Cryptage RSA

Présentation

Le chiffrement RSA (nommé par les initiales de ses trois inventeurs) est un algorithme de cryptographie asymétrique, très utilisé dans le commerce électronique, et plus généralement pour échanger des données confidentielles sur Internet.

Le chiffrement RSA est asymétrique : il utilise une paire de clés (des nombres entiers) composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles. Les deux clés sont créées par une personne, souvent nommée par convention Alice, qui souhaite que lui soient envoyées des données confidentielles. Alice rend la clé publique accessible. Cette clé est utilisée par ses correspondants (Bob, etc.) pour chiffrer les données qui lui sont envoyées. La clé privée est quant à elle réservée à Alice, et lui permet de déchiffrer ces données. La clé privée peut aussi être utilisée par Alice pour signer une donnée qu'elle envoie, la clé publique permettant à n'importe lequel de ses correspondants de vérifier la signature.

Une condition indispensable est qu'il soit « calculatoirement impossible » de déchiffrer à l'aide de la seule clé publique, en particulier de reconstituer la clé privée à partir de la clé publique, c'est-à-dire que les moyens de calcul disponibles et les méthodes connues au moment de l'échange (et le temps que le secret doit être conservé) ne le permettent pas.

But

Le but de ce TP est de rappeler quelques fondamentaux du programme de première année.

Contenu	**commentaire**
lecture et écriture de données depuis ou vers un fichier	On encourage l'utilisation de fichiers en tant que support de données.
Principe de la représentation des nombres entiers en mémoire.	entiers "illimités" de python, séquence d'octets (hors programme).
Instructions itératives : boucles conditionnelles	Les sorties de boucle (instruction break) peuvent être présentées et se justifient uniquement lorsqu'elles contribuent à simplifier notablement la programmation sans réelle perte de lisibilité des conditions d'arrêt

Mise en œuvre

lecture de fichier

On se propose dans cette partie quelques manipulation simple sur les octets d'un fichier quelconque.

la méthode recommandée pour manipuler les octets d'un fichier est illustrée dans la fonction suivante :

```
def visualise_fichier(nom_du_fichier,n):
    """
        lit un fichier par bloc de n caractères et affiche les octets
corespondants.

        Entrées
        -------
        nom_du_fichier : str
            nom complet du chemin.

        n : int
          taille des blocs lus. le dernier peut être plus court si
          la taille du fichier n'est pas multiple de n.

        Sortie
        ------

        Aucune
    """
    with open(nom_du_fichier,"rb") as f: # "rb" pour ouverture en lecture
(r=read) , b pour binaire
        octets = f.read(n) # lit au moins n bits
        while len(octets) > 0 :
            print(octets)
            octets = f.read(n)
```

- Q1 : Tester cette fonction en l'appliquer au fichier `TD1.py` que vous être en train d'écrire par exemple, pour diverses valeurs de `n`. Si votre fichier n'est pas situé dans votre répertoire de travail, son nom complet peut être donné sous forme de chaîne de caractère brute ave le préfixe `r`(raw, brut en anglais), qui évite une mauvaise interprétation de caractères comme `\n` . Exemple : `r'C:\Users\norbert\informatique\TD1.py'`.
- Q 2: Comment sont représentés les octets ?
- Q3 : En s'inspirant du code de la fonction précédente, écrire une fonction `longueur_fichier(nom_du_fichier)` .
- QF (s'il reste du temps): Pour éviter d'écrire 2 fois `octets = f.read(n)`, on peut utiliser l'instruction `break` qui sort de la boucle en cours. Ecrire une seconde version de la fonction `visualise_fichier`.

Représentation binaire naturelle

La méthode `read` renvoie un objet de types `bytes`, séquences non mutables (non modifiables) d'octets. ces séquences peuvent être converties en entiers grâce à la représentation binaire naturelle.

- Q4 : Quel est l'entier (écrit en base en base 10 ...) associé à l'octet de représentation binaire $(00111100)_2$:
- Q5 : Quel est l'entier associé à l'octet de représentation binaire $(0011110000111100)_2$:

Les fonctions `int.to_bytes` et `int.from_bytes` permettent de convertir respectivement un entier en séquence d'octets et une séquence d'octets en entier, en représentation naturelle, si l'on spécifie le paramètre `byteorder='big'`.

