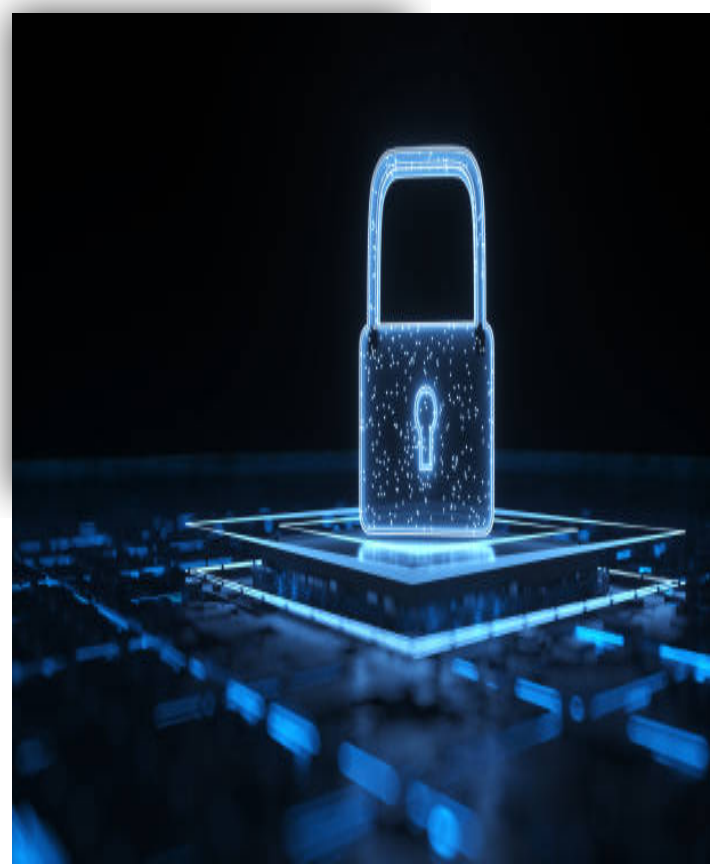
Université Sultan Moulay Slimane
Ecole Nationale des Sciences Appliquées
Khouribga

Module : cryptographie

Projet

Programmation de certains Cryptosystèmes



Réalisé par :

AROUBITE Adham
GHOUNDAL Youssef
HROUCH Aymane
KHATIB Asmaa
OUBADDOU Abdellah
RAKIB Ziyad
SOKRI Yasser
ZAHOUANI Zineb

Encadré par : Prof. Aziz IFZARNE

Année Universitaire 2021/2022

Remerciement

Avant tout développement sur cette expérience, Nous tenons à exprimer notre profonde reconnaissance à notre professeur ,M.AZIZ IFZARNE et le remercier pour nous avoir donné l'occasion de travailler sur ce projet qui nous a permet d'acquérir des nouvelles connaissances et de développer nos compétences dans le domaine.

Sommaire

Introduction :……2

I. Cryptosystemes anciennes :……..3

1. Code Vigenere……3

2. Chiffrement de Hill……..4

II. Cryptosystemes modernes :……...7

1. Masque jetable…….....7

2. Code Rabin…….....9

- Principe de fonctionnement
- Rappel sur le théorème des restes chinois
- Cryptanalyse
- Code Rabin sur python
- Méthode de Remplissage (Padding)
- Exemple illustratif
- Algorithme de signature numérique
- Vérification d'une signature

Conclusion :……15

INTRODUCTION

1. POURQUOI LA CRYPTOGRAPHIE ?

L'homme a toujours ressenti le besoin de dissimuler des informations, bien avant même l'apparition des premiers ordinateurs et de machines à calculer.

Depuis sa création, le réseau Internet a tellement évolué qu'il est devenu un outil essentiel de communication. Cependant, cette communication met de plus en plus en jeu des problèmes stratégique liés à l'activité des entreprises sur le Web. Les transactions faites à travers le réseau peuvent être interceptées, d'autant plus que les lois ont du mal à se mettre en place sur Internet, il faut donc garantir la sécurité de ces informations, c'est la cryptographie qui s'en charge.

