Projet 3- Génération de clés, VM et entropie

Table des matières

Sujet

- Dans une VM de votre choix, générer d'un million à quelques millions de clés RSA de 256, puis 512, 1024 ou 2048 bits
- Vérifier s'il y a des doublons
- Lancer batch gcd sur les autres clés, il est possible de trouver soit p soit q commun a une clé sachant que $n = pq$

Pourquoi

Nous avons choisi ce projet dans le but d'observer s'il y a une similitude entre la génération pseudo-aléatoire de plusieurs clés RSA. Dans le cas où il y aurait une similitude, nous tenterons de déterminer la source, si possible.

Définition

Pseudo-aléatoire

Dans l'informatique, il n'est pas possible d'avoir de l'aléatoire idéal, cependant, il est possible d'avoir une séquence de nombre pseudo-aléatoire, ce qui signifie que le résultat ressemble beaucoup à de l'aléatoire. En effet, en informatique, les algorithmes et les fonctions dites "aléatoires" permettent en réalité d'avoir des résultats pseudo-aléatoires car, ils utilisent soit un phénomène extérieur à l'ordinateur pour générer des nombres aléatoires (True Random Number Generators), ou soit, ils s'appuient sur des algorithmes prédéfinis pour simuler le caractère aléatoire (Pseudo Random Number Generators).

True Random Number Generator

Les nombres générés par ce type de dispositif ne sont pas prévisibles par une quelconque logique mathématique et dépendent principalement d'informations physiques comme le bruit atmosphérique. Ils sont également appelés nombres aléatoires **non déterministes** générés par le matériel.

Afin de convertir un phénomène physique en un nombre, un TRNG doit avoir plusieurs composants matériels :

- Un transducteur : il convertit le phénomène à mesurer en un signal électrique
- Un amplificateur : il augmente l'amplitude des variations aléatoires du signal afin qu'elles puissent être identifiées par le dispositif
- Un convertisseur analogique-numérique : il convertit le signal en un nombre numérique

Pseudo Random Number Generator

Les nombres aléatoires générés par ce dispositif sont prévisibles car ils utilisent des algorithmes mathématiques par le biais d'un logiciel. Ils sont également appelés nombres aléatoires déterministes. Comme le suggère le préfixe "pseudo", la séquence n'est pas réellement aléatoire, elle en a simplement l'apparence. Les deux principaux composants ce type de générateur, sont une valeur initiale ou "graine", et un algorithme préétabli. Le générateur :

1. Réception d'une valeur initiale ou une "graine" en entrée

2. Génération d'un nouveau nombre en appliquant des opérations mathématiques à la valeur d'entrée
3. Utilisation de la valeur obtenue précédemment comme valeur d'entrée pour la prochaine itération
4. Répétition du processus jusqu'à ce que la longueur souhaitée soit atteinte

Les PRNG sont **déterministes** et **périodiques**. Ils sont déterministes parce qu'une fois l'algorithme et la "graine" définis, rien ne peut changer le résultat obtenu.

OpenSSL

Par défaut, OpenSSL utilise le générateur, qui utilise la fonction de hachage MD5 comme fonction pseudo-aléatoire.

La plupart des générateurs de nombres aléatoires nécessitent une graine. Une graine est une séquence secrète et imprévisible d'octets qui est transformée puis utilisée pour définir l'état initial du générateur. La graine garantit que chaque instance unique d'un générateur produit un flux unique de bits. Deux générateurs ne devraient jamais produire la même séquence de nombres aléatoires.

Si le générateur n'est pas initialisé, OpenSSL tentera d'initialiser la valeur initiale du générateur de nombres aléatoires automatiquement lors de l'instanciation.

OpenSSL initialise la valeur initiale du générateur de nombres aléatoires en utilisant une source d'entropie spécifique au système, qui est **/dev/urandom** sur les systèmes d'exploitation de type UNIX, et qui est une combinaison de **CryptGenRandom** et d'autres sources d'entropie sous Windows.

Si le générateur de nombres aléatoires n'a pas été correctement initialisé, il affichera le message d'erreur "PRNG not seeded error".

