Université de Nantes

BUT Info 2

Cryptographie

2022-2023

Exercice 1 Fonctions de base

1.1. Ecrire une fonction permettant de tester si un nombre est premier (on pourra éventuellement utiliser le fait que s'il ne l'est pas son plus petit diviseur premier est inférieur ou égal à $\sqrt{n}$

1.2. Ecrire une fonction permettant de renvoyer la liste des nombres premiers inférieurs à un nombre passé en argument. Calculer la proportion de nombre premiers inférieurs à n. Comparer à $\frac{1}{(ln(n)-1)}$

1.3. Ecrire une fonction réalisant l'algorithme d'Euclide étendu en prenant en arguments deux entiers naturels a et b et en renvoyant le triplet formé du pgcd d de ces deux entiers et de deux entiers u et v vérifiant $au + bv = d$.

1.4.

- **a.** Ecrire une fonction permettant de tester si les nombres 13, 1009, 10007,... 100000000000031 sont premiers (nombres rappelés dans une liste dans le fichier `tp1_crypto.py`). Consigner dans une liste le temps mis pour obtenir la réponse.
- **b.** Représenter avec matplotlib le nuage de points montrant la relation entre le nombre de chiffres apparaissant dans l'écriture décimale de chacun de ces nombres et le temps mis pour renvoyer le résultat.
- **c.** Remarquer la nature de ce lien si on représente en ordonnées le logarithme du temps de calcul plutôt que le temps de calcul lui-même. Effectuer une régression linéaire avec la bibliothèque scipy.stats entre les deux variables représentées.

Exercice 2 Les nombres de Mersenne

On utilisera la relation $x^n-1 = (x-1)\times(x^{n-1}+x^{n-2}+...+x+1)$ et $1+x+x^2+...+x^n = \frac{1-x^{n+1}}{1-x}$ (avec $x \neq 1$).

2.1. Montrer que si $a^n - 1$ est premier alors $a = 2$.

2.2. Montrer que $2^n - 1$ ne peut être premier que si n est premier

2.3. Vérifier que $2^{11} - 1$ n'est pas premier

Exercice 3 chiffrement affine

Un dictionnaire est fourni pour permettre d'associer à chaque lettre une valeur correspondant à sa position dans l'alphabet en plus de certains caractères spéciaux (comme on va travailler dans $\mathbb{Z}_{/31\mathbb{Z}}$ la 31ème position attribué au caractère ' :' est associée à la valeur 0.

3.1. Mettre en place un dictionnaire permettant d'associer à chaque valeur dans $\mathbb{Z}_{/31\mathbb{Z}}$ le caractère qu'elle représente.

3.2. Réaliser une fonction `chiff_affine(mess,a,b)` permettant le chiffrement d'un message par transformation de chaque caractère à partir du calcul dans $\mathbb{Z}_{/31\mathbb{Z}}$ de $a * position(caractere) + b$. La fonction renverra le message chiffré sous forme de texte

3.3. Réaliser une fonction `dechiff_affine(mess,a,b)` permettant de déchiffrer un message reçu

3.4. Si on ignore la clé utilisée, combien de clés va-t-on devoir tester si on souhaite casser ce système en explorant exhaustivement toutes les solutions ? Réaliser une

fonction `cass_chiff_affine(mess)` permettant de tester ceci avec un message en clair en argument. La fonction générera une clé obtenue aléatoirement puis affichera tous les messages déchiffrés avec toutes les clés testées et renverra le message chiffré et la clé ayant permis de retrouver le message en clair donné en argument.

Exercice 4 Problème du logarithme discret

4.1. Ecrire une fonction, renvoyant pour un nombre premier p passé en argument un générateur de $(Z_{/p\mathbb{Z}})^*$ choisi aléatoirement (choisir un nombre aléatoire et tester s'il est générateur et reproduire l'opération jusqu'à ce que ce soit le cas).

4.2. Ecrire une fonction, renvoyant le log discret lorsqu'il existe pour un élément de $\mathbb{Z}_{/p\mathbb{Z}}$ et un générateur g de $(Z_{/p\mathbb{Z}})^*$.

