

## *Cryptographie appliquée*

### *Projet #3 - Génération de clés, VM et entropie*



#### **Participants :**

- **YOUCEF AIT EL HADJ 4A – Systèmes embarqués et autonomes**
- **MOHAMED AMEZIANE SIDI MAMMAR 4A – Software Engineering**

## Introduction :

Le besoin de communication entre les humains est essentiel. Il arrive juste après le besoin de survie, les moyens de communication entre les êtres humains ont évolué d'une génération à une autre. Aujourd'hui, nous vivons à l'ère des ordinateurs, des systèmes et des réseaux informatiques où l'évolution des technologies de l'information suit une courbe quasi exponentielle.

Dans la société de l'information de ces dernières années, où toutes les informations sont échangées sur des réseaux, nous sommes constamment exposés à des menaces de sécurité telles que l'écoute clandestine et la falsification. La technologie cryptographique est utilisée pour faire face à ces problèmes, elle est indispensable pour protéger les informations contre ces menaces. En cryptographie, le cryptage est le processus de codage d'un message ou d'une information de manière à ce que seules les parties autorisées puissent y accéder et que celles qui ne sont pas autorisées ne le puissent pas, l'utilisation de la technologie de cryptage des données est un moyen important pour la communication sécurisée.

## Définition du projet :

### 1. Générations des clés RSA :

Dans ce projet nous allons générer des clés RSA (publiques, privés).

Le chiffrement RSA est un algorithme de cryptographie asymétrique, il utilise une paire de clés (nombres entiers) composée d'une clé publique pour chiffrer et d'une clé privée pour déchiffrer des données confidentielles.

### Etapes de création des clés RSA :

- Choisir  $p$  et  $q$ , deux nombres premiers distincts.
- Calculer  $\phi(n) = (p - 1) * (q - 1)$
- Choisir un entier naturel *premier avec*  $\phi(n)$  et strictement inférieur à  $\phi(n)$ , appelé *exposant de chiffrement* ;
- Calculer l'entier naturel  $d$ , inverse modulaire de  $e$  pour la multiplication modulo  $\phi(n)$  et strictement inférieur à  $\phi(n)$ , appelé *exposant de déchiffrement*.

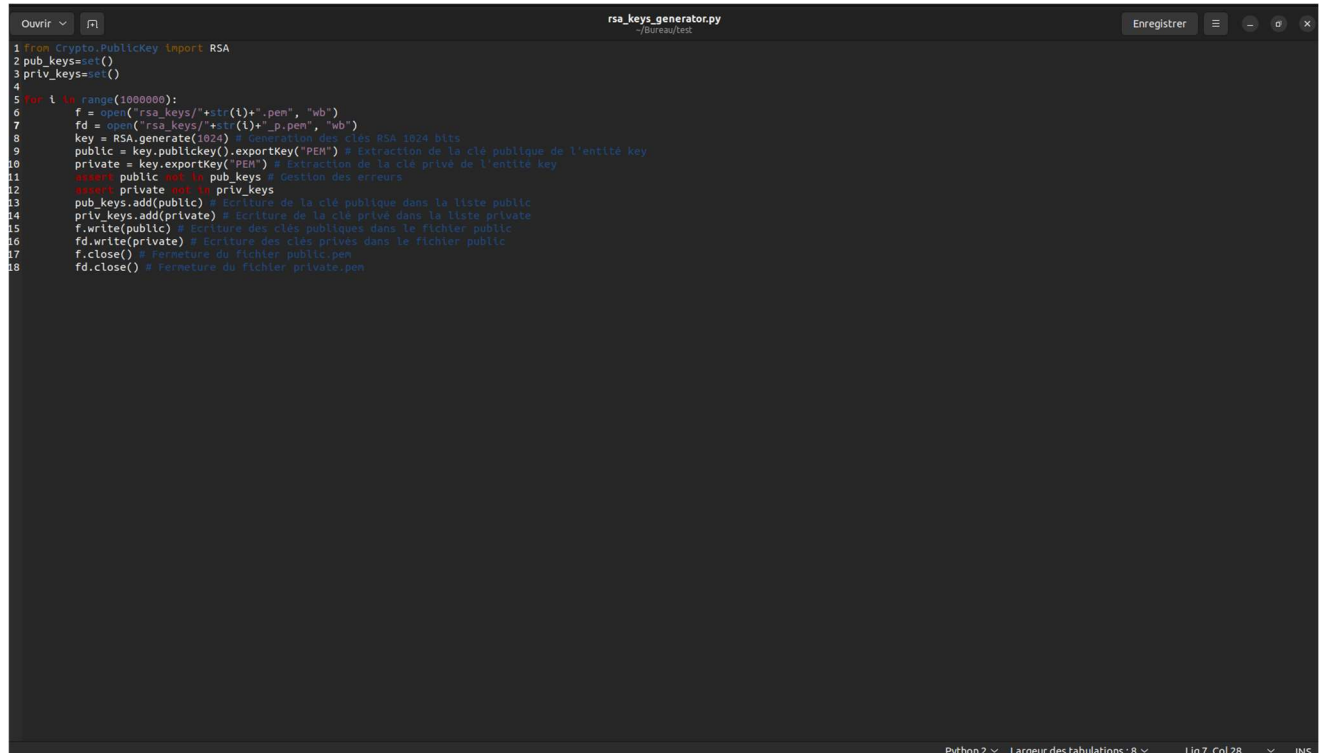
Afin de procéder à la génération des clés RSA comme demandé on utilisera la librairie python « **PyCryptodome** » :

