TP1 – Cryptographie symétrique

Exercice 1. *GnuPG et openSSL*

Les logiciels libres GNUPG (*GNU Privacy Guard*) et openSSL permettent d'utiliser de nombreuses primitives cryptographiques. On en explore quelques unes dans cet exercice. Pour cela, on utilise les utilitaires en ligne de commande `gpg` et `openssl`.

Remarque. Ces deux logiciels fournissent beaucoup de fonctionnalités, dont de très nombreuses fonctionnalités de cryptographie asymétrique. Dans cet exercice, on s'intéresse à quelques fonctionnalités très basiques, uniquement de cryptographie symétrique.

1. **i.** Chiffrer un fichier avec la commande `gpg -c --cipher-algo AES --no-sym-key-cache MonFichier.txt`. Comment est sauvegardé le résultat ? *Note. L'option `--no-sym-key-cache` permet simplement d'avoir à rentrer le mot de passe à chaque fois, et éviter qu'il ne soit retenu en cache par le logiciel.*
   ii. Déchiffrer le fichier chiffrer avec la commande `gpg -d ...`. Que se passe-t-il en cas de mauvais mot de passe ?
2. **i.** Répéter les mêmes questions avec `openssl`. Le chiffrement se fait avec la commande `openssl enc -aes-128-cbc -in Fichier.txt -out Fichier.txt.enc` par exemple, et le déchiffrement avec `openssl dec -aes-128-cbc --in Fichier.txt.enc`.
   ii. Que signifie les lettres `aes-128-cbc` dans les commandes précédentes ? Que peut-on choisir à la place (utiliser `openssl enc -help`) ? que se passe-t-il si on déchiffre avec un autre algorithme que celui de chiffrement ?
   iii. Chiffrer le fichier `tux.ppm` avec différents algorithmes : `aes-128-cbc`, `aes-128-ecb`, `aes-128-ctr`, `aes-128-ofb`. Ajouter aux fichiers chiffrés l'en-tête du fichier d'origine (les trois premières lignes). Ouvrir les fichiers obtenus avec un visionneur d'image et commenter.
3. **i.** Calculer une étiquette liée à un fichier à l'aide de `openssl dgst -hmac <CLEF> Fichier.txt`.
   ii. Comment peut-on vérifier si la signature est correcte ?
   iii. Quelle est la bonne solution si on souhaite à la fois chiffrer et signer un fichier avec OpenSSL ?
4. Récuperer les fichiers `mystere1.txt.enc`, `mystere2.txt.enc`, ... ainsi que leurs MAC. La clef que l'on partage est `HAI709I`. Quel(s) fichier(s) est (sont) correct(s) ?

Exercice 2. *Les modes opératoires*

Le chiffrement par bloc utilise différents modes opératoires (dont ECB, CBC, OFB, CTR). On souhaite implanter ces modes en Python. On va utiliser la bibliothèque `pycryptodome` qui fournit de nombreuses primitives cryptographique. Pour l'installer sur les machines de la FdS, entrer la commande suivante avant de commencer l'exercice :

```
~$ pip3 install pycryptodome
```

Note. La bibliothèque fournit beaucoup plus de fonctions que celles qu'on utilise dans ce TP. En particulier, pour la plupart des questions qui suivent, il existe une fonction dans la bibliothèque qui effectue la tâche demandée. L'objectif n'est pas d'utiliser ces fonctions, mais de les coder soi-même. En pratique, vous devez compléter le fichier `TP1.py` fourni, en ne chargeant aucune autre fonction que celles déjà chargées.