4.3. Ecrire une fonction, avec l'algorithme de Shanks des pas de bébé, pas de géant :

- Soit $s = 1 + \lfloor\sqrt{p}\rfloor$
- Calculer g^{-s}
- Créer deux listes :
  - $L_1 : 1,\ g,\ g^2,\ g^3, \cdots, g^{s-1}$
  - $L_2 : y,\ y.g^{-s},,\ y.g^{-2s},,\ y.g^{-3s}, \cdots ,,\ y.g^{-(s-1)\times s}$
- Trouver une occurrence commune g_0^r et $y.g^{-k_0 s}$ respectivement dans les listes L_1 et L_2
- $x = r_0 + k_0.s$ est une solution

4.4. Justifier que cet algorithme permet bien de renvoyer le logarithme discret de y (s'il existe). Pour cela en posant x_0 (avec $1 \leq x_0 \leq p-1$ ce logarithme discret tel que $g^{x_0} = y$, considérer la division euclidienne de x_0 par s : $x_0 = k_0 \times s + r_0$.

4.5. Comparer les temps d'exécution moyens, des deux fonctions réalisées pour l'obtention du log discret, sur un échantillon de 30 essais avec $p = 4999$ et des générateurs g et des valeurs cibles y choisis « aléatoirement » .

Exercice 5 Signature numérique d'un message non confidentiel

Bernard et Alice communiquent ensemble en utilisant le cryptosystème d'El Gamal. Bernard souhaite envoyer un message (non confidentiel) signé numériquement. Pour cela il va envoyer à Alice le message m (que vous imaginerez) ainsi que le chiffrement d'une empreinte de celui-ci générée par une fonction de hachage soit $e(h(m))$. Pour cela il dispose de la clé publique d'Alice (p, g, A)

5.1. Vous générerez cette clé avec une fonction `key_creation()` renvoyant un tuple `(p,g,A,a)` :

- en exploitant une fonction `list_prime(b,n)` renvoyant `n` nombres premiers supérieurs à `b` et en effectuant un choix aléatoire d'un nombre premier p supérieur à 1000 dans une liste renvoyée par cette fonction
- en choisissant aléatoirement un entier entre 2 et $p-2$, ce sera la clé secrète a d'Alice
- en cherchant un générateur g de $(\mathbb{Z}_{/p\mathbb{Z}})^*$

5.2. Pour l'envoi du message de Bernard vous allez :

a. Calculer l'empreinte du message de Bernard en utilisant la bibliothèque `hashlib` :

```python
import hashlib as hash
message="Je déclare être bien l'auteur de ce texte par lequel\
je te déclare ma flamme. Bien romantiquement. Bernard"
#Mais Bernard c'est un peu confidentiel ça quand même
message_utf8=message.encode()
'''utilisation de la fonction de hashage SHA256 et encode
en hexadécimal :'''
message_hash=hash.sha256(message).hexdigest()
#conversion en entier base 10 :
number_message_hash=int(message_hash,16)
```

b. Définir une clé privée b pour Bernard en choisissant aléatoirement un entier entre 2 et $p-2$ puis procéder au chiffrement de l'empreinte du message (en ayant pris soin de découper le message par paquet de chiffres inférieurs à 1000 (pour s'assurer de retrouver le message d'origine en travaillant modulo p). Quels sont les éléments que Bernard doit envoyer à Alice ?

c. En considérant qu'un élément tiers de confiance connaît la clé privée b de Bernard, comment Alice peut-elle se faire assurer que Bernard est bien l'auteur de ce message (non répudiation) ? Si cela s'avère bien être le cas on remarquera que l'opération permet aussi de s'assurer de l'intégrité du message reçu par rapport à celui envoyé. Procédez à ces vérifications.

Exercice 6 Code détecteur

On utilise dans cet exercice l'association entre des trames de bits et l'écriture d'un polynôme dans $\mathbb{Z}_{/2\mathbb{Z}}[X]$. Ainsi par exemple la liste `[1,0,0,1,1]` peut être associée au polynôme X^4+X+1.

6.1. Réaliser une fonction `deg(A)` renvoyant le degré du polynôme A et une fonction `plus(A,B)` renvoyant le polynôme résultant de l'addition de A et B.

6.2. Réaliser une fonction `div_eucl_pol(A,B)` renvoyant le quotient et le reste de la division euclidienne d'un polynôme de $\mathbb{Z}_{/2\mathbb{Z}}[X]$ par un autre.

6.3. Réaliser une fonction `envoi_message_CRC(mess,G)` renvoyant le message à transmettre par la méthode du CRC à partir d'un message contenant l'information et d'un polynôme générateur. Les messages et le polynôme générateur sont à représenter par une liste de bits.