2. Qu'est-ce que la cryptographie ?

Le mot cryptographie est un terme générique désignant l'ensemble des techniques permettant de chiffrer des messages, c'est-à-dire permettant de les rendre inintelligibles sans une action spécifique. Le verbe crypter est parfois utilisé mais on lui préfèrera le verbe chiffrer.

La cryptologie est essentiellement basée sur l'arithmétique : Il s'agit dans le cas d'un texte de transformer les lettres qui composent le message en une succession de chiffres (sous forme de bits dans le cas de l'informatique car le fonctionnement des ordinateurs est basé sur le binaire), puis ensuite de faire des calculs sur ces chiffres.

I. Cryptosystemes anciennes:

1. Code Vigénère:

Présentation de l'algorithme :

Le chiffre de Vigenère est un chiffrement symétrique utilisant une substitution poly-alphabétique pour chiffrer et déchiffrer le message secret. Ceci signifie que la clé de chiffrement est une chaîne de caractères, et c'est là-dessus que repose la sécurité de l'algorithme, car une même lettre ne sera alors pas forcément chiffrée de la même façon (elle dépendra de sa place dans le message, mais aussi de la clé utilisée).

Programmation du cryptosystème sur python :

```
def vigenere(text, key, type='decrypt'):
    characters = [c for c in (chr(i) for i in range(32,127))] # 32 - 127 sont tous les
characteres visibles
    char_len = len(characters)

    if type not in ('decrypt', 'encrypt'):
        print('type doit être "decrypt" ou "encrypt".')
        return
    if any(t not in characters for t in key):
        print('caracteres invalides dans Clef. Il doit être symboles.')
        return
    if any(t not in characters for t in text ):
        print('caracteres invalides dans texte. Il doit être symboles.')
        return

    ret_text = '' # texte retourné
    k_len = len(key) # logneur de key

    for i, l in enumerate(text):
        # l'indice de l'alphabet qu'on va crypter ou decrypter
        text_idx = characters.index(l)
        # i % k_len :  index de cet alphabet clef par ex {1,5,6,9} i % k_len soit 0, 1, 2
ou 3 et key[i % k_len]  soit 1,5,6, our 9
        k = key[i % k_len]
        # le nombre qu'on va ajouter au char de texte correspondant
        key_idx = characters.index(k)
        if type == 'decrypt':
        # en cas au decryptage on va soustraire ce nombre
            key_idx *= -1
```

```
    # cryptage en cas key_idx >= 0 decryptage sinon
    code = characters[(text_idx + key_idx) % char_len]

    ret_text += code

  return ret_text
```

2.chiffrement de Hill

Présentation de l'algorithme :

Le chiffre de Hill utilise un alphabet et une matrice carrée M de taille n composée de nombres entiers et appelée matrice de chiffrement.

Exemple : Chiffrer le texte DCODE avec l'alphabet ABCDEFGHIJKLMNOPQRSTUVWXYZ et la matrice M d'ordre 2:M=[2357]

Découper le texte en n-grammes. Compléter tout n gramme incomplet final avec des lettres aléatoires si besoin.

Exemple : La matrice M est une matrice 2x2, DCODE devient DC,OD,EZ (un Z a été rajouté pour compléter le dernier bigramme)

Substituer les lettres du message clair par leur rang dans l'alphabet à partir de 00.

Exemple : L'alphabet ABCDEFGHIJKLMNOPQRSTUVWXYZ devient A=0,B=1,…,Z=25.

Les groupes de lettres DC, OD, EZ deviennent les groupes de valeurs (3,2), (14,3), (4,25)

Il est envisageable (mais déconseillé) d'utiliser ZABCDEFGHIJKLMNOPQRSTUVWXY pour avoir A=1,B=2,…,Y=25,Z=0.