<https://pypi.org/project/pycryptodome/>

Afin d'installer le package on utilise la commande **pip** :

**pip install pycryptodome**

Ensuite on procède à la génération des clés avec notre script :

A screenshot of a code editor window titled 'rsa\_keys\_generator.py'. The editor shows a Python script that generates RSA keys. The script imports 'RSA' from 'Crypto.PublicKey', initializes two sets for public and private keys, and then enters a loop to generate 100,000 keys. For each key, it generates a 1024-bit RSA key, extracts the public and private components, and adds them to their respective sets. The script also includes error handling for duplicate keys. The code is written in Python 2 and uses standard file operations to save the keys to 'pub\_keys.pem' and 'priv\_keys.pem'. The editor interface includes a menu bar with 'Ouvrir', 'Enregistrer', and window controls. The status bar at the bottom indicates 'Python 2', 'Largeur des tabulations : 8', 'Lig 7, Col 28', and 'INS'.

## 2. Vérifier la présence de doublons :

L'analyse de vérification des doublons permet de tester les clés définies ou sélectionnées afin de détecter la présence éventuelle de valeurs de clé primaire en double. Pour appliquer cela nous allons utiliser le programme « **rmlint** ».

Afin de l'installer nous lançons la commande :

**sudo apt install rmlint**

Nous le lancerons ensuite dans le répertoire où se trouve les clés générées :

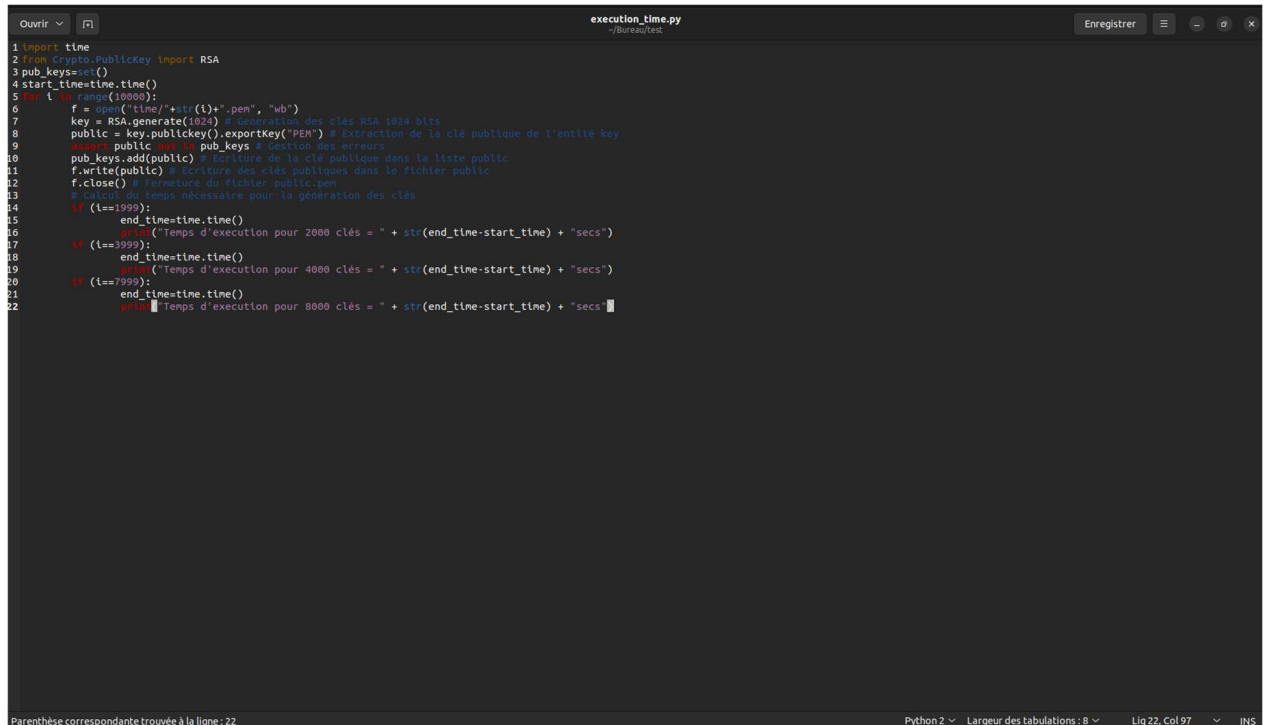
```
itcef@itcef-Lenovo-Legion-5-15IMH05H:~/Bureau/test/rsa_keys$ rmlint
==> Note: S'il vous plaît, utilisez le script enregistré ci-dessous pour l'enlèvement, pas la sortie ci-dessus.
==> 2000000 fichiers au total, dont 0 doublons répartis en 0 groupe(s).
==> Équivalent à 0 8 de doublons qui peuvent être supprimés.
==> L'examen à pris au total 1m 17,283s.

Fichier sh écrit: /home/itcef/Bureau/test/rsa_keys/rmlint.sh
Fichier json écrit: /home/itcef/Bureau/test/rsa_keys/rmlint.json
itcef@itcef-Lenovo-Legion-5-15IMH05H:~/Bureau/test/rsa_keys$
```