Entropie

L'entropie désigne la mesure de "aléatoire" d'une séquence de bits. L'idéal serait d'atteindre une probabilité de 50% d'obtenir un 1, et une probabilité de 50% d'obtenir un 0, et que chaque bit soit indépendant de tous les autres bits. A l'inverse, si 1 ou les 2 conditions ne sont pas respectées, l'entropie serait définie comme faible.

Dans le cadre de la cryptographie, l'entropie est importante car elle définit le niveau de sécurité d'un système. En effet, si un système à une bonne entropie, cela signifie que son système dit pseudo-aléatoire se rapproche beaucoup de l'aléatoire idéal. De ce fait, il ne sera pas possible de "prédire" les résultats ou une partie des résultats générer par le système, et donc, il sera considéré comme étant sécuriser. A l'inverse, si un système à une mauvaise (faible) entropie, cela signifie que son système peut être partiellement ou totalement prévisible, ce qui rendrait le système vulnérable.

Démarche

Après la bonne compréhension du sujet, nous avons recherché comment générer des clés de différentes tailles. La meilleure option fut avec la commande bash, ''openssl'' qui permet de générer des clés de taille 512, 1024 et 2048. Nous nous sommes heurtés à des problèmes, la génération de clé de taille 256 bits, ce problème est lié à la sécurité de l'outil openssl. En effet, openssl ne permet pas de générer des clés de moins de 512 bits. Cette limite (logiciel) est définie par openssl qui applique par mesure de sécurité une taille de clé minimale. Cette taille minimale est définie dans le programme C de l'outil openssl. Nous avons essayé différentes solutions :

- Prendre une VM ayant une ancienne version d'openssl pour voir si cette limite n'était pas là avant. Cependant cela n'a pas résolu le problème.
- Modifier la valeur de la taille minimale de la clé dans le programme C de openssl et recompiler le programme. Nous avons tenté de mettre en place cette solution, (en installant le package développeur de openssl, ensuite aller dans les fichiers .h qui permettent de modifier une constante et enfin il faut recompiler tous le programme), pour des raisons de temps, nous avons préféré changer de solution.
- De ce fait, nous avons utilisé la librairie RSA de python qui n'était pas assujettis à cette limite logicielle. Ainsi on a pu générer des clés de différentes tailles.

Avant d'avoir trouvé la solution au problème de génération de clé de 256, nous avons décidé de lancer la génération de clés avec openssl pour éviter de perdre du temps, comme la génération de plus de 1 millions de clés prend beaucoup de temps.

Après avoir généré 1.5M des clés pour chaque taille ; 256 (python), 512, 1024 et 2048 (openssl) ; nous nous sommes dit que l'utilisation de systèmes différents pour la génération ne nous permettrait pas de bien comparer. Ainsi par manque de temps nous avons généré 10000 clés de tailles 512, 1024 et 2048 avec le programme en python. Cela nous permet alors de comparer les deux systèmes qu'on a utilisé.

Nous avons utilisé une librairie de gestion de donnée (pandas) en python pour récupérer toutes les clés pour vérifier la présence de doublons.

Ensuite nous avons appliqué le batch GCD (avec plusieurs fonctions créées en python) sur toutes ces clés et nous avons comparé la proportion de clé vulnérables (des clés ayant des facteurs communs) entre les différentes tailles de clés pour le même système de génération puis entre les clés de même taille mais avec des systèmes de génération différents.

Démarche technique :

La configuration de la VM utilisées pour la génération a pour système d'exploitation Debian (64-bit) avec 4 Go de RAM.

Pour les clés de 256, Rehan a utilisé 6 CPU car c'est beaucoup plus rapide et son PC avait les performances suffisantes.
Pour les clés 512, 1024 et 2048 (avec python et openssl), Théo a utilisé 2 CPU qui cela est la performance maximale du PC utilisé.

Nous avons rédigé 3 scripts Bash "script_2048", "script_1024", "script_512" dans lequel on utilise l'outil **openssl** afin de générer des millions de clés RSA de différentes tailles.

