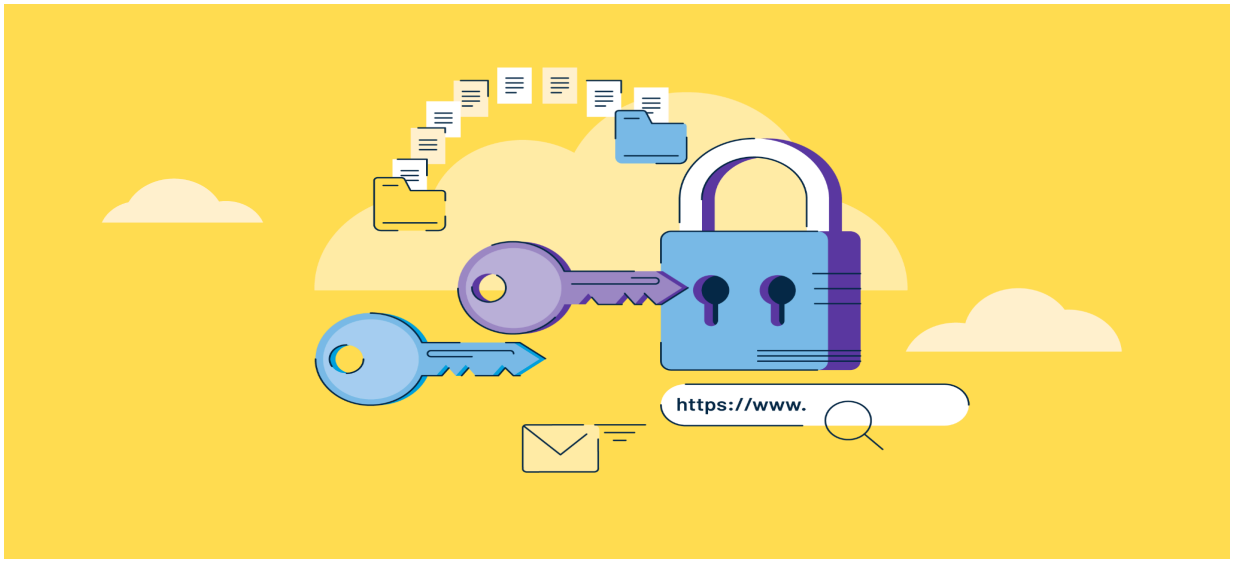
Cryptographie avancé

Compte Rendu
RSA



Année Universitaire: 2023/2024

LEKOUAGHET imad eddine

Introduction à l'Algorithme de Chiffrement RSA

Introduction à l'Algorithme de Chiffrement RSAL'algorithme RSA (Rivest-Shamir-Adleman) est l'un des systèmes de chiffrement asymétrique les plus largement utilisés dans le domaine de la cryptographie moderne. Développé en 1977 par Ron Rivest, Adi Shamir et Leonard Adleman, il repose sur des principes mathématiques complexes, notamment la théorie des nombres, pour permettre un échange sécurisé de données sur des réseaux informatiques non sécurisés.

Contrairement aux systèmes de chiffrement symétriques, où une seule clé est utilisée pour à la fois chiffrer et déchiffrer les données, RSA utilise un système à clé publique et clé privée. Ce système permet à chaque participant de générer une paire de clés : une clé publique, connue de tous, et une clé privée, conservée secrètement.

Dans ce contexte, les fonctions de chiffrement et de déchiffrement jouent un rôle crucial. La fonction de chiffrement utilise la clé publique pour transformer les données en un format illisible sans la connaissance de la clé privée correspondante, tandis que la fonction de déchiffrement utilise la clé privée pour reconvertir les données chiffrées en leur forme originale.

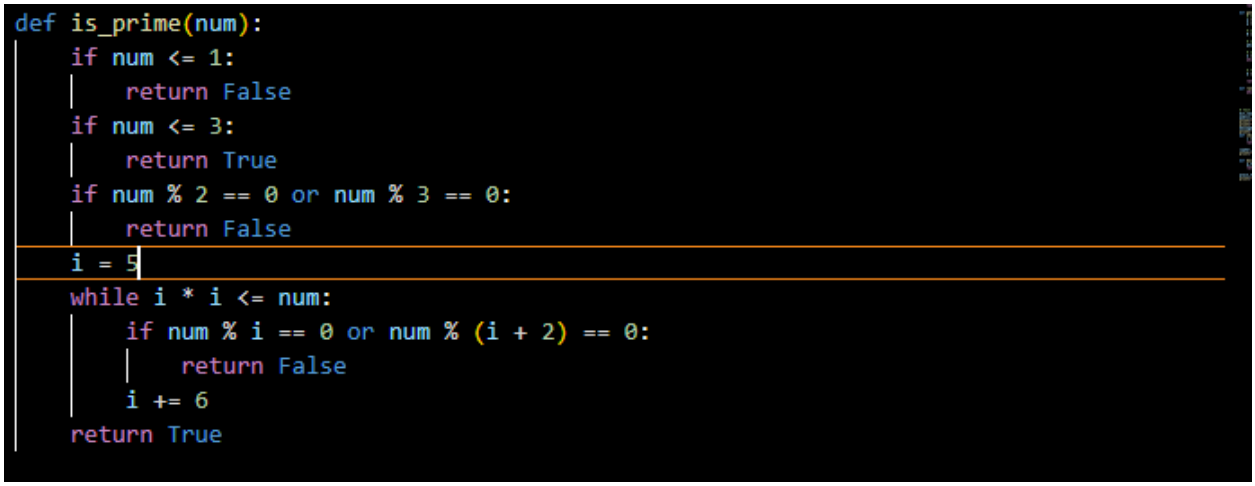
Dans ce rapport, nous allons explorer en détail les différentes fonctions impliquées dans l'algorithme RSA, en mettant particulièrement l'accent sur les fonctions de chiffrement et de déchiffrement, ainsi que sur les principes mathématiques qui les sous-tendent.

