UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Licence Informatique
Info0603 – 2021/2022

Travaux dirigés et TP n° 7

Chiffrement asymétrique

Exercice 1 (Recherche de Clés RSA par Miller-Rabin)

1. Construire un graphique permettant d'estimer la densité des nombres premiers. Combien, en moyenne, faut-il tester de nombres à n bits, pris au hasard, pour trouver un nombre premier ?
2. Evaluer (en TD) puis mesurer la vitesse d'exécution du test de primalité d'un nombre premier selon son nombre de bits en reprenant l'exemple suivant :

```
import timeit
timeit.timeit(lambda: sum( [k for k in range(100000)]), number=100)
```

3. Construire la fonction estPremierOuPseudoPremierDansLaBase(n,a) qui renvoie le test de l'égalité $a^{n-1} \equiv 1[n]$
   Pour cela on pourra utiliser les égalités et les notations suivantes :
   $a^{n-1} \equiv 1[n] \iff a^{n-1} - 1 \equiv 0[n]$
   En notant $n - 1 = k.2^r$
   Avec n=401 on a $k = 25$ et $r = 3$ et :
   $a^{401-1} \equiv 1[n] \iff a^{400} - 1 \equiv 0[n] \iff (a^{200} + 1).(a^{200} - 1) \equiv 0[n]$
   $\iff (a^{200} + 1).(a^{100} + 1).(a^{50} + 1).(a^{25} + 1).(a^{25} - 1) \equiv 0[n]$
   Ce qui rends l'algorithme transparent (il est implémenté dans la cours de A.Ninet p30 et p31).

```
def estPremierOuPseudoPremierDansLaBase(n,a):
    """ Teste la pseudo-primalité d'un entier n en base a
    c.a.d si a**(n-1)=1[n]
    >>> estPremierOuPseudoPremierDansLaBase(121,3)
    True
    >>> estPremierOuPseudoPremierDansLaBase(121,2)
    False
```

4. Construire une fonction renvoyant une liste des nombres de Poulet (pseudo-premiers en base 2).

   Le débat fait encore rage pour savoir si cette dénomination vient du nom de Paul Poulet (1887-1946) ou tout simplement du nombre de cuisses de la volaille.

```
def lNombresDePoulet(nbits=16):
    """ Renvoie la liste des nombres de Poulet inférieurs à 2**nbits
    cad pseudopremiers en base 2 Voir suite A001567 de l'OEIS
    >>> >>> lNombresDePoulet(10)
    [341, 561, 645]
```

5. Construire une fonction renvoyant pour un entier n donné, les bases a sur lesquelles il faut tester ce nombre pour être certain qu'il est premier.

```
def lbasesDeTestsDePrimalite(nbits=32,verbose=True):
    """Renvoie la liste des bases à testés selon la valeur du nombre premier à tester :
    >>> lbasesDeTestsDePrimalite(10)
    [(341, [2, 3]), (1105, [2, 3, 5])]
    Donc pour tester la primalité des nombres inférieurs à 341
    il suffit de tester avec la base 2
    pour tester ceux inférieurs à 1105 il suffit de tester les bases 2 et 3.
```

6. En déduire un algorithme de recherche de primalité.
7. Proposer une optimisation supplémentaire.
   Solution : *Voir quelles combinaisons seraient optimum pour les temps de calcul ou minimiser le nombre de cas différents https ://en.wikipedia.org/wiki/Miller*

8. Une autre optimisation : Si $a^{2m} \equiv 1[n]$ avec $a^m \neq 1[n]$ et $a^m \neq -1[n]$ " alors n n'est pas premier. Expliquer pourquoi
   Solution : *En effet, dans ce cas, $x = a^m$ est une solution de l'équation $x^2 \equiv 1[n]$ et donc $(x-1).(x+1) \equiv 0[n]$*
9. Grâce à tout cela, proposer un algorithme efficace de recherche des nombres premiers.
10. Implanter cette algorithme (on pourra aussi voir :

    `https://qastack.fr/codegolf/123645/miller-rabin-strong-pseudoprimes`
    `ou même https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#cite_note-13`

11. Construire une fonction renvoyant des clés RSA

Exercice 2 (Implémentation RSA)
Important : installer la nouvelle version de Binaire603 présente dans les documents généraux pour profiter notamment de sa méthode ajouteMot et lis mot qui permet d'ajouter directement un entier de plus d'un octet (sont alors écris son nombre d'octets et la valeur de ces octets).

Programmer les méthodes suivantes de CodeurRSA32Bits :

— Le générateur de clés

```
def generateurDeCle(p=0,q=0,d=0,verbose=False):
       """p et q doivent être premiers et e premier avec (p-1)*(q-1)
       Si ce n'est pas le cas on prends à la place le nombre premier suivant
       En cas de paramètre nul on prends des nombres au hasard
       >>> CodeurRSA32Bits.generateurDeCle(p=0x12345,q=0x43215,d=0xabcba)
       (703679, 7959681319, 20503911691)
```

— Le codeur et le décodeur

```
def binCode(self,monBinD:Binaire603,verbose=False)->Binaire603:
   >>> CodeurRSA32Bits(703679, 7959681319, 20503911691).
                                                binCode(Binaire603([ 0x02, 0x03, 0x04, 0x05]))
   Binaire603([ 0x01, 0x04, 0x02, 0x5f, 0x6c, 0x4d, 0xba])
```

A noter que le début du Binaire603 est la longueur du Binaire603 d'origine (ici 1 et 4 pour le nombre d'octeurs et la valeurs du nombre d'élément qui est 4).

```
def binDecode(self,monBinC:Binaire603,verbose=False)->Binaire603:
   >>> CodeurRSA32Bits(703679, 7959681319, 20503911691).
binDecode(Binaire603([ 0x01, 0x04, 0x02, 0x5f, 0x6c, 0x4d, 0xba]))
   Binaire603([ 0x02, 0x03, 0x04, 0x05])
```

Exercice 3 (PGP)
PGP (Pretty Good Privacy) a été le premier logiciel de chiffrement utilisable par le grand public. Il est fondé sur la technique de la clé asymétrique. Il a valu à son créateur, Philip Zimmermann, une enquête criminelle de trois ans de la part des Douanes américaines, au prétexte d'avoir violé les restrictions sur l'exportation de logiciels de cryptographie en diffusant PGP dans le monde entier (PGP avait été publié en 1991 sur le web comme logiciel libre). (Source wikipedia voir aussi : [**?**]).

