

Système de Chiffrement Asymétrique Basé sur RSA

Rapport de Projet - Cryptographie

17 novembre 2025

Table des matières

Chapitre 1

Introduction

Ce projet implémente un système complet de chiffrement asymétrique utilisant l'algorithme RSA (Rivest-Shamir-Adleman). L'objectif est de démontrer les concepts fondamentaux de la cryptographie asymétrique, incluant :

- Le chiffrement et déchiffrement de messages
- La création et vérification de signatures numériques
- La gestion sécurisée des clés
- Les applications pratiques de la cryptographie asymétrique

1.1 Motivation

La cryptographie asymétrique est essentiellement pour :

- Sécuriser les communications (HTTPS, TLS)
- Authentifier les utilisateurs
- Signer des documents numériquement
- Implémenter les signatures de blockchain

1.2 Objectifs du Projet

1. Implémenter une classe Python complète pour RSA
2. Démontrer les opérations de chiffrement/déchiffrement
3. Implémenter les signatures numériques
4. Gérer la persistance des clés
5. Créer un tutoriel interactif pédagogique
6. Fournir un menu interactif pour expérimenter

Chapitre 2

Fondamentaux de la Cryptographie Asymétrique

2.1 Principes de Base

La cryptographie asymétrique utilise une **paire de clés** :

- **Clé Publique** (K_{pub}) : Partagée publiquement, utilisée pour chiffrer les messages
- **Clé Privée** (K_{priv}) : Gardée secrète, utilisée pour déchiffrer les messages

2.1.1 Analogie

Clé publique = Cadenas (tout le monde peut l'utiliser)

Clé privée = Clé du cadenas (seul vous la possédez)

2.2 Algorithme RSA

2.2.1 Génération des Clés

1. Choisir deux nombres premiers distincts p et q
2. Calculer $n = p \times q$
3. Calculer $\phi(n) = (p - 1) \times (q - 1)$
4. Choisir un entier e tel que $1 < e < \phi(n)$ et $\gcd(e, \phi(n)) = 1$
5. Calculer $d = e^{-1} \pmod{\phi(n)}$

Clé Publique = (e, n)

Clé Privée = (d, n)

2.2.2 Chiffrement

Pour chiffrer un message M :

$$C = M^e \pmod{n}$$

Où :

- C = ciphertext (message chiffré)
- M = plaintext (message clair)
- e = exposant public
- n = modulus

2.2.3 Déchiffrement

Pour déchiffrer le message C :

$$M = C^d \pmod{n}$$

Où d est l'exposant privé.

2.3 RSA-OAEP

Notre implémentation utilise **RSA-OAEP** (Optimal Asymmetric Encryption Padding), qui est plus sécurisée que RSA brut :

- **OAEP** ajoute du remplissage aléatoire au message avant chiffrement
- Chaque chiffrement du même message produit un ciphertext différent
- Protège contre les attaques par analyse de pattern
- Utilise SHA-256 pour la fonction de hachage

2.4 Signatures Numériques

2.4.1 Création d'une Signature

Pour signer un message M :

$$\text{Signature} = \text{SHA256}(M)^d \pmod{n}$$

Seul le propriétaire de la clé privée peut créer une signature valide.

2.4.2 Vérification d'une Signature

Pour vérifier :

$$\text{SHA256}(M) \stackrel{?}{=} \text{Signature}^e \pmod{n}$$

N'importe qui avec la clé publique peut vérifier l'authenticité.

Chapitre 3

Architecture du Système

3.1 Structure du Projet

```
1 FINAL_SUBMISSION/  
2     presentation/  
3         Cryptage_Asymetrique.pdf  
4     sources/  
5         asymmetric_crypto.py  
6         tutorial.py  
7     documentation/  
8         requirements.txt
```

3.2 Classe AsymmetricCrypto

La classe principale fournit tous les mécanismes de cryptographie :

3.2.1 Attributs

- **key_size** : Taille de la clé RSA (2048, 3072, ou 4096 bits)
- **private_key** : Clé privée chargée
- **public_key** : Clé publique chargée

3.2.2 Méthodes Principales

generate_key_pair() Génère une nouvelle paire de clés RSA

encrypt(plaintext) Chiffre un message avec la clé publique

decrypt(ciphertext_b64) Déchiffre avec la clé privée

sign(message) Crée une signature numérique

verify(message, signature_b64) Vérifie une signature

save_private_key(filename, password) Sauvegarde la clé privée (avec protection par mot de passe optionnel)

load_private_key(filename, password) Charge la clé privée depuis un fichier

save_public_key(filename) Sauvegarde la clé publique

load_public_key(filename) Charge la clé publique d'un autre utilisateur
encrypt_large_data(data, chunk_size) Chiffre de grandes données en morceaux
decrypt_large_data(encrypted_chunks) Déchiffre des morceaux chiffrés