Type de données. On souhaite manipuler des données sous forme de bits, ou plus précisément d'octets. Pour cela, on utilise les *bytestrings* Python. Une *bytestring* s'écrit comme une chaîne de caractère, préfixée du caractère « `b` » : par exemple, `m = b'ceci est une bytestring'`. Les caractères de la chaîne sont soit des caractères ASCII, soit directement un octet quelconque spécifié par ses huit bits. Pour cela, on écrit `\xhh` où `hh` est l'écriture hexadécimale (base 16) de l'entier représenté par les huit bits. Par exemple, l'octet `00101101` s'écrit `2d` en hexadécimal (car `1101` vaut 13, et que `a`, `b`, ..., `f` représentent 10, 11, ..., 15). Pour entrer cet octet, on écrit donc `b'\x2d'` dans une *bytestring*. De manière assez surprenante, si `s` est une *bytestring*, la syntaxe `s[i]` renvoie le caractère d'indice i *sous la forme d'un entier* : par exemple, si `s = b'\x2d'`, `s[0]` renvoie l'entier 45. À l'inverse, `s[a:b]` renvoie bien une *bytestring*. La fonction `get_random_bytes(n)` permet de générer une *bytestring* alétoire de n octets.

1. *Jouer avec les bytestrings* : définir la chaîne `s = b'Cryptographie'` et essayer de répondre aux questions ci-dessous, avant de tester en pratique le résultat.
   - **i.** Que renvoient `s[0]`, `s[4]`, `s[-2]` ? Vérifier la cohérence avec l'encodage ASCII.
   - **ii.** Que renvoie `s[7:10]` ?
   - **iii.** Comment tester si une *bytestring* commence par le caractère `x` ?

Pour implanter les modes opératoires, on fournit dans `TP1.py` les fonctions `Enc` et `Dec` qui prennent en entrée un bloc de 128 bits (16 octets) et une clef de la même taille, et renvoient ce bloc chiffré ou déchiffré grâce à AES.

2. Pour utiliser un mode opératoire, la chaîne en entrée doit posséder un nombre de bits qui soit un multiple de 128. Il faut donc *padder* la chaîne en entrée : on rajoute les **bits** `10...0` de sorte à ce que la chaîne ait une longueur multiple de 128 (en bits !). On rajoute au moins un `1`, donc si la longueur de la chaîne est un multiple de 128 bits, on l'allonge de 128 bits exactements.
   - **i.** Écrire une fonction `pad(s)` qui prend en entrée une *bytestring* `s` et renvoie une *bytestring* paddée.
   - **ii.** Écrire une fonction `unpad(s)` qui retire le *padding* de `s`. *Si `s` n'a pas été paddée correctement, la fonction soulève une exception, grâce à l'instruction* `raise ValueError("message non correctement paddé")`.
   - **iii.** Tester en ajoutant du *padding* puis en le retirant, à des chaînes de diverses longueurs.

3. - **i.** Implanter le chiffrement dans le mode opératoire ECB, sous la forme d'une fonction `Enc_ECB(clair, clef)` qui renvoie le chiffré de `clair` avec la `clef`, en utilisant AES en mode ECB.
   - **ii.** Implanter de même le déchiffrement, sous la forme d'une fonction `Dec_ECB(chiffre, clef)` qui renvoie le clair de `chiffre` avec la `clef`, en utilisant AES avec le mode ECB.
   - **iii.** Tester vos fonctions en chiffrant et déchiffrant des messages !
4. - **i.** Répéter la question précédente (**i.**, **ii.** et **iii.**) avec le mode CBC. Le vecteur d'initialisation devra être tiré aléatoirement et fourni dans le résultat. *Pour effectuer un XOR bit-à-bit de deux bytestrings, utiliser la fonction fournie.*
   - **ii.** Faire à nouveau de même avec le mode OFB.
5. Enfin, on effectue l'implantation du mode CTR. Au lieu d'utiliser la construction $IV \| i$ du cours, on utilise à la place $IV \oplus i$.
   - **i.** Quelle est le nombre maximal de bloc qu'on peut gérer en mode CTR ? Estimer la taille de fichier que cela représente.
   - **ii.** Implanter le chiffrement et déchiffrement en mode CTR. *Utiliser la méthode `long_to_bytes(i, 16)` pour transformer i en bytestring de 128 bits.*

Exercice 3. *Attaque sur ECB*

On utilise dans cet exercice les fonctions de chiffrement et déchiffrement en mode ECB, écrites à l'exercice précédent.