6.4. Réaliser une fonction `test_CRC(mess,G)` permettant de tester la présence d'une erreur dans un message reçu

6.5. Vérifier l'absence d'erreur dans un message reçu à partir d'une information initiale `MI=[1,0,1,1,1,0,1,0,0,1]` et d'un polynôme générateur `G=[1,0,0,1,1]`.

6.6. Réaliser une fonction `test_echantillon(mess,G,long,n_essais)` générant sur `n_essais` un paquet d'erreur de longueur `long` à positionner dans le message (ce paquet est positionné aléatoirement, ses extrémités représentent des erreurs et les bits entre ses extrémités ont subi ou non des erreurs). La fonction renverra le nombre de fois, sur les `n_essais`, où l'erreur est détectée.

Vérifier avec l'étude d'un échantillon de 10000 transmission d'un même message qu'avec le polynôme générateur « CRC-16 » : $X^{16}+X^{15}+X^2+1$:

a. La présence d'une seule erreur semble être systématiquement détectée (on peut démontrer que la présence d'un nombre impairs d'erreurs est détectée avec ce polynôme générateur)

b. Tous les paquets d'erreur de longueur inférieure ou égale à 16 (i.e. 2 erreurs séparées par 14 bits contenant des erreurs ou non) semblent être détectés

On pourrait par ailleurs, avec un nombre d'essais encore plus grand (100 000 par exemple... ça prend un peu de temps), chercher à vérifier que la probabilité de détection d'un paquet d'erreur de longueur supérieure ou égale à 17 reste supérieure à $99,99\%$

Exercice 7 Code correcteur Hamming

7.1. Réaliser une fonction qui, à partir d'un message d'information sous forme de liste de bits, renvoie le message à envoyer intégrant des bits de parité suivant la méthode vue en cours.

7.2. Réaliser une fonction qui, à partir d'un message reçu, renvoie le fait qu'une erreur ait été détectée ou non et en partant de l'hypothèse de la présence d'une seule erreur éventuelle, renvoie le message corrigé.

Exercice 8

Les fonctions de permutations
Ces fonctions vont permettre de mélanger les bits dans le mot. Par exemple la permutation $P = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$ se comprend comme la transformation plaçant le premier bit en quatrième position, le deuxième en première... Cette opération peut être réalisée par l'exploitation d'une matrice de permutation P :

$$\begin{pmatrix} 2 \\ 4 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

Dans le fichier tpdes.py proposé, les permutations seront exposées comme une liste de valeurs l[i] interprétable comme le fait que le l[i]-ème bit se retrouve en ième position (synthétique ici mais moins compatible avec une exploitation mathématique). Ainsi avec l'exemple précédent on donnerait la liste : [2, 4, 3, 1].

Fonction d'expansion
Elle permet de transformer un bloc de 32 bits en un bloc de 48 bits. Elle s'utilise comme les fonctions de permutation mais duplique par ailleurs 16 bits (ceux aux positions 1, 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28 et 29) et les dissémine parmi les 48 bits renvoyés. Elle va être utile lors dans la mise en place de la fonction f qui exploite le bloc de droite avec une sous-clé. Voici un extrait de la fonction que l'on utilisera (présentée ici comme une permutation mais notons bien que ça n'en est pas une car elle n'est pas injective) :

$$E = \begin{pmatrix} 1 & 2 & 3 & 4 & ... & 29 & 30 & 31 & 32 \\ \{2,48\} & 3 & 4 & \{5,7\} & ... & \{42,44\} & 45 & 46 & 47 \end{pmatrix}$$

Vous retrouverez, dans le fichier, la liste de 48 valeurs donnant l'expression intégrale de cette fonction (on pourra la traduire sous forme de matrice de dimensions 48×32).