On a alors trois fichiers :

« save_pub_key_512.txt » : 1 500 000 clés RSA de taille 512 qui ont mis 8h50 pour être générées.
« save_pub_key_1024.txt » : 1 500 000 clés RSA de taille 1024 qui ont mis environ 28h20 pour être générées.
« save_pub_key_2048.txt » : 1 571 161 clés RSA de taille 2048 qui ont mis 74h32 pour être générées.

Ainsi nous avons utilisé un deuxième système de génération (**libraire RSA de python**) et nous avons trois programmes ; "gen_rsa_keys_256.py", "gen_rsa_keys_512.py", "gen_rsa_keys_1024.py" "gen_rsa_keys_2048.py" ; pour générer des clés RSA (de 256, 512, 1024 et 2048 bits).

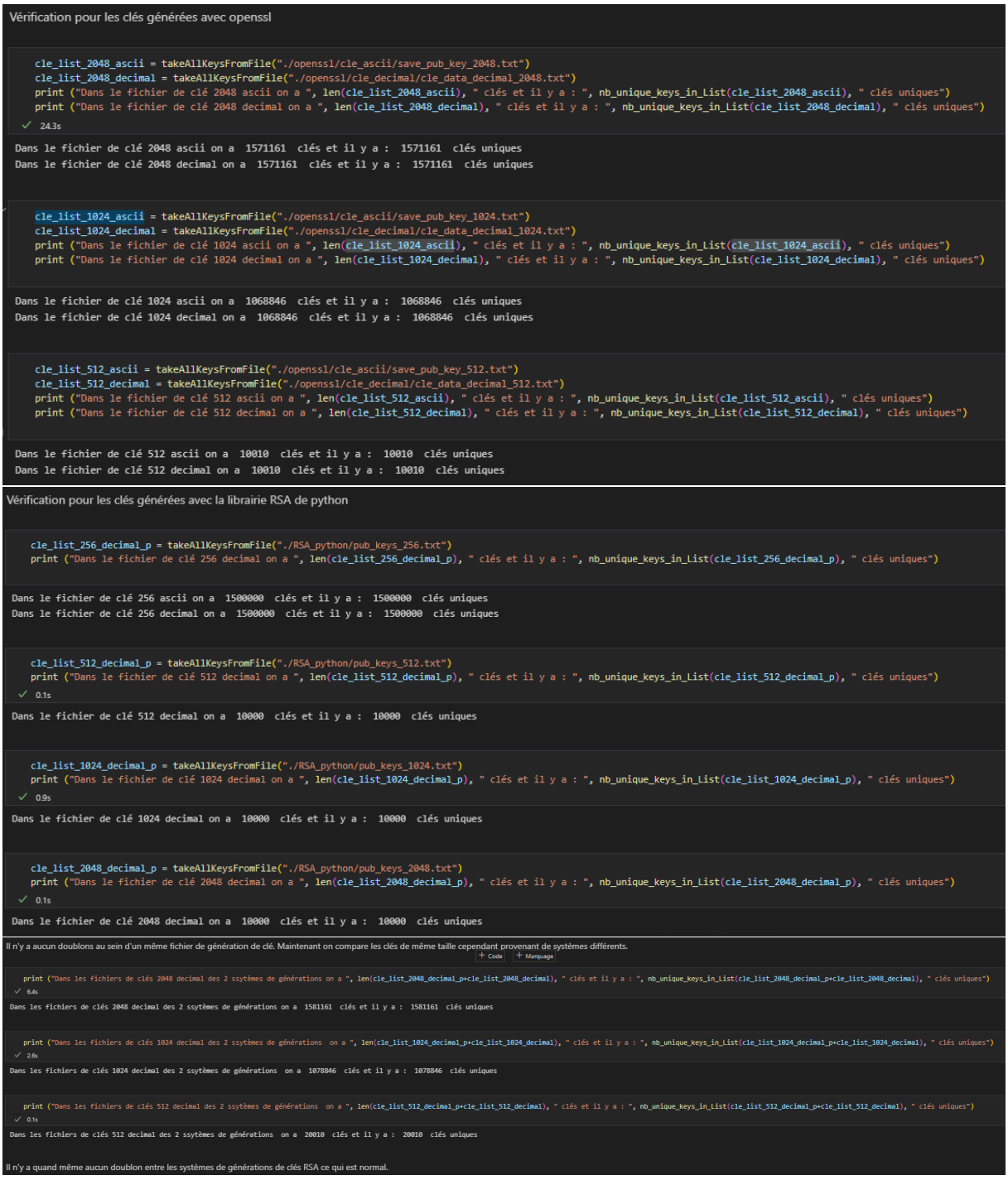
« pub_keys_256.txt » : 1 500 000 clés RSA de taille 256 qui ont mis environ 3h10 pour être générées.
« pub_keys_512.txt » : 10 000 clés RSA de taille 512 qui ont mis 9 min pour être générées.
« pub_keys_1024.txt » : 10 000 clés RSA de taille 1024 qui ont mis 2h08 pour être générées.
« pub_keys_2048.txt » : 10 000 clés RSA de taille 2048 qui ont mis environ 12h48 pour être générées.

