Récapitulatif des travaux pratiques de traitement de l'information

Loïc BERTRAND - S4A2

Tableau récapitulatif de toutes les réalisations

Réalisation	Pourcentage d'aboutissement
RSA	100 %
El Gamal	90 %

Utilisation des outils

Sha1sum

La commande sha1sum permet de calculer l'intégrité d'une donnée. Cela est par exemple utile dans le cadre d'un transfert de données. Dans ce cas on compare l'empreinte du fichier reçu avec l'empreinte du fichier avant l'envoi. Ci-dessous figure un exemple d'utilisation de cette commande.

```
#!/bin/sh

comparaison_des_empreintes(){
  echo "NOTICE: comparaison des empreintes"
  local empreinte1=$1
  local empreinte2=$2
  echo "NOTICE: empreinte1: $empreinte1"
  echo "NOTICE: empreinte2: $empreinte2"
  [ "$empreinte1" = "$empreinte2" ] \
    && echo "RESULTAT: empreintes identiques !" && return 0 \
    || echo "RESULTAT: empreintes différentes !" || return 1
}

FICHIER1="./fichier1"
FICHIER2="./fichier2"

printf "\n\n"

echo "**** Simulation d'un transfert de fichier avec succès****"
echo "NOTICE: génération du fichier 1"
```

```
touch $FICHIER1
echo "NOTICE: génération de l'empreinte du fichier1"
empreinte_fichier_1=$(sha1sum $FICHIER1 | cut --delimiter=' ' --
field=1)
echo "NOTICE: simulation du transfert du fichier 1 vers le fichier 2."
cp $FICHIER1 $FICHIER2
echo "NOTICE: génération de l'empreinte du fichier2"
empreinte_fichier_2=$(sha1sum $FICHIER2 | cut --delimiter=' ' --
field=1)
comparaison_des_empreintes $empreinte_fichier_1 $empreinte_fichier_2

printf "\n\n"

echo "**** Simulation d'un transfert de fichier avec échec****"
echo "NOTICE: génération du fichier 1"
touch $FICHIER1
echo "NOTICE: génération de l'empreinte du fichier1"
empreinte_fichier_1=$(sha1sum $FICHIER1 | cut --delimiter=' ' --
field=1)
echo "NOTICE: simulation du transfert du fichier 1 vers le fichier 2."
cp $FICHIER1 $FICHIER2
echo "Bonjour Monde" > $FICHIER2
echo "NOTICE: génération de l'empreinte du fichier2"
empreinte_fichier_2=$(sha1sum $FICHIER2 | cut --delimiter=' ' --
field=1)
comparaison_des_empreintes $empreinte_fichier_1 $empreinte_fichier_2
```

PyDes

Le chiffrement DES (=Data Encryption Standart) est un un algorithme de chiffrement symétrique. Le module python pyDes permet de chiffrer et déchiffrer des messages comme le témoigne l'exemple ci-dessous.

```
import pyDes

# clé consituée de 8 octets
cle = pyDes.des("DESCRYPT", pyDes.CBC, "\0\0\0\0\0\0\0\0")
# Chiffrement du message et complétion des caractères manquants par un
espace pour atteindre un nombre de caractères multiple de la taille de
la clé
donnees = cle.encrypt("Bonjour Monde !, ", ' ')
# Déchiffrement du message
print(cle.decrypt(donnees, ' '))
```

RSA

Le chiffrement RSA, du nom de ses trois inventeurs Ronald Rivest, Adi Shamir et Leonard Adleman, est un algorithme de chiffrement asymétrique. Le module python Crypto permet de générer une paire de clé. L'une est privée, l'autre publique. Deux tiers peuvent ainsi échanger leurs clés publiques afin de communiquer de manière sécurisée. On peut utiliser ce module comme cela est décrit ci-dessous.

```
```

```
# pip install pycryptodome pour obtenir les modules ci-dessous
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

# Génération d'une paire de clé publique / clé privée de 512 bits
paire_cles = RSA.generate(1024)
cle_publique_rsa =
PKCS1_OAEP.new(RSA.importKey(paire_cles.publickey().exportKey()))
cle_privee_rsa =
PKCS1_OAEP.new(RSA.importKey(paire_cles.export_key()))

# Chiffrement du message avec la clé publique
message = "Hello Monde !"
print("Message: " + message)
message_chiffre = cle_publique_rsa.encrypt(str.encode(message))
print("Message chiffré: " + str(message_chiffre))

# Déchiffrement du message avec la clé privée
message_dechiffre = cle_privee_rsa.decrypt(message_chiffre)
print("Message déchiffré: " + str(message_dechiffre))
```