Résultat : Aucun doublon n'a été détecté.

### 3. Calcul du temps de génération des clés RSA :

Afin de calculer le temps de génération des clés publiques nous avons ajouté au script de génération principal quelques lignes de code :



```
Ouvrir  execution_time.py  Enregistrer  X
~/Bureau/test

1 import time
2 from Crypto.PublicKey import RSA
3 pub_keys = []
4 start_time = time.time()
5 for i in range(10000):
6     f = open("time/"+str(i)+".pem", "wb")
7     key = RSA.generate(1024) # Génération des clés RSA 1024 bits
8     public = key.publickey().exportKey("PEM") # Extraction de la clé publique de l'entité key
9     # export public not in pub_keys # Gestion des erreurs
10    pub_keys.add(public) # Ecrsure de la clé publique dans la liste public
11    f.write(public) # Ecrsure des clés publiques dans le fichier public
12    f.close() # Fermeture du fichier public.pem
13    # Calcul du temps nécessaire pour la génération des clés
14    if (i==1999):
15        end_time = time.time()
16        print("Temps d'exécution pour 2000 clés = " + str(end_time-start_time) + "secs")
17    if (i==3999):
18        end_time = time.time()
19        print("Temps d'exécution pour 4000 clés = " + str(end_time-start_time) + "secs")
20    if (i==7999):
21        end_time = time.time()
22        print("Temps d'exécution pour 8000 clés = " + str(end_time-start_time) + "secs")
```

Parentèse correspondante trouvée à la ligne: 22 Python 2 Largeur des tabulations: 8 Lig 22, Col 97 INS

Temps pour la génération de :

2000 Clés : 205 secs ==> 3.41 mins

4000 Clés : 424 secs ==> 7.06 mins

8000 Clés : 875 secs ==> 14.58 mins

Après extrapolation nous arrivons au résultat suivant :  $(102500+106000+109375)/3 = 105958$  secs ==> Il faut environ **29,43 Heures** à notre machine pour générer 1 million de clés RSA 1024 bits.

#### 4. Batch GCD :

Nous avons codé un script afin de lancer le calcul du batch GCD sur nos clés RSA.

Nous avons d'abord installé la librairie batch-gcd :

<https://libraries.io/pypi/batch-gcd>

**pip install batch-gcd==0.0.3**

```
Ouvrir  [Python icon] batch_gcd.py
~/Bureau/test

1 from Crypto.PublicKey import RSA
2 from batch_gcd import batch_gcd
3
4 gcd_list = set()
5 for i in range(1000000):
6     f = open("rsa_keys/"+str(i)+".pem", "r").read() # Ouverture des fichiers contenant les clés publiques RSA
7     key = RSA.importKey(f) # Extraction des clés des fichiers
8     gcd_list.add(key.n) # Ajout des modulus n dans la liste
9
10 g = batch_gcd(*gcd_list) # Application de l'algorithme Batch GCD sur les modulus des clés
11 print(g)
12
```

Après avoir attendu énormément de temps le résultat du calcul du Batch GCD sur les 1 million de clés RSA 1024 bits, nous avons diminué drastiquement le nombre de clés à 100000 clés.

Nous avons obtenu le résultat suivant :

[illegible]

Comme on le voit la liste résultante du Batch GCD ne contient que des 1, une amélioration du script pourrait se faire afin de n'afficher que les valeurs différentes de 1 et leur index correspondant (afin de connaître à quelle clé correspond le facteur commun trouvé).

Sur les 100000 premières clés RSA testées par Batch GCD nous pouvons dire qu'il n'y a aucun facteur commun  $(p,q)$  entre les clés RSA.

### **Conclusion :**

L'attaque Batch GCD n'a donc pas donné résultat sur cet échantillon de clés, si par chance on trouvait un nombre premier commun  $p$  ou  $q$ , à partir d'un des deux on calculerait l'autre et on pourrait reconstruire la clé privée correspondante à la clé publique, et ainsi pouvoir déchiffrer les données cryptées avec cette clé.