Nous voyons que peu importe le système utilisé le temps de génération augmente drastiquement en fonction de la taille de la clé (et aussi en fonction du nombre clés générées logique).
Il y a d'autres facteurs sur le temps de génération, le système de génération utilisé, les spécifications de la machine qui génère les clés.

Résultat & Analyse

Doublon

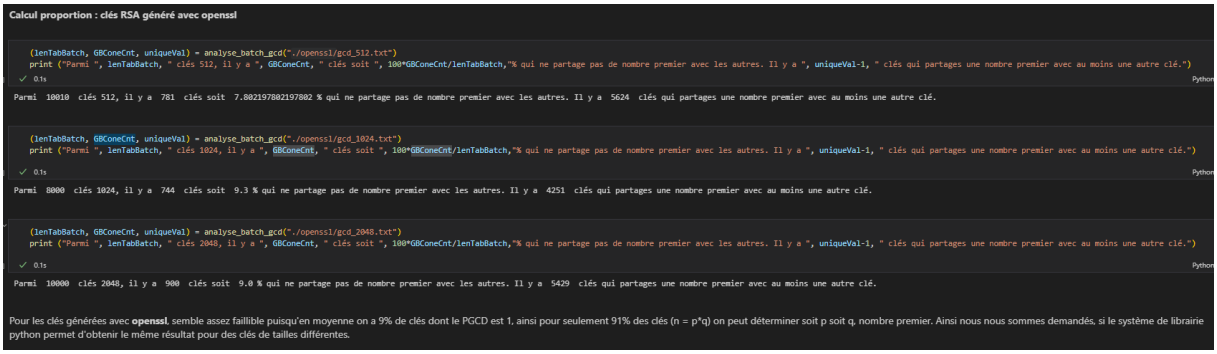
Nous n'avons pas trouvé de clé commune en comparant les millions de clé RSA de même taille que nous avons générés. Nous avons alors vérifié l'absence de doublon entre les clés des 2 systèmes générations différents et nous avons obtenu le même résultat, il n'y a pas de doublon. Nous pouvons donc déduire que la génération de clé RSA dite « pseudo-aléatoire » est, à ce jour, une méthode de sécurité robuste.



Partie résultat vérification de la présence de doublon du fichier « Crypto_projet_3_RSA.ipynb »

Batch GCD

Le résultat du batch GCD donne une liste contenant tous les PCGD calculés entre toutes les clés de la même taille. On regarde alors le nombre de 1 ce qui signifie qu'on n'a pas pu factoriser les clés.