Méthode de chiffrement
Création de sous-clés La clé K donnée est présentée sur 56 bits, l'ajout de 8 bits de contrôle de parité étant prévus pour s'assurer de son intégrité après la transmission transmission. Nous utiliserons la clé K suivante :

$$K = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

- On commence par lui faire subir une première permutation $CP1$. Vous devrez pour cela commencer par créer une fonction `trans_permut_list_to_array(l,n,p)` permettant le renvoi de la matrice de permutation de taille `(n,p)` associée à la

liste `l` explicitant la transformation associée à cette permutation selon la description donnée ci-dessus (la liste fournie dans le fichier étant ici `list_CP1`. Une fois la permutation réalisée :

- On sépare la clé obtenue en deux parties de 28 bits G (`cleg`) et D (`cled`)

On va alors réaliser le processus suivant pour obtenir 16 sous-clés qu'on pourra stocker dans une matrice de dimensions `(16,48)` :

- Ecraser G et D en effectuant pour chacune un décalage de 1 bit vers la gauche (le premier bit devenant le dernier)
- La clé K_1 est le résultat de la "permutation" par $CP2$ de la concaténation de G et D (attention CP2 est plutôt une compression permutée car elle transforme une clé de 56 bits en clé de 48 bits, 8 d'entre eux étant supprimés)
- Ecraser G et D en effectuant un nouveau décalage de 1 bit vers la gauche
- La clé K_2 est le résultat de la "permutation" par $CP2$ de la concaténation de G et D
- on continue de même jusqu'à la création de K_{16}

Chiffrement du message

Paquetage : On divise le message en paquet de 64 bits en complétant éventuellement les bits manquant par des 0 au début. On chiffre ensuite séparément les différents paquets. On détaille la méthode pour un paquet noté M.

- On commence par lui faire subir une permutation initiale PI : créer pour cela la matrice de permutation de taille 64×64 à partir de `list_PI`.
- On note MG la partie de gauche du résultat (les 32 premiers bits) et MD sa partie droite.

On va ensuite effectuer 16 rondes. Chacune de ces rondes suit le même schéma à ceci près qu'à la ronde k on utilisera la clé K_k. Le processus des rondes suivant est à réaliser dans une `fonction_ronde(Left_block,Right_block,num_ronde)` :

- On applique la fonction d'expansion au bloc MD. On obtient un message $E[MD]$ non plus sur 32 mais sur 48 bits.
- On calcule $E[MD] \oplus K_k$ (lors de la ronde k)
- On découpe ensuite $E[MD] \oplus K_k$ en 8 blocs de 6 bits. Notons $B_1, \cdots, B_8$ ces 8 blocs avec $B_i = x_1x_2x_3x_4x_5x_6$. On considère l'entier n dont l'écriture en binaire est x_1x_6 (donc $0 \leq n \leq 3$). On considère l'entier m dont l'écriture en binaire est $x_2x_3x_4x_5$ (donc $0 \leq m \leq 15$). On va remplacer chaque B_i par l'écriture en binaire du nombre que l'on trouve à la ligne $n+1$ et la colonne $m+1$ de la matrice de substitution S_i (données dans le fichier `tp_des.py`). Par construction chaque B_i a alors été remplacé par un bloc de 4 bits. En regroupant les 8 blocs de 4 bits on obtient alors $S[E[MD] \oplus K_k]$ en notant S l'application transformant successivement les 8 blocs .
- On applique au résultat précédent la permutation des rondes PR (à définir via une matrice de dimensions 32×32 à partir de `list_PR`) et on obtient ainsi $PR[S[E[MD] \oplus K_k]]$.
- On remplace :

$$MD \text{ par } \underbrace{PR[S[E[MD] \oplus K_k]]}_{f(MD,K_k)} \oplus MG \text{ et } MG \text{ par } MD$$

Une fois les 16 rondes effectuées :

- On recolle la partie gauche et la partie droite obtenue pour obtenir M'
- On applique la permutation inverse de la permutation initiale à M (comment obtient-on PI^{-1} ?) pour obtenir le message chiffré : $C = PI^{-1}[M']$

Déchiffrement du message : Il suffit d'appliquer les mêmes opérations mais dans l'ordre inverse ! En effet on a après chaque ronde $MD_k = f(MD_{k-1}, K_k) \oplus MG_{k-1} = f(MG_k, K_k) \oplus MG_{k-1}$. D'où (comme comme soustraction et addition sont identiques dans $\mathbb{Z}_{/2\mathbb{Z}}$:

$$MG_{k-1} = RG_k \oplus f(MG_k, K_k)$$

Ainsi connaissant MG_k,RG_k et K_k on retrouve MG_{k-1} (et de manière évidente MD_{k-1}. En utilisant à bon escient la fonction

`fonction_ronde(Left_block,Right_block,num_ronde)` définie précédemment vous devez donc pouvoir à moindre frais déchiffrer le message chiffré obtenu.