L'algorithme RSA, un pilier de la cryptographie moderne, repose sur plusieurs fonctions clés pour assurer la sécurité des échanges de données. Dans cette revue, nous explorerons brièvement chacune de ces fonctions, soulignant leur rôle dans le processus de chiffrement et de déchiffrement.

Les fonctions clés de l'algorithme RSA sont : **is_prime(num), gcd(a, b), mod_inverse(a, m)**, **generate_keypair(bit_length)**, **random_prime(bit_length)**, **chiffrement(message, public_key, e)**, et **dechiffrement(message, private_key)**.

Fonction is_prime(num):

Cette fonction détermine si un nombre est premier ou non en vérifiant s'il est divisible par un nombre autre que 1 et lui-même. Elle suit l'algorithme de test de primalité appelé "Test de primalité de Rabin-Miller", qui est efficace pour des nombres de tailles raisonnables.



```
def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True
```

Fonction gcd(a, b):

Cette fonction calcule le plus grand commun diviseur (PGCD) de deux nombres a et b en utilisant l'algorithme d'Euclide. Elle réduit récursivement les nombres jusqu'à ce que l'un d'eux soit égal à zéro, puis renvoie l'autre nombre.

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Fonction mod_inverse(a, m):

Cette fonction calcule l'inverse modulaire de a modulo m. C'est-à-dire, elle trouve un entier x tel que (a * x) % m == 1. Elle utilise l'algorithme d'inversion modulo étendu, qui est une modification de l'algorithme d'Euclide étendu.

```python
def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1
```

Fonction generate_keypair(bit_length):

Cette fonction génère une paire de clés RSA (une clé publique et une clé privée) en utilisant l'algorithme de génération de clés RSA.

Elle génère deux nombres premiers distincts p et q de la taille spécifiée en bits.Calcule le produit N = p * q.Calcule la fonction d'Euler φ(N) = (p - 1) * (q - 1).Choisisse un exposant public e tel que 1 < e < φ(N) et gcd(e, φ(N)) == 1.Trouve l'exposant privé d tel que (d * e) % φ(N) == 1.

```python
def generate_keypair(bit_length):
    # Choisir deux nombres premiers distincts p et q
    p = random_prime(bit_length)
    q = random_prime(bit_length)

    # Calculer le produit de ces nombres premiers
    N = p * q

    # Calculer la fonction d'Euler ϕ(N)
    phi_N = (p - 1) * (q - 1)

    # Choisir un entier e tel que 1 < e < ϕ(N) et gcd(e, ϕ(N)) == 1
    e = random.randrange(1, phi_N)
    while gcd(e, phi_N) != 1:
        e = random.randrange(1, phi_N)

    # Trouver l'entier d tel que (d * e) % ϕ(N) == 1
    d = mod_inverse(e, phi_N)

    return ((N, e), (N, d))
```

Fonction random_prime(bit_length):

Cette fonction génère un nombre premier aléatoire de la taille spécifiée en bits. Elle utilise la fonction is_prime(num) pour vérifier si le nombre généré est premier ou non.

```
def random_prime(bit_length):
    while True:
        num = random.getrandbits(bit_length)
        if is_prime(num):
            return num
```

Fonction chiffrement(message, public_key, e):

Cette fonction prend en entrée un message à chiffrer, la clé publique RSA et l'exposant public e. Elle chiffre le message en utilisant l'algorithme de chiffrement RSA, qui repose sur l'exponentiation modulaire.

Exponentiation modulaire : Elle élève le message à la puissance de l'exposant public e modulo N, où N est le produit des nombres premiers utilisés pour générer la clé publique.

Renvoi du message chiffré : Le résultat de l'exponentiation modulaire est le message chiffré, qu'elle renvoie à l'appelant.

```
def chiffrement ( message, public_key, e):
    c = pow(message, e, public_key[0])
    return c
```

Fonction dechiffrement(message, private_key):

Cette fonction prend en entrée un message chiffré et la clé privée RSA. Elle déchiffre le message chiffré en utilisant l'algorithme de déchiffrement RSA.

Exponentiation modulaire : Elle élève le message chiffré à la puissance de l'exposant privé d modulo N, où N est le produit des nombres premiers utilisés pour générer la clé privée.

Renvoi du message déchiffré : Le résultat de l'exponentiation modulaire est le message déchiffré, qu'elle renvoie à l'appelant.

```
def dechiffrement( message , private_key):
    p_text = pow(message, private_key[1], private_key[0])
    return p_text
```

L'Exécution:

```
Clé publique: (613466014250324167, 145478947300507213)
Clé privée: (613466014250324167, 111643468681163917)
le texte en clair est  123456
le texte chiffré :  250315150265280821
le texte déchiffré :  123456
```