Pour chaque groupe de valeurs P du texte clair (mathématiquement équivalent à un vecteur de taille n) effectuer le calcul matriciel :$M.P \equiv C \bmod 26$ où C est le groupe de valeurs chiffrées calculé et 26 la longueur de l'alphabet.

Exemple :$[2357]*[32] \equiv [123] \bmod 26$

A partir des valeurs chiffrées C, retrouver les lettres chiffrées de même rang dans l'alphabet.

Exemple : 1212 équivaut à M et 33 équivaut à D etc.

Ainsi de suite, DCODEZ se chiffre MDLNFN

Programmation du cryptosystème sur python :

```
            import string
import numpy as np
from sympy import Matrix #inverse key
```

Programmation de certains cryptosystemes

```
def letterToNumber(letter):
        return string.ascii_lowercase.index(letter)

def numberToLetter(number):
    return chr(int(number) + 97)

module = 26 #alphabets

def hill(raw_message, key, type = "encrypt"):
    message = []

    key_rows = key.shape[0]
    key_columns = key.shape[1]

    if type == "encrypt":
        for i in range(0, len(raw_message)):
                current_letter = raw_message[i:i+1].lower()
                if current_letter != ' ':
                    letter_index = letterToNumber(current_letter)

                    message.append(letter_index)


        #Le message doit être MULTIPLE de ligne de vecteurs
        #sinon on ajoute la premier alphabet de message à la fin
        if len(message) % key_rows != 0:
            for i in range(0, len(message)):
                message.append(message[i])
                if len(message) % key_rows == 0:
                    break

        #transformer de message à numpy array
        message = np.array(message)
        message_length = message.shape[0]
       #transformer le message array en matrice
        message.resize(int(message_length/key_rows), key_rows)
       encryption = np.matmul(message, key)
        encryption = np.remainder(encryption, module)
        encrypt_text = ""
        for i in range(len(encryption)):
            for j in range(len(encryption[i])):
                encrypt_text = encrypt_text + chr(encryption[i][j])
        return encrypt_text

       if type == "decrypt":
        arr = []
```

```
    for i in raw_message:
        arr.append(ord(i) % 26)
    arr = np.array(arr).reshape(-1,3)

    inverse_key = Matrix(key).inv_mod(module)
    inverse_key = np.array(inverse_key) #sympy to numpy
    inverse_key = inverse_key.astype(float)

    print("decryption:")
    decryption = np.matmul(arr, inverse_key)
    decryption = np.remainder(decryption, module).flatten()
    # print("decryption: ",decryption)

    decrypted_message = ""
    for i in range(0, len(decryption)):
        letter_num = int(decryption[i])
        letter = numberToLetter(decryption[i])
        decrypted_message = decrypted_message + letter
    return decrypted_message



 # Test
key = np.array([
[3, 10, 20],
[20, 9, 17],
[9, 4, 17]
])

# key = np.array([
#   [3, 5],
#   [6, 17]
# ])
encrypt = hill('ensakhouribga', key, 'encrypt')
decrypt = hill(encrypt, key, 'decrypt')
print("Message chiffré:   " + encrypt)
print("Message déchiffré: " + decrypt)
```

II. Cryptosystème moderne:

1. masque jetable :

Présentation de l'algorithme :

Le masque jetable est le seul algorithme de cryptage connu comme étant indécryptable. C'est en

fait un chiffre de Vigenère avec comme caractéristique que la clef de chiffrement a la même longueur que le message clair. Le système du masque jetable fut inventé par Gilbert Vernam en 1917, puis perfectionné par le major Joseph O. Mauborgne en 1918, qui inventa le concept de clef aléatoire

Le chiffrement par la méthode du masque jetable consiste à combiner le message en clair avec une

clé présentant les caractéristiques très particulières suivantes :

- La clé doit être une suite de caractères au moins aussi longue que le message à chiffrer.
- Les caractères composant la clé doivent être choisis de façon totalement aléatoire.
- Chaque clé, ou « masque », ne doit être utilisée qu'une seule fois (d'où le nom de masque jetable).