PGP offre des services d'authentification, de confidentialité, de compression et de segmentation, tout en étant compatible avec de nombreux systèmes de messagerie électronique :

— authentification : l'expéditeur crée un condensat de son message (avec par exemple SHA-1), chiffre ce condensat avec sa clé privée et l'ajoute en début de message. Le destinataire déchiffre l'ajout en début de message avec la clé publique de l'émetteur et en extrait le condensat. Il calcule ensuite lui-même un condensat du message en utilisant la même fonction de condensat et le compare à celui qu'il a déchiffré ; même résultat :> expéditeur authentifié et message intègre. Le couple clé publique/clé privée peut être fourni par RSA ou DSA ;

— confidentialité (chiffrer des messages à transmettre ou des fichiers à enregistrer) : génération d'une clé secrète de taille 128 bits par exemple (nommée clé de session, valable pour un seul fichier ou un seul message). Le message ou le fichier est chiffré au moyen de cette clé de session avec un algorithme de cryptographie symétrique. Puis cette clé secrète est chiffrée au moyen de la clé publique RSA ou ElGamal du destinataire et ajoutée au début du message ou du fichier. Le destinataire du message déchiffre l'en-tête du message avec sa clé privée RSA ou ElGamal et en extrait la clé secrète qui lui permet de déchiffrer le message. Pour que la sécurité de l'échange soit plus sûre il ne faudrait pas utiliser le chiffrement sans authentification. PGP générant des clés très souvent (à chaque fichier ou message), le générateur aléatoire associé à PGP doit être particulièrement efficace afin de ne pas générer des séquences de clés prévisibles ;

1. Reprendre les points de cours en rapport avec le texte précédent.
2. Écrire les commandes Python utilisant, notre CodeurRSA, permettant de reproduire le fonctionnement de PGP tels qu'indiqués dans sa documentation :
   OpenPGP fonctionne avec un cadenas, et une clé : - votre cadenas est public - la clé qui ouvre votre cadenas est secrète : vous êtes le seul à détenir cette clé.
   2.1 Cryptage d'un message : on ferme le "cadenas" (clé PGP du destinataire)

Lorsque vous envoyez un message crypté, vous fermez le cadenas : vous cliquez sur l'icône OpenPGP du logiciel e-mail et le message va être automatiquement crypté avec le cadenas du destinataire.

2.2 Déchiffrement du message : le destinataire ouvre le cadenas avec sa clé secrète (privée)

Le destinataire déchiffre automatiquement le message crypté car il possède la clé du cadenas (sa clé secrète).

4. Mise en place des clés PGP

Avant d'utiliser OpenPGP, il est nécessaire de se créer sa propre paire de clés et de se procurer la clé publique de ses correspondants.

4.1 Générer votre paire de clés

Cette paire de clés sera unique normalement, et vous pouvez la conserver durant des années. Donc, entrainez-vous avant de diffuser la clé publique issue de cette paire de clés.

GPG vous proposent de générer votre paire de clés lors du premier lancement.

Cette paire de clés contient une clé publique (le cadenas) et une clé privée (la clé ouvrant le cadenas)

4.2 Exporter votre clé publique et envoyer une copie de cette clé publique à vos correspondants

Cette clé publique est le "cadenas" qui permettra à vos correspondants de crypter les e-mails qu'ils vous envoient.

GPG ou PGP© permettent l'exportation de votre clé publique par leur fonction "export".

Ces correspondants doivent avoir une copie de votre clé publique PGP, qui ressemblera à ceci (en plus long) :

—BEGIN PGP PUBLIC KEY BLOCK—

Version : GnuPG v1.0.6 (GNU/Linux)

mQGiBDm+dJYRBACyoHzCRdJXXXFai0bENERmPYFQwx9gOWm7kZRnD27tzLjuQVWtoFgooN/li04QIAn0o6fXolGIbPH//
x4QstrZDVqxC8iEwEghHkjfJJM8GBECAAwFAjm+dL0FCQPCZwAACgkQvatgyKeVS0gbuwCePu5P6uEzIeOKtXGVOoCZ
B1C8yPkAoJFot6R8KbweB58KBR4fCihwKhKa=fytL

—END PGP PUBLIC KEY BLOCK—

4.3 Importer la clé publique de ses correspondants pour la stocker dans votre "trousseau"

GPG ou PGP© permettent l'importation de la clé de vos correspondants dans votre trousseau de clés publiques par la fonction "import".

Ensuite, lorsque vous enverrez un e-mail à un de ces correspondants, le plug-in courrier se chargera de trouver le "cadenas" de ce correspondant (sa clé publique) dans votre trousseau de clés publiques PGP, puis il cryptera automatiquement le message avant envoi.

Exercice 4 (Implémentation de Signature)

Déposer sur le forum votre clé publique en ajoutant votre prénom et votre nom signés grâce à votre clé privée.

Exercice 5 (Implémentation de Messagerie)

Aller sur le forum, déposer votre clé publique, et ajouter quelques messages.