Gnupg

GPG de l'acronyme GNU Privacy Guard permet le chiffrement symétrique ou asymétrique ainsi que la signature de données. Dans l'exemple ci-dessous, on voit comment générer des clés.

```
#!/bin/sh

mkdir -p ./gpg
export GNUPGHOME="$(pwd)/gpg"
cat > fichier_temporaire << FIN
     %echo "Génération d'une clé GPG"
     Key-Type: DSA
     Key-Length: 1024
     Subkey-Type: ELG-E
     Subkey-Length: 1024
     Name-Real: Prénom Nom
     Name-Comment: commentaire
     Name-Email: exemple@exemple.fr
     Expire-Date: 0
     Passphrase: mot_de_passe
     %commit
     %echo "REUSSI"
FIN
gpg --batch --generate-key fichier_temporaire
gpg --list-secret-keys
rm fichier_temporaire
```

Openssl

Openssl est un outil de chiffrement qui permet entre autres de générer des certificats SSL qui sont par exemple utiles pour sécuriser les échanges via le protocole https. Je l'utilise par exemple pour générer des certificats auto-signés pour mon reverse-proxy Nginx lorsque le robot de certification Let's Encrypt échoue dans la génération automatique de certificats SSL valides. Ci-dessous, voici comment générer un couple certificat / clé privé.

```
openssl req -x509 -nodes -days 3 -newkey rsa:4096 -keyout
./cle_privee.pem -out ./certificat.pem -subj
"/C=FR/ST=State/L=Location/O=Organization/OU=Unit/CN=Name"
```

Explication des algorithmes

RSA

- Dans un premier, je génère une paire clé privée / clé publique.

```
def generer_cle():
    """
    :return: un tuple contenant deux tuples qui sont respectivement la
clé privée et la clé publique
    :rtype: tuple
    """
    # Génération aléatoire de deux nombres premiers p1 et p2
    p1 = numpy.random.choice(1000, 1)
    p2 = numpy.random.choice(1000, 1)
    while est_premier(p1) is False:
        p1 = numpy.random.choice(1000, 1)
    while est_premier(p2) is False:
        p2 = numpy.random.choice(1000, 1)
    # Calcul de n
    n = p1 * p2
    # Calcul de m
    m = (p1 - 1) * (p2 - 1)
    # Recherche de c tel que pgcd(m,c)=1 ) et de d = pgcde(m,c) tel
que 2 < d < m
    r = 10
    d = 0
    c = 0
    while r != 1 or d <= 2 or d >= m:  # Tant que r≠1, on réitère
        c = numpy.random.choice(1000, 1)  # On tire c aléatoirement
        r, d, v = trouver_pgcde_avec_bezout(c, m)
    n, c, d = int(n), int(c), int(d)
    return (n, c), (n, d)
```

- Ensuite, je chiffre un message grâce à la clé privée.

```
def chiffrer_message(cle_privee, message):
    """
    :param cle_privee
    :type cle_privee: tuple
    :param message à chiffrer
```

```
    :type message: str
    :return message chiffré
    :rtype: list
    """
    (n, c) = cle_privee
    # Conversion du message en ASCII
    message_ascii = [str(ord(caractere)) for caractere in message]
    # Vérification que chaque code ASCII ait bien une longueur de 3
digits en complétant par des 0
    for index, element in enumerate(message_ascii):
        if len(element) < 3:
            while len(element) < 3:
                element = '0' + element
            message_ascii[index] = element
    # Regroupement du message ascii
    message_ascii = ''.join(message_ascii)
    # Définition de la taille des groupes du message
    debut, fin = 0, 4
    # Ajout éventuel de zéros au message afin qu'il soit un multiple
de la taille d'un groupe, içi 4
    while len(message_ascii) % fin != 0:
        message_ascii += '0'
    # Groupement
    groupes = []
    while fin <= len(message_ascii):
        groupes.append(message_ascii[debut:fin])
        debut = fin
        fin += 4
    # Chiffrement et retour des groupes
    return [str(((int(groupe)) ** c) % n) for groupe in groupes]
```