Programmation du cryptosystème sur python :

```
def masque_jetable(message, cle, type = 'encrypt'):
    if(len(cle) < len(message)):
        raise Exception('La cle doit etre au mois aussi longue que le message a
chiffrer.')
                        lettres = 'abcdefghijklmnopqrstuvwxyz'
    message_chiffre = ""
                 p = 0

                p = 0

    for char in message:
        int_char_message = lettres.index(char)
        int_char_cle = lettres.index(cle[p])
        if type == 'encrypt':
            int_char_chif = int((int_char_message + int_char_cle) % 26)
        elif type == 'decrypt':
            int_char_chif = int((int_char_message - int_char_cle) % 26)
        message_chiffre += lettres[int_char_chif]
        p += 1
        p = p % len(cle)

    return message_chiffre
```

```
CLE = "jqmalkzerwbaezrnjoiqr"
print("\n\n---Vernam Cypher---")

# Test
encrypt = vernam_chiffrement('ensakhouribga', CLE, 'encrypt')
decrypt = vernam_chiffrement(encrypt, CLE, 'decrypt')
print("Message chiffré:   " + encrypt)
print("Message déchiffré: " + decrypt)
```

2.code Rabin :

Principe du fonctionnement :

✓ Génération des clés :

Explicitement, la génération de clés est comme suit:

- Choisir deux grands nombres premiers, tels que **p ≡ 3 [4] et q ≡ 3 [4]**
- Posons **n=p*q,**ce qui fait de n la clé publique. Les nombres premiers p et q constituent la clé privée.

Pour chiffrer, on n'a besoin que de la clé publique, **n**, et pour déchiffrer, les facteurs de **n, p q**, sont nécessaires.

✓ Principe de chiffrement :

Pour le chiffrement, seule la clé publique, n, est utilisée. On produit le texte chiffré à partir du texte en clair m comme suit : **P = {1,.., n-1}** l'espace des textes en clair possibles (tous des nombres) et posons **m ϵ P** comme étant le texte en clair. Le texte chiffré **c** se détermine comme suit :

$$c = m^2 [n]$$

✓ Principe de déchiffrement :

Pour déchiffrer, la clé privée est nécessaire. Le processus est comme suit.

On calcule tout d'abord m_p *et* m_q *tels que*

$$m_p = c[p] \text{ et } m_q = c[q].$$

L' permet de calculer y_p et y_q tel que :

$$y_p * p + y_q * q = 1$$

On invoque alors le théorème des restes chinois pour calculer les quatre racines carrées +r, -r, +s, -s tels que:

$$r = (y_p * p * m_p + y_q * q * m_q)[n]$$

$$-r = n - r$$

$$s = (y_p * p * m_p - y_q * q * m_q)[n]$$

$$-s = n - r$$

Remarque :

Apres avoir déchiffré le message on obtient 4 résultats différents pour chaque caractère , Pourtant le récepteur n'arrive pas a détecté le caractère exacte mis en premier temps , c'est pour cette raison qu'on fait recourt aux fonctions de remplissage (voir l'implémentation sur python ci-dessous) afin d'obtenir le vrai message.

✓ ***Rappel sur le théorème des restes chinois :***

Soient des objets en nombre inconnu. Si on les range par 3 il en reste 2. Si on les range par 5, il en reste 3 et si on les range par 7, il en reste 2. Combien a-t-on d'objets ?

Théorème : Prenons m_1, ..., m_n des entiers supérieurs à 2 deux à deux premiers entre eux, et $a_1, \ldots, a_n$ des entiers. Le système d'équations :

$$\begin{cases} x & \equiv & a_1 \ [m_1] \\ \vdots & \vdots & \vdots \\ x & \equiv & a_n \ [m_n] \end{cases}$$

admet une unique solution modulo $M = m_1 \times \cdots m_n$ donnée par la formule :

$$x = a_1 M_1 y_1 + \cdots + a_n M_n y_N$$

où $M_i = M/m_i$ et $y_i \equiv M_i^{-1} \ [m_i]$ pour i compris entre 1 et n.