1. Chiffrer le fichier `tux.ppm` avec AES en mode ECB et une clef aléatoire. *Pour lire le fichier (sous forme de bytestring), on peut utiliser* `clair = open('tux.ppm', 'rb').read()`.
2. Écrire le résultat du chiffré dans un fichier `tux.ecb.ppm`. *Pour écrire dans le fichier, on peut utiliser* `open('tux.ecb.ppm','wb').write(chiffre)`. En rajoutant un en-tête adapté, visualiser le résultat.

On cherche maintenant à trouver quelle image mystère se cache dans `image.ecb.ppm`. La difficulté est qu'on ne connaît pas[1] les dimensions de l'image et donc l'en-tête à rajouter.
L'en-tête d'une image au format PPM possède trois lignes : la première contient les caractères `P6` (pour indiquer le format de fichier), la seconde contient les dimensions en pixels, et la troisième la valeur maximale d'un pixel (255 dans notre cas). La suite contient les pixels, ligne par ligne, codés chacun sur trois octets (pour les trois couleurs).

3. - **i.** Si on dispose d'une image de dimension $L \times H$ en pixels, quelle est la taille du fichier en octet, en ignorant l'en-tête ?
   - **ii.** Dans l'autre sens, si on dispose d'un fichier de N octets, de combien de pixels est constituée l'image (environ, en estimant vaguement la taille de l'en-tête) ?
   - **iii.** Si on sait qu'une image fait N octets, quelles peuvent être ses dimensions ?

1. Enfin, *vous* ne connaissez pas… moi si !

4. À l'aide de la question précédente, et d'un peu de tâtonnement, trouver quelle image se cache derrière le fichier `image.ecb.ppm`. *Calculer le nombre approximatif de pixels de l'image, estimer alors assez finement la taille en octet de l'en-tête, puis raffiner le calcul du nombre de pixels (qui est toujours une approximation à cause du padding). Produire beaucoup de fichiers de différentes dimensions et les parcourir visuellement. Remarque : pour ne pas saturer l'espace disque, produire les images dans le dossier temporaire `/tmp`.*

Exercice 4. *Collisions*

On cherche à produire des *collisions partielles* pour des fonctions de hachage. Si h est une fonction de hachage, (x^0, x^1) est une collision partielle de taille k si les k premiers bits de leurs hachés sont égaux, c'est-à-dire si $h(x^0)_{[1,k[} = h(x^1)_{[1,k[}$. On ne cherche pas des collisions partielles quelconque : on veut imposer que les deux mots x^0 et x^1 aient un préfixe commun. Par exemple, on fixe que x^0 et x^1 commence par la chaîne `b'Collision:'`.
Pour trouver une collision partielle, on utilise l'algorithme suivant : on tire des x^i aléatoires (qui commencent par le préfixe qu'on souhaite) jusqu'à en trouver deux dont les k premiers bits des hachés sont égaux.
L'objectif de l'exercice est d'écrire une fonction `RechercheCollision(h, prefixe, nbits)` qui trouve une collision partielle de `nbits` bits, pour la fonction `h`, entre deux mots de 128 bits qui commencent par `prefixe` (qui est une *bytestring*). On souhaite de plus que les mots soient constitués de caractères ASCII *affichables*, c'est-à-dire de caractère dont le code est compris entre 31 et 126 inclus. On appliquera la fonction avec `h` qui vaut `md5` ou `sha1` ou `sha3_256` de la bibliothèque `hashlib`, qui s'utilisent sous la forme `h(mot).digest()`.
Indications :

- on peut utiliser un dictionnaire Python pour stocker les x^i et leurs hachés ;
- écrire une fonction qui produit un mot aléatoire affichable de longueur n, en utilisant `randrange` et `long_to_bytes` ;
- dans un premier temps, on peut supposer que `nbits` est un multiple de 8 pour simplifier, puis faire le cas général.

Exemple :

```
>>> x0, x1 = RechercheCollision(md5, b'Collision:', 16)
>>> x0, x1
(b'Collision:QQ9^]i', b'Collision:[T~:Hi')
>>> md5(x0).digest()
b'\xcb\x87g\x1c\x90iB\x98a\xb2\x18\x0c\xef\xe6\x9dQ'
>>> md5(x1).digest()
b'\xcb\x87\xb8\x81\xe1\xa2\x1ba0\xd8UQx&\x9fH'
```