unmodified (default: 0)
  -c {rc4,aes-ecb,aes-cbc,aes-ctr}, --cipher {rc4,aes-ecb,aes-cbc,aes-ctr}
                        Cipher to use
  --decrypt             Decrypt (default is to encrypt instead)
  -k KEYFILE, --keyfile KEYFILE
                        Key file name
  --iv IV               Initialization Vector file name (CBC only)
  -i INFILE, --infile INFILE
                        Input file name
```

```
  -o OUTFILE, --outfile OUTFILE
                        Output file name
```

Pour vous assurer que votre implémentation de RC4 est correcte, vous pouvez vérifier que :

- le fichier résultat doit faire exactement la même taille (en octets) que le fichier source
- le déchiffrement d'un fichier chiffré par votre programme est identique à l'original
- vous obtenez les mêmes résultats pour les 3 « vecteurs de test » fournis dans l'annexe A page 7 du draft RFC suivant : http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt

Pour vous faciliter la tâche, ces 3 vecteurs de test sont convertis en binaire dans le répertoire `rc4-test-vectors` : pour chacun des 3 tests $i \in \{1, 2, 3\}$, le fichier `t`*i*`-key.bin` contient la clé RC4, `t`*i*`-msg.bin` les données d'origine et `t`*i*`-enc.bin` les données chiffrées.

Ces fichiers binaires ont été générés depuis le texte de la spécification via le programme `hex-write.py` ci-joint, qui peut vous servir de squelette pour une implémentation `rc4.py` en Python, en particulier pour l'analyse de la ligne de commande :

```
% ./hex-write.py -h
usage: hex-write.py [-h] -i INFILE -o OUTFILE

Write a binary file from its hexadecimal contents.

optional arguments:
  -h, --help            show this help message and exit
  -i INFILE, --infile INFILE
                        Input file name
  -o OUTFILE, --outfile OUTFILE
                        Output file name
```

Deux fichiers binaires peuvent être comparés en ligne de commande avec la commande « `comp` *fichier1 fichier2* » sous DOS/Windows, et avec « `diff` » ou « `cmp` » sous Unix.

Le contenu des fichiers binaires peut être examiné en hexadécimal avec

- « `hexdump -C` » sous Unix
- Un utilitaire tel que « Free Hex Editor Neo 5.01 » sous Windows : http://www.hhdsoftware.com/free-hex-editor

Attention à ne pas rajouter de retours à la ligne superflus dans les fichiers, en particulier le fichier contenant la clé… Chaque octet présent dans le fichier fait partie de la clé ou des données.

Un exemple de programme `rc4.py` sera fourni en Python – il fonctionne aussi bien sous Unix (où Python 3 est souvent pré-installé) que sous Windows, où Python 3 peut être téléchargé depuis http://www.python.org/download/

## Chiffrement d'une image

L'objectif de cette partie est de chiffrer une image et de visualiser le résultat. Pour ce faire, la solution la plus simple est :

- de choisir un format bitmap « BMP » non compressé, d'origine Windows mais lisible sur toute plateforme
- de ne pas chiffrer l'en-tête du fichier, afin que les métadonnées sur l'image soient préservées et sensées, et ainsi que l'image « chiffrée » soit affichable par les programmes graphiques courants

L'image de départ est le fichier `penguin.bmp`, contenant les octets suivants :

```
% hexdump -C penguin.bmp | head
00000000  42 4d 06 91 17 00 00 00  00 00 36 00 00 00 28 00  |BM........6...(.|
00000010  00 00 94 02 00 00 0c 03  00 00 01 00 18 00 00 00  |................|
00000020  00 00 d0 90 17 00 13 0b  00 00 13 0b 00 00 00 00  |................|
00000030  00 00 00 00 00 00 ff ff  ff ff ff ff ff ff ff ff  |................|
00000040  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
0000cb60  ff ff ff ff ff ff ff ff  ff ff fe fe fe ff ff ff  |................|
```

L'en-tête s'interprète ainsi, les positions des octets ou décalages (« offsets ») étant indiqués entre parenthèses :

- (0x0000) Les deux premiers octets, « `42 4d` » en hexadécimal soit les caractères « BM », indiquent qu'il s'agit d'une image bitmap au format BMP.
- (0x0002) 32 bits : taille du fichier en octets : « `06 91 17 00` », octet de poids faible en premier, soit en remettant les octets à l'endroit 0x00179106 ou 1 544 454 octets.
- (0x0006) 4 octets réservés.
- (0x000A) 32 bits : position de l'octet où les données (pixels) de l'image débutent, ici 0x00000036 soit le 54e octet, début de la série des « `ff` ».
- (0x000E) 32 bits : taille de l'en-tête DIB à partir de cette position, soit 0x00000028 ou 40 octets supplémentaires jusqu'à l'octet 54, début de l'image.
- (0x0012) 32 bits : largeur de l'image, soit 0x00000294 ou 660 pixels.
- (0x0016) 32 bits : hauteur de l'image, soit 0x0000030c ou 780 pixels.
- (0x001A) 16 bits : nombre de plans de couleur, doit avoir la valeur 0x0001.
- (0x001C) 16 bits : nombre de bits par pixel, 0x0018 soit 24 bits.
- (0x001E) 32 bits : méthode de compression utilisée, 0x00000000 soit aucune.
- (0x0022) 32 bits : taille des données bitmap, 0x001790d0 ou 1 544 400 octets, soit la taille du fichier sans les en-têtes de 54 octets.
- (0x0026) 32 bits : résolution horizontale de l'image : 0x00000b13 ou 2835 pixels par mètre
- (0x002A) 32 bits : résolution verticale de l'image : 0x00000b13 ou 2835 pixels par mètre
- (0x002E) 32 bits : nombre de couleurs dans la palette, ici 0.
- (0x0032) 32 bits : nombre de couleurs importantes, ici 0 (ignoré).

Les données bitmap proprement dites commencent donc à l'offset 54 (0x00000036).

Chiffrez l'image `penguin.bmp` via l'algorithme RC4 et la clé de votre choix, par exemple la clé de 128 bits contenue dans le fichier t3-key.bin, en laissant inchangés les 54 premiers octets de l'en-tête BMP/DIB :

```
% ./rc4.py -s 54 -k t3-key.bin -i penguin.bmp -o penguin-rc4.bmp
```

puis affichez l'image `penguin-rc4.bmp` résultat.

## Chiffrement par bloc

Trouvez une implémentation d'AES (par exemple, ou tout autre algorithme de chiffrement par bloc) pour votre plateforme : par exemple, sous Ubuntu Linux 12.10, le package python-crypto est installé par défaut pour l'environnement Python 3. Le code source peut également être récupéré sur https://www.dlitz.net/software/pycrypto/

Modifiez votre programme précédent pour ajouter une option « `-c` » permettant de choisir l'algorithme de chiffrement à utiliser, et implémentez (en plus de RC4) le chiffrement AES en mode ECB, CBC et CTR.

Chiffrez l'image `penguin.bmp` via RC4, AES (et la clé de votre choix, 128 bits par exemple) en mode ECB, CBC et CTR, en laissant inchangés les 54 premiers octets de l'en-tête BMP/DIB, et visualisez le résultat. Que remarquez-vous ?