Chapitre 4

Implémentation Détaillée

4.1 Module asymmetric_crypto.py

Ce module contient :

4.1.1 Classe AsymmetricCrypto

```
1 from cryptography.hazmat.primitives.asymmetric import rsa,
   padding
2 from cryptography.hazmat.primitives import hashes, serialization
3 from cryptography.hazmat.backends import default_backend
4 import base64
5 import os
6
7 class AsymmetricCrypto:
8     def __init__(self, key_size=2048):
9         self.key_size = key_size
10        self.private_key = None
11        self.public_key = None
```

4.1.2 Chiffrement avec RSA-OAEP

```
1 def encrypt(self, plaintext):
2     if isinstance(plaintext, str):
3         plaintext = plaintext.encode('utf-8')
4
5     ciphertext = self.public_key.encrypt(
6         plaintext,
7         padding.OAEP(
8             mgf=padding.MGF1(algorithm=hashes.SHA256()),
9             algorithm=hashes.SHA256(),
10            label=None
11        )
12    )
13
14    return base64.b64encode(ciphertext).decode('utf-8')
```

Points clés :

- Utilise OAEP avec SHA-256
- Encode en base64 pour transmission facile
- Accepte chaînes ou octets en entrée

4.1.3 Déchiffrement

```
1 def decrypt(self, ciphertext_b64):
2     ciphertext = base64.b64decode(ciphertext_b64)
3
4     plaintext = self.private_key.decrypt(
5         ciphertext,
6         padding.OAEP(
7             mgf=padding.MGF1(algorithm=hashes.SHA256()),
8             algorithm=hashes.SHA256(),
9             label=None
10        )
11    )
12
13    return plaintext.decode('utf-8')
```

4.1.4 Signatures Numériques

```
1 def sign(self, message):
2     if isinstance(message, str):
3         message = message.encode('utf-8')
4
5     signature = self.private_key.sign(
6         message,
7         padding.PSS(
8             mgf=padding.MGF1(hashes.SHA256()),
9             salt_length=padding.PSS.MAX_LENGTH
10        ),
11        hashes.SHA256()
12    )
13
14    return base64.b64encode(signature).decode('utf-8')
```

Points clés :

- Utilise PSS (Probabilistic Signature Scheme)
- Chaque signature est unique (aléatoire)
- Seule la clé privée peut signer

4.1.5 Vérification de Signature

```
1 def verify(self, message, signature_b64):
2     if isinstance(message, str):
3         message = message.encode('utf-8')
4
```

```

5     signature = base64.b64decode(signature_b64)
6
7     try:
8         self.public_key.verify(
9             signature,
10            message,
11            padding.PSS(
12                mgf=padding.MGF1(hashes.SHA256()),
13                salt_length=padding.PSS.MAX_LENGTH
14            ),
15            hashes.SHA256()
16        )
17        return True
18    except Exception:
19        return False

```

4.1.6 Gestion des Clés

Sauvegarde de Clé Privée

```

1  def save_private_key(self, filename, password=None):
2      if password:
3          encryption_algorithm = serialization.
4              BestAvailableEncryption(
5                  password.encode()
6              )
7      else:
8          encryption_algorithm = serialization.NoEncryption()
9
10     pem = self.private_key.private_bytes(
11         encoding=serialization.Encoding.PEM,
12         format=serialization.PrivateFormat.PKCS8,
13         encryption_algorithm=encryption_algorithm
14     )
15
16     with open(filename, 'wb') as f:
17         f.write(pem)

```

Format PEM : Format texte standard pour les clés cryptographiques.

Chargement de Clé Privée

```

1  def load_private_key(self, filename, password=None):
2      with open(filename, 'rb') as f:
3          pem = f.read()
4
5      pwd = password.encode() if password else None
6
7      self.private_key = serialization.load_pem_private_key(
8          pem,

```

```
9         password=pwd,
10         backend=default_backend()
11     )
12     self.public_key = self.private_key.public_key()
```

4.1.7 Chiffrement de Grandes Données

RSA a une limite de taille (basée sur la taille de clé). Pour les grands messages :

```
1 def encrypt_large_data(self, data, chunk_size=190):
2     if isinstance(data, str):
3         data = data.encode('utf-8')
4
5     chunks = [data[i:i+chunk_size] for i in range(0, len(data),
6         chunk_size)]
7     encrypted_chunks = []
8
9     for chunk in chunks:
10         encrypted = self.encrypt(chunk)
11         encrypted_chunks.append(encrypted)
12
13     return encrypted_chunks
```

Note : 190 octets est l'optimal pour RSA-2048.

