TP - Sécurisation des communications

M. Tellene

Conseil d'organisation : il vous est conseillé pour ce TP de créer dans votre dossier personnel, un dossier « securisation communications ». C'est ce dossier qui contiendra le travail demandé lors de ce TP.

1 Cryptographie symétrique

Créer un premier fichier `crypto_symetrique.py`. C'est dans ce fichier que devra être écrite toutes vos fonctions de cette partie. Pour faciliter la conception, nous considérerons que tous les messages en clair sont uniquement constitués de lettres majuscules.

1.1 Codage des cryptosystèmes

Exercice 1

Écrire une fonction `codage_cesar()`. Cette fonction prend en argument un message en clair `m` et le décalage `k`. La fonction devra renvoyer un texte `c`, chiffré avec la méthode César. Vous **devrez** passer par le code `ASCII` afin de chiffrer vos message.

```
>>> codage_cesar("PROGRAMMATION", 6)
VXUMXGSSGZOUT
```

Exercice 2

Écrire une fonction `decodage_cesar()`. Cette fonction prend en argument un message chiffré `c` et le décalage `k`. Cette fonction devra renvoyer le message en clair `m`, déchiffré à l'aide de la méthode de César.

```
>>> decodage_cesar("VXUMXGSSGZOUT", 6)
PROGRAMMATION
```

Exercice 3

Écrire une fonction `codage_xor()`. Cette fonction prend en argument un message en clair `m` et la clé de chiffrement `k`. Cette fonction devra renvoyer le message chiffré `c` (sous forme de tableau). Le chiffrement se fera en suivant la méthode du codage `xor`. Il ne sera pas nécessaire de renvoyer le texte avec des caractères, une liste de chiffres suffit.

```
>>> codage_xor("PROGRAMMATION", "NSI")
[30, 1, 6, 9, 1, 8, 3, 30, 8, 26, 26, 6, 0]
```

Exercice 4

Écrire une fonction `decodage_xor()`. Cette fonction prend en argument un message chiffré `c` (sous forme de tableau) et la clé de chiffrement `k`. Cette fonction devra renvoyer le message en clair `m`, en suivant la méthode du décodage `xor`.

```
>>> decodage_xor([30, 1, 6, 9, 1, 8, 3, 30, 8, 26, 26, 6, 0], "NSI")
PROGRAMMATION
```

1.2 A l'attaque des cryptosystèmes symétriques

Une première approche pour cracker des cryptosystèmes est la méthode brute, aussi appelée brute force. Cette méthode consiste à tester toutes les combinaisons possibles. Prenons l'exemple du code César. Ce cryptosystème repose sur le décalage de lettre, étant donné qu'il n'y que 26 lettres dans l'alphabet, il y a 26 combinaisons à tester ! Un travail simple pour un ordinateur.

Exercice 5

Écrire une fonction `attaque_brute_force_cesar()`. Cette fonction prend en argument un message chiffré `c` et renvoie un tableau contenant tous les messages en clair possibles.

```
>>> attaque_force_brute_cesar("VXUMXGSSGZOUT")
['VXUMXGSSGZOUT', 'UWTLWFRRFYNTS', 'TVSKVEQQEXMSR', 'SURJUDPPDWLRQ', '
    RTQITCOOCVKQP', 'QSPHSBNNBUJPO', 'PROGRAMMATION', 'OQNFQZLLZSHNM', '
    NPMEPYKKYRGML', 'MOLDOXJJXQFLK', 'LNKCNWIIWPEKJ', 'KMJBMVHHVODJI', '
    JLIALUGGUNCIH', 'IKHZKTFFTMBHG', 'HJGYJSEESLAGF', 'GIFXIRDDRKZFE', '
    FHEWHQCCQJYED', 'EGDVGPBBPIXDC', 'DFCUFOAAOHWCB', 'CEBTENZZNGVBA', '
    BDASDMYYMFUAZ', 'ACZRCLXXLETZY', 'ZBYQBKWWKDSYX', 'YAXPAJVVJCRXW', '
    XZWOZIUUIBQWV', 'WYVNYHTTHAPVU']
```

Exercice 6

Il est possible d'améliorer cette attaque brute force. En effet, une attaque consiste à faire une analyse fréquentielle d'apparition des lettres. Écrire une fonction `attaque_analyse_frequentielle_cesar()`, qui prend en argument un message chiffré `c` et qui renvoie un tableau avec les messages clairs possibles, du plus probable au moins probable. Dans notre cas, nous allons :