- Q6 : Lire la documentation de ces deux fonctions :
  - directement dans `Spyder` en tapant leur noms dans la console puis `Ctrl` + `I` .
  - en ligne et en francais (https://docs.python.org/fr/3/library/stdtypes.html#additional-methods-on-integer-types) moins écologique.
- Q7 : convertir en séquence d' octets : 60, 97, 10, 600, puis 6000.
- Q8 : Convertir en entier `b'\x32'`, `b'LyceeWallon'` .
- Q9 : Ecrire une fonction `octets(entier,n)` qui convertit un entier `entier` en un séquence de `n` octets. on supposera que n est assez grand.
- Q10 : Ecrire une fonction `entier(octets)` qui convertit une séquence d'octets en entier naturel.

RSA

le cryptage RSA s'appuie sur la fourniture de trois grand entiers naturels`n,e` et `d`, ayant des propriétés mathématiques bien spécifiques, tels que :

> ***Chiffrement du message***
> *Si `M` est un entier naturel strictement inférieur à `n` représentant un message, alors le message chiffré sera représenté par :*
>
> $$C \equiv M^e \pmod{n}$$
>
> *l'entier naturel `C` étant choisi comme l'unique entier positif ou nul strictement inférieur à `n`.*
>
> ***Déchiffrement du message***
> *Pour déchiffrer `C`, on utilise d, et l'on retrouve le message clair M par :*
>
> $$M \equiv C^d \pmod{n}$$
>
> *La paire `(n,e)` représente la clé publique (`e` pour encodage), tout le monde peut la connaitre. La paire `(n,d)` constitue la clé privée, (`d` pour décodage) connue seulement par le destinataire.*

On admettra que la transformation est bijective de $[0, n[$ dans $[0, n[$. Cet encodage/décodage est possible directement en python avec la fonction `pow` utilisée avec 3 arguments:

- `C = pow(M,e,n)` encode le message `M`.
- `M = pow(C,d,n)` décode le message `C`.

- Q11 : Prévoir et évaluer le résultat de `pow(9,2)`, puis `pow(9,2,10)`.

Génération de clé

- Q12 : Taper les lignes suivantes pour générer des clefs aléatoires:

```
from Crypto.PublicKey import RSA
clef = RSA.generate(1024) # clefs 1024 bits
n, e, d = clef.n, clef.e, clef.d
```

- Q13 : encoder puis décoder le "message" 42 avec vos clefs.

Cryptage de fichiers

Compte tenu de ce qui précède, on peut encoder ou décoder n'importe quel fichier.

On note k l'unique entier tel que $2^{8k} < n < 2^{8(k+1)}$ (n est impair positif).

- Q14 : Ecrire une fonction `k(n)` qui calcule le nombre k tel qu'il faille au moins `k+1` octets pour stocker l'entier naturel `n`.

L'algorithme d'encodage est le suivant:

```
   - lire le fichier par bloc de k octets, et pour chaque bloc:
           - convertir le bloc en entier M
           - encoder l'entier M en un entier C
           - convertir C en une séquence de k+1 octets.
           - sauver la sequence dans un fichier de sortie
   -sauver la taille du dernier bloc lu sur 4 octets
```

L'ouverture et l'écriture simultanée de deux fichiers en mode binaire se fait par

```
with open('nom_du_fichier_a_crypter','rb') as fe,
open('nom_du_fichier_crypte','wb') as fs :
```

et la lecture/écriture par

```
fe.read(k)
fs.write(octets)
```

- Q14 : Ecrire une fonction `encode(fichier,n,e)` qui encode un fichier avec la clef publique `(n,e)`. On ajoutera le suffixe `cry` au nom du fichier crypté créé : ainsi `mes_secrets.odt sera crypté en mes_secrets.docx.odt`.
- Q15 : Ecrire une fonction `decode(fichier,n,d)` qui décode un fichier avec la clef privée `(n,d)`. On supprimera le suffixe `cry` du nom du fichier crypté : ainsi `mes_secrets.odt.cry` sera décodé en `mes_secrets.odt`.