4.2 Module tutorial.py

Le tutoriel interactif guide l'utilisateur à travers 14 étapes :

4.2.1 Structure du Tutoriel

1. Génération de paire de clés
2. Chiffrement d'un message
3. Déchiffrement du message
4. Démonstration de l'aléatoire d'encryption
5. Création de signature numérique
6. Vérification de signature
7. Détection de tamponnage (signature invalide)
8. Sauvegarde des clés dans des fichiers
9. Chargement des clés depuis des fichiers
10. Test de chiffrement inter-instance
11. Communication sécurisée Alice & Bob
12. Message chiffré de Bob vers Alice
13. Réponse signée d'Alice vers Bob
14. Résumé et cas d'usage

4.2.2 Fonctions Utilitaires

```
1 def print_section(title, num):
2     print(f"\n{'='*70}")
3     print(f"STEP_{num}:_{title}")
4     print(f"{'='*70}\n")
5
6 def print_explanation(text):
7     print(f"    _{text}\n")
8
9 def print_code(code):
10    print("    _CODE:")
11    print(f"    __{code}\n")
12
13 def wait_input():
14    input("Press ENTER to continue...")
```

4.3 Fonctions de Démonstration

Le module `asymmetric_crypto.py` contient 5 démonstrations :

4.3.1 `demo_basic_encryption()`

Démontre le chiffrement/déchiffrement basique.

4.3.2 `demo_digital_signature()`

Montre la création et vérification de signatures.

4.3.3 `demo_key_persistence()`

Sauvegarde et charge des clés entre instances.

4.3.4 `demo_large_data()`

Chiffre et déchiffre des données volumineuses.

4.3.5 `demo_secure_communication()`

Simule une communication sécurisée Alice & Bob avec :

- Échange de clés publiques
- Message chiffré de Bob vers Alice
- Réponse signée d'Alice vers Bob

Chapitre 5

Utilisation du Système

5.1 Installation

```
1 pip install -r requirements.txt
```

Les dépendances :

— **cryptography** : Bibliothèque de cryptographie (version \geq 41.0.0)

5.2 Exécution du Tutoriel

```
1 python tutorial.py
```

Le tutoriel est interactif : attendez entre les étapes pour continuer.

5.3 Exécution du Menu Interactif

```
1 python asymmetric_crypto.py
```

Options du menu :

1. Générer une nouvelle paire de clés
2. Chiffrer un message
3. Déchiffrer un message
4. Signer un message
5. Vérifier une signature
6. Sauvegarder les clés
7. Charger les clés
8. Voir les infos des clés
9. Exécuter les 5 démonstrations
10. Quitter

5.4 Exemple d'Utilisation

```
1 from asymmetric_crypto import AsymmetricCrypto
2
3 crypto = AsymmetricCrypto(key_size=2048)
4 crypto.generate_key_pair()
5
6 message = "Secret_Message"
7 encrypted = crypto.encrypt(message)
8 print(f"Encrypted: {encrypted}")
9
10 decrypted = crypto.decrypt(encrypted)
11 print(f"Decrypted: {decrypted}")
12
13 signature = crypto.sign(message)
14 is_valid = crypto.verify(message, signature)
15 print(f"Signature valid: {is_valid}")
16
17 crypto.save_private_key('my_key.pem', password='secure')
```

Chapitre 6

Cas d'Utilisation Pratiques

6.1 Chiffrement de Messages

Envoi de messages secrets via un canal non sécurisé :

- Alice génère une paire de clés
- Alice partage sa clé publique
- Bob chiffre un message avec la clé publique d'Alice
- Seule Alice peut déchiffrer (clé privée)

6.2 Authentification

Prouver qui vous êtes :

- Vous signez un message avec votre clé privée
- N'importe qui peut vérifier avec votre clé publique
- Prouve l'authenticité et l'intégrité

6.3 HTTPS et Certificats SSL/TLS

- Les certificats contiennent une clé publique
- Le serveur signe avec sa clé privée
- Le client vérifie avec la clé publique du serveur
- Établit une connexion sécurisée

6.4 Email Sécurisé (PGP/GPG)

- Chiffrer les emails avec la clé publique du destinataire
- Signer les emails avec votre clé privée
- Destinataire déchiffre et vérifie

6.5 Blockchain et Bitcoin

- Les adresses Bitcoin sont dérivées de clés publiques
- Les transactions sont signées avec des clés privées
- N'importe qui peut vérifier les signatures

- Impossible à contrefaire (mathématiquement)