1. générer un dictionnaire des occurrences des lettres de `c`
2. trier ce dictionnaire en mettant les éléments qui apparaissent le plus fréquemment au début
3. parcourir le dictionnaire par les clés et pour chaque clé, calculer le décalage avec la lettre qui apparaît le plus dans la langue française
4. lancer le décodage avec le pas calculé et mettre le résultat dans un tableau
5. renvoyer le tableau une fois que toutes les clés ont été parcourues

```
>>> attaque_analyse_frequentielle_cesar("VXUMXGSSGZOUT")
['CEBTENZZNGVBA', 'FHEWHQCCQJYED', 'TVSKVEQQEXMSR', 'HJGYJSEESLAGF', '
    EGDVGPBBPIXDC', 'NPMEPYKKYRGML', 'ACZRCLXXLETZY', 'LNKCNWIIWPEKJ', '
    GIFXIRDDRKZFE']
# avec cet exemple nous pouvons voir que cette méthode possède ses limites
```

Une fois votre code fonctionnel, déterminer le décalage et le message en clair des messages chiffrés suivants, il se peut que `attaque_analyse_frequentielle_cesar()` ne donne rien :

- `OLBYLBZLTLUA`
- `SDKKDMD`
- `DGSXCPITJG`
- `ZBYQBKWWKDSYX`
- `CPOKPVS`
- `GLDMPKYRGOSC`

Essayons de cracker le chiffrement `xor`. Lorsque l'on ne connaît que le message chiffré dans un chiffrement `xor` et que l'on ne dispose d'aucune autre information, il n'y a pas beaucoup d'attaques possibles qui permettent de récupérer la clé de chiffrement ou le message en clair. En effet, le chiffrement `xor` est réputé pour sa sécurité si la clé est choisie de manière aléatoire et est suffisamment longue.

Cependant, il existe tout de même une attaque possible : l'attaque par force brute. Si la clé de chiffrement est une chaîne de caractères plutôt qu'un nombre entier, une attaque par force brute est possible, mais elle est beaucoup plus complexe. En effet, pour chaque caractère de la clé, il y a 256 possibilités (car chaque caractère peut prendre n'importe quelle valeur de 0 à 255 en `ASCII` ou en `Unicode`), ce qui rend le nombre total de clés possibles très élevé.

Si la longueur de la clé est connue, alors l'attaquant peut tester toutes les combinaisons possibles de caractères de la longueur de la clé. Par exemple, si la clé est de longueur 5, il y a $256^5 = 1\,099\,511\,627\,776$ clés possibles à tester.

Cette attaque par force brute nécessite de connaître la longueur de la clé de chiffrement. Si la longueur de la clé de chiffrement est inconnue, l'attaquant devra tester toutes les longueurs de clé possibles, ce qui rend l'attaque encore plus difficile et peut prendre beaucoup plus de temps.

Exercice 7

Écrire une fonction `attaque_force_brute_xor()`. Cette fonction prend en argument un message chiffré `c` et renvoie un dictionnaire où les clés seront les différentes clés générées et les valeurs seront les messages en clair déchiffrés à partir de `c` et des différentes clés. Pour simplifier, nous partirons du principe que la clé de chiffrement est de taille 2.

```
>>> attaque_force_brute_xor([4, 26, 11, 27, 31, 25, 12, 0, 4, 5, 24, 17])
{
    'AA': 'E[JZ^XMAEDYP', 'AB': 'EXJY^[MBEGYS', 'AC': 'EYJX^ZMCEFYR',
    'AD': 'E^J_^]MDEAYU', 'AE': 'E_J^^\\MEE@YT', 'AF': 'E\\J]^_MFECYW',
    'AG': 'E]J\\^^MGEBYV', 'AH': 'ERJS^QMHEMYY', 'AI': 'ESJR^PMIELYX',
    ...
}
```

Le message en clair `m` est présent dans notre dictionnaire, mais rechercher ce message à la main est très fastidieux (surtout qu'ici, nous savons que la clé est de longueur 2, imaginez si la clé fait 10 caractères de long ou que nous ne connaissons pas sa taille !). Nous allons donc essayer d'automatiser notre recherche.