La plus petite valeur pour le nombre d'objets est alors 23

✓ ***Exemple illustratif :***

Dans notre exemple simple, se trouve notre espace en texte brut. Nous prendrons comme texte brut le texte chiffré est donc **P**={0,…..,76} m=20 c=m^2 mod n=400 mod 77 =15.

Pour exactement quatre valeurs différentes de **m**,le texte chiffré 15 est produit, c'est-à-dire pour. Cela est vrai pour la plupart des textes chiffrés produits par l'algorithme de Rabin, c'est-à-dire qu'il s'agit d'une fonction quatre pour un.**m** ∈ {13,20,57,64}

Code Rabin sur python(Algorithme de chiffrement) :

- ***Génération des nombres premiers :***

Générer deux très grands nombres premiers, p et q, qui satisfont à la condition **p ≠ q → p≡ q ≡ 3 (mod 4)**

Par exemple **: p=139 et q=191**

Calculer la valeurde **n = p.q**

Publier **n** en tant que clé publique et enregistrer **p** et **q** en tant que clé privée

```
import math
def isPrime(n):
    if n < 2 or n == 4:
```

```
        return False
    end = int(math.sqrt(n))
    if(end==2):
        end += 1
    for i in range(2, end):
        if n % i == 0:
            return False
    return True

arr = [i for i in range(150,200) if isPrime(i) and i%4==3]
arr
```

```
[151, 163, 167, 179, 191, 199]
```

- ***Fonction de chiffrement :***
  - Obtenez la clé publique n
  - Appliquer la formule : **C** = m^2**mod n**
  - Envoyer **C** au destinataire.

```
def chiffrer(text_clair, n):
    c = []
   for char in text_clair:
        char_ascii = ord(char)
#         char_ascii = ajouter_remplissage(char_ascii)
        int_encrypted = (char_ascii ** 2) % n
        c.append(chr(int_encrypted))
    return c
```

Cryptanalyse (Algorithme de déchiffrement):

- ***Bézout :***

```
def bezout(a, b):
    u, v, uu, vv = 1, 0, 0, 1
    while b:
        q, rr = divmod(a, b)
       a, b = b, rr
        u, uu = uu, u - q*uu
        v, vv = vv, v - q*vv

    return u, v
```

Remplissage :

Le remplissage est l'une des nombreuses pratiques distinctes qui incluent toutes l'ajout de données au début, au milieu ou à la fin d'un message avant le cryptage. En cryptographie classique, le remplissage peut inclure l'ajout de phrases absurdes à un message pour masquer le fait que de nombreux messages se terminent de manière prévisible, par exemple *sincèrement vôtre* .

La méthode qu'on a choisit pour ce projet est de convertir le nombre en binaire puis le doubler (101 devient 101101)

Ajouter_remplissage : elle convertit le message en valeur ASCII. Ensuite, le convertit en binaire et étend la valeur binaire avec lui-même, puis elle convertit la valeur binaire en décimal.

Verifier_remplissage\supp_remplissage: Convertit-les caractères en binaire et les divises toutes en deux. Détermine dans quelles moitiés gauche et droite sont identiques. Conserve la moitié de ce binaire et convertit-le en m décimal. On obtient donc le caractère ASCII pour la valeur décimale m. Le caractère résultant donne le message correct envoyé par l'expéditeur.

```
def ajouter_remplissage(n):
    binaire = bin(n)[2:]
    binaire = binaire + binaire
    return int(binaire, 2)

def verifier_remplissage(n):
    binaire = bin(n)[2:]
    length = len(binaire)
    if binaire[length//2:] == binaire[:length//2]:
        return True
    return False

def supp_remplissage(n):
    binaire = bin(n)[2:]
    length = len(binaire)
    binaire = binaire[length//2:]
    return int(binaire,2)
```