- Je déchiffre alors le résultat grâce à la clé publique précédemment générée.

```
def dechiffrer_message(cle_publique, groupes):
    """
    :param cle_publique
    :type cle_publique: tuple
    :param groupes: groupes à déchiffrer
    :type groupes: list
    :return message déchiffré
    :rtype str
    """
    (n, d) = cle_publique
    # Déchiffrage des groupes
    groupes_dechiffres = [str((int(groupe) ** d) % n) for groupe in
groupes]
    # Formation de groupes de 4
    for index, element in enumerate(groupes_dechiffres):
        if len(element) < 4:
            while len(element) < 4:
                element = '0' + element
            groupes_dechiffres[index] = element
    # Groupement
    groupes_dechiffres = ''.join(groupes_dechiffres)
```

```
    # Conversion en ascii
    message_ascii = ""
    debut = 0
    fin = 3
    while fin < len(groupes_dechiffres):
        message_ascii += chr(int(groupes_dechiffres[debut:fin]))
        debut = fin
        fin += 3
    return message_ascii
```

El Gamal

- Dans un premier temps, je génère un nombre premier aléatoire.

```
def est_premier(entier):
    """
    :param entier: entier à tester
    :type entier: int
    :return: vrai si l'entier passé en paramètre est premier, faux
sinon
    :rtype: bool
    """
    if entier == 1 or entier == 2:  # On traite déjà 1 et 2 qui sont
premiers
        return True
    if entier % 2 == 0:  # Si le nombre est divisible par 2 ici, il
n'est pas entier
        return False
    if entier ** 0.5 == int(entier ** 0.5):  # Si la racine carée de
l'entier n'est pas entière, alors il n'est pas premier
        return False
    for diviseur in range(3, int(entier ** 0.5), 2):  # On teste tous
les diviseurs entiers jusqu'à la racine carée de l'entier
        if entier % diviseur == 0:
            return False
    return True


def generer_nombre_premier_aleatoire():
    """
    :return: nombre premier aléatoire
    :rtype: int
    """
    p = numpy.random.choice(1000, 1)
    while est_premier(p) is False:
        p = numpy.random.choice(1000, 1)
    return int(p[0])
```

- Ensuite, je trouve sa racine primitive. Ici, j'ai eu un peu de mal à écrire l'algorithme de l'exponentiation rapide (comme en témoignent mes brouillons fournis avec le code).

```
def trouver_racine_primitive(nombre_premier):
    """
```

```
    :param nombre_premier
    :type nombre_premier: int
    :return: racine primitive du nombre premier passé en paramètre
    :rtype: int
    """
    for i in range(1, nombre_premier):
        racines = []
        for k in range(1, nombre_premier):
            racines.append(i ** k % nombre_premier)
        if set(range(1, nombre_premier)).issubset(racines):
            return i
    return -1
```

- Je calcule ensuite l'indicatrice d'Euler.

```
def selectionner_a_dans_0_a_p_moins_2(p):
    """
    :param p: nombre premier p
    :return: choix d'un entier dans {0,...,p-2}
    :rtype: int
    """
    return random.choice(range(0, p - 1))
```

- Je peux alors générer le couple clé privée / clé publique.

```
def generer_cles(p, g, a):
    """
    :param p: nombre premier
    :type p: int
    :param g: racine primitive
    :type g: int
    :param a: indicatrice d'Euler
    :type a: int
    :return: tuple clé publique, clé privée
    :rtype: tuple
    """
    A = g ** a % p
    cle_publique = (p, g, A)
    cle_privee = a
    return cle_publique, cle_privee
```

- Je calcule une deuxième indicatrice d'Euler et je chiffre un message grâce cette dernière et la clé publique précédemment générée.

```
def chiffrer_message(message, cle_publique, b):
    """
    :param message: message à chiffrer
    :type message: int
    :param cle_publique:
    :type cle_publique tuple
    :param b: indicatrice d'Euler
    :type b: int
    :return: message chiffré
    :rtype: int
```

```
    """
    (p, g, A) = cle_publique
    B = g ** b % p
    c = message * A ** b % p
    return B, c
```

- Enfin, je peux déchiffrer le message grâce aux clés.

```
def dechiffrer_message(cryptogramme, cle_publique, cle_privee):
    """
    :param cryptogramme: message chiffré
    :type cryptogramme: int
    :param cle_publique:
    :type cle_publique: tuple
    :param cle_privee:
    :type cle_privee: tuple
    :return: message déchiffré
    :rtype: int
    """
    (p, g, A), a = cle_publique, cle_privee
    (B, c) = cryptogramme
    return B ** (p - 1 - a) * c % p
```