Exercice 8

Complétant la fonction `recherche_automatique()` du fichier `ressources.py`. Cette fonction

parcourt un fichier texte (`tous_les_mots.txt`) contenant tous les mots de la langue française. Cette fonction doit afficher toutes les clés qui ont pu être utilisées pour créer le message chiffré `c`.

Indiquer la clé de chiffrement du message `c = [4,26,11,27,31,25,12,0,4,5,24,17]`

2 Cryptographie asymétrique

Créer un premier fichier `crypto_asymetrique.py`. C'est dans ce fichier que devra être écrite toutes vos fonctions de cette partie. Dans un premier temps, nous chiffrerons des nombres, puis nous chiffrerons des messages.

2.1 Importation du module

Afin de nous simplifier la tâches, nous allons utiliser un module Python permettant de faire des actions liées à la cryptographie asymétrique, notamment la génération de grands entiers premiers.

1. Ouvrir un terminal en faisant `ctrl` + `alt` + `t`
2. Taper la commande `python3`
3. Une fois dans l'interpréteur Python, écrire l'instruction suivante : `import Crypto.Util.number`
4. S'il n'y a pas d'erreur, vous pouvez passer à la partie suivante, sinon écrire (toujours dans l'interpréteur) l'instruction `quit()`
5. Vous devriez être revenu dans le terminal, taper `pip3 install -U PyCryptodome`
6. Laisser le téléchargement se faire puis réessayer d'importer le module, cela devrait fonctionner

2.2 Codage de `RSA`

Le cryptosystème `RSA` est composé de trois fonctions :
- une fonction de génération de clé
- une fonction de chiffrement
- une fonction de déchiffrement

Exercice 9
Écrire une fonction `key_gen()` qui va générer une paire de clé. Cette fonction prend en argument un nombre de bits (très grand) et renvoie la paire `(clé publique, clé privée)`.

Afin de générer les valeurs de `p` et `q`, vous utiliserez la fonction `getPrime()` du module `Crypto.Util.number`. De plus, vous fixerez la valeur `e = 65537` [1].

Exercice 10
Écrire une fonction `encrypt()` qui prend en argument un message en clair `m` ainsi que la clé publique. Cette fonction doit renvoyer un message `c` chiffré à l'aide de la clé publique.

Exercice 11
Écrire une fonction `decrypt()` qui prend en argument un message chiffré `c` ainsi que la clé privée. Cette fonction doit renvoyer le message en clair `m` déchiffré à l'aide de la clé privée.

1. La valeur `e = 65537` est couramment utilisée pour le chiffrement RSA car elle satisfait deux conditions importantes : `e` doit être un grand nombre premier et `e` doit être relativement petit par rapport à $\phi(n)$

Exercice 12

Tester vos fonctions. Pour ce faire :

1. vous générerez les clés sur 1024 bits
2. vous générerez un message `m` en utilisant la fonction `getrendbits` du module `random`
3. vous créerez une variable `c`, chiffrement du message `m`
4. vous déchiffrerez `c` afin d'avoir `m2`
5. vous réaliserez l'affichage suivant pour vérifier que votre cryptosystème fonctionne :

```
print(f"Message original : {m}")
print(f"Message chiffré : {c}")
print(f"Message déchiffré : {m2}")
```

Vous pouvez effectuer les tests plusieurs fois avec la même valeur `m`, vous verrez que le message chiffré `c` sera différent.

Avec `RSA`, nous n'avons chiffré que des nombres, passons maintenant aux messages. Le mode de fonctionnement est le même que pour les cryptosystèmes symétriques : utiliser le code `ASCII`.

Exercice 13

Écrire une fonction `encrypt_texte()` qui prend en argument le clé publique `kp` et un message en clair `m`. Cette fonction devra renvoyer un tableau contenant le chiffrement du code `ASCII` de chaque caractère du message.

Exercice 14

Écrire une fonction `decrypt_texte()` qui prend en argument le clé privée `kpr` et un message chiffré `c` (sous forme de tableau). Cette fonction devra renvoyer le message en clair `m`.

Exercice 15

Comme l'exercice 12, tester vos fonctions. Vous pouvez effectuer les tests plusieurs fois avec le même texte `m`, vous verrez que le message chiffré `c` sera différent.