Exemple illustratif :

```
# TEST
# CONDITION: n > ajouter_remplissage(127)
# i.e. n > 16383
```

```
  p, q = 199, 179
  n = p * q
  M = "ensa"
  print(f"p = {p}, q = {q}, n = p x q = {n}")
  print(f'M = "{M}"')
  print("-"*50)

  message_chiffre = chiffrer(M, n)
  print("Message chiffrée")
  print(encrypted)
  print("-"*50)

  print("Message dechiffrée")
  message_dechiffre = dechiffrer(message_chiffre, p, q)
  for item in message_dechiffre:
      print(item)
  print("-"*50)
```

```
p = 199, q = 179, n = p x q = 35621 M = "ensa"
--------------------------------------------------
Message chiffrée ['僨', '嘁', '◉', '冊']
--------------------------------------------------
Message dechiffrée ['Ⴠ', '疤', 'e', '諀']
['', '茆', 'n', '調']
 ['潠', '丐', '課', 's']
['諄', 'a', '⊕', '楏']
--------------------------------------------------
```

Apres avoir appliqué les fonctions de remplissage on obtient le résultat suivant :

```
p = 199, q = 179, n = p x q = 35621 M = "ensa"
-------------------------------------------------- Message chiffrée ['僨', '嘁', '◉',
'冊'] -------------------------------------------------- Message dechiffrée ['e']
['n'] ['s'] ['a'] --------------------
```

Algorithme de signature numérique :

Le Cryptosystème Rabin peut être utilisé pour créer et vérifier des signatures numériques. La création d'une signature nécessite la clé privée(p,q). La vérification d'une signature nécessite la clé publique$n$.

Un message m<n peut être signé avec une clé privée (p,q) comme suit.

- Générez une valeur aléatoire$u$.
- Utilisez une fonction de hachage cryptographique H pour calculer $c=H(m\vee u)$, où la barre indique la concaténation. c doit être un entier inférieur à n.
- Traitez c comme une valeur chiffrée par Rabin et essayez de la déchiffrer, en utilisant la clé privée(p,q). Cela produira les quatre résultats habituels, r_1,r_2,r_3,r_4.

- On pourrait s'attendre à ce que le cryptage de chaque $r_i$produise$c$. Cependant, cela ne sera vrai que si c est un résidu quadratique modulo p et q. Pour déterminer si c'est le cas, chiffrez le premier résultat de déchiffrementr_1. S'il ne crypte pas en$c$, répétez cet algorithme avec un nouveau u aléatoire. Le nombre attendu de fois où cet algorithme doit être répété avant de trouver un c approprié est de 4.
- Ayant trouvé un $r_1$qui crypte en c, la signature est $(r¿¿1, u)¿$.

➪ ***Vérification d'une signature :***

Une signature$(r¿¿1, u)¿$. Pour un message m peut être vérifiée à l'aide de la clé publique n comme suit.

- Calculez $c = H(m \vee u)$.
- Chiffrez r à l'aide de la clé publique$n$.
- La signature est valide si et seulement si le chiffrement de r est égal à c.

CONCLUSION

La cryptographie asymétrique est intrinsèquement lente à cause des calculs complexes qui y sont associés, alors que la cryptographie symétrique brille par sa rapidité. Toutefois, cette dernière souffre d'une grave lacune, on doit transmettre les clés de manière sécurisée (sur un canal authentifié). Pour pallier ce défaut, on recourt à la cryptographie asymétrique qui travaille avec une paire de clés : la clé privée et la clé publique. La cryptographie hybride combine les deux systèmes afin de bénéficier des avantages (rapidité de la cryptographie symétrique pour le contenu du message) et utilisation de la cryptographie « lente » uniquement pour la clé.