Chapitre 7

Propriétés de Sécurité

7.1 Confidentialité

- Le chiffrement RSA-OAEP protège le secret du message
- Seul le propriétaire de la clé privée peut déchiffrer
- Attaque par force brute nécessite des millénaires (pour RSA-2048)

7.2 Authentification

- Les signatures prouvent qui a créé le message
- Seule la clé privée peut créer une signature
- N'importe qui peut vérifier avec la clé publique

7.3 Intégrité

- Si le message est modifié, la signature échoue
- Détecte le tamponnage
- Impossible de forger une signature sans la clé privée

7.4 Non-Répudiation

- Le signataire ne peut pas nier avoir signé
- La signature est cryptographiquement liée à la clé privée
- Légalement valide (dans certaines juridictions)

Chapitre 8

Tailles de Clés et Sécurité

8.1 Recommandations

Taille de Clé	Sécurité	Cas d'Usage
1024-bit	Faible	Déprécié (ne pas utiliser)
2048-bit	Bon	Utilisations générales jusqu'en 2030
3072-bit	Meilleur	Données sensibles à long terme
4096-bit	Excellent	Sécurité maximale (plus lent)

8.2 Considérations de Performance

- Génération de clé : 2048-bit \approx quelques secondes
- Chiffrement : 2048-bit \approx quelques millisecondes
- Déchiffrement : Plus lent que chiffrement
- 4096-bit \approx 10x plus lent que 2048-bit

Chapitre 9

Résultats et Tests

9.1 Résultats du Tutoriel

Le tutoriel a été exécuté avec succès, démontrant :

- Génération de clés RSA-2048
- Chiffrement et déchiffrement corrects
- Aléatoire d'encryption (chaque chiffrement différent)
- Signature valide
- Détection de tamponnage (signature invalide)
- Sauvegarde et chargement des clés
- Chiffrement inter-instance réussi
- Communication Alice & Bob sécurisée

9.2 Observations

1. **Sécurité RSA-OAEP** : Chaque chiffrement du même message produit un ciphertext différent, augmentant la sécurité.
2. **Intégrité des Signatures** : Même un changement d'un seul caractère rend la signature invalide.
3. **Persistance des Clés** : Les clés sauvegardées peuvent être restaurées dans différentes instances.
4. **Format PEM** : Format standard reconnu par tous les outils cryptographiques.

Chapitre 10

Limitations et Améliorations Futures

10.1 Limitations Actuelles

- **Taille de clé fixe** : Doit redémarrer pour changer la taille
- **Pas de gestion de certificats** : Juste les clés brutes
- **Limitation de taille pour RSA** : Nécessite le chunking pour les gros fichiers
- **Pas d'authentification multi-utilisateur**

10.2 Améliorations Potentielles

- Implémenter les certificats X.509
- Ajouter le chiffrement hybride (RSA + AES)
- Implémenter les keyrings sécurisés
- Interface graphique (GUI) pour les non-programmeurs
- Support pour d'autres algorithmes (ECDSA, EdDSA)
- Authentification par token JWT
- Intégration avec des systèmes PKI

Chapitre 11

Conclusion

Ce projet démontre les concepts fondamentaux de la cryptographie asymétrique. L'implémentation complète fournit :

- Une classe robuste et facile à utiliser
- Des opérations de chiffrement et signature
- La gestion sécurisée des clés
- Un tutoriel interactif pédagogique
- Des exemples pratiques

Les applications de la cryptographie asymétrique sont vastes :

- Sécuriser la communication
- Authentifier les utilisateurs
- Vérifier l'intégrité des données
- Implémenter les signatures numériques
- Supporter les systèmes blockchain

La compréhension de ces concepts est essentielle pour tout développeur travaillant sur des systèmes sécurisés.

Chapitre 12

Références et Ressources

12.1 Bibliographie

- Rivest, R. L., Shamir, A., & Adleman, L. (1978). "A method for obtaining digital signatures and public-key cryptosystems"
- NIST SP 800-38D - Recommendation for Block Cipher Modes of Operation
- RFC 3447 - PKCS #1 : RSA Cryptography Specifications
- OWASP - Cryptographic Storage Cheat Sheet

12.2 Documentation

- Cryptography.io - <https://cryptography.io>
- Python Documentation - <https://docs.python.org>
- Wikipedia - RSA Algorithm - <https://en.wikipedia.org/wiki/RSA>

Annexe A

Code Source Complet

A.1 asymmetric_crypto.py

Voir le fichier source fourni.

A.2 tutorial.py

Voir le fichier source fourni.

A.3 requirements.txt

```
1 cryptography >=41.0.0
```