Partie résultat, Calcul proportion : clés RSA généré avec openssl du fichier « Crypto_projet_3_RSA.ipynb »

Pour les clés 512 sur un échantillon de 10010 clés,
- 781 fois '1'
- 5624 PGCD différents

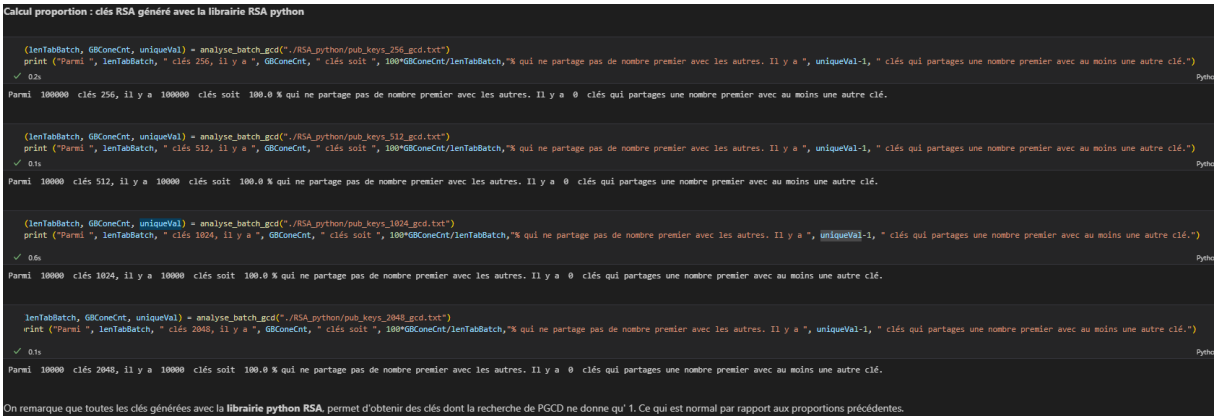
On a alors une proportion de 7.8% de clés dont on ne peut pas trouver de PGCD.

Pour les clés 1024, sur un échantillon de 8000 clés,
- 744 fois '1'
- 4251 PGCD différents

On a alors une proportion de 9.3% de clés dont on ne peut pas trouver de PGCD.

Pour les clés 2048 sur un échantillon de 10000 clés,
- 900 fois '1'
- 5429 valeurs PGCD différents

On a alors une proportion de 9% de clés dont on ne peut pas trouver de PGCD. On remarque qu'on a environ 9% de clés étant non-vulnérable (non factorisable) car avec le batch GCD on n'a pu trouver soit p soit q. Ainsi il y a 91% de clés faillibles cependant cela semble bizarre comme résultat puisque ces clés sont utilisées pour sécuriser des systèmes de communication.



Partie résultat, Calcul proportion : clés RSA généré avec la librairie RSA de python du fichier « Crypto_projet_3_RSA.ipynb »

Pour les clés 256 sur un échantillon de 100 000 clés,
- 100 000 fois ‘1’
- 1 PGCD différents

Pour les clés 512,1024 et 2048 sur un échantillon de 10 000 clés,
- 10 000 fois ‘1’
- 1 PGCD différents

Le résultat est explicite, on ne peut pas trouver de p ou q avec des clés générées en python RSA, en tout cas pas avec ce batch GCD. Quelque soit la taille de la clé, la factorisation de ces clés RSA n’est pas réalisable ce qui est plutôt attendu.

Documentation :

Tous les documents ne pouvaient pas être envoyé sur moodle dû aux fichiers volumineux contenants les clés RSA. Ainsi vous pouvez tout télécharger sur gitlab.

Nous avons sauvegardé tous nos documents sur gitlab : M. LARINIER a été ajouté en tant que "Maintainer".

https://gitlab.esiea.fr/akoka/crypto_projet_3_rsa_analyse.git

Ainsi vous retrouverez tous nos programmes ici ainsi que nos fichiers de résultats.

Bibliographie :

BATCH GCD PYTHON

https://github.com/fionn/batch-gcd/blob/master/batch_gcd/__init__.py

https://facthacks.cr.yp.to/product.html

https://facthacks.cr.yp.to/remainder.html

https://facthacks.cr.yp.to/batchgcd.html

Random number

https://wiki.openssl.org/index.php/Random_Numbers#:~:text=OpenSSL%20provides%20a%20number%20of,%2C%20ANSI%20X9%20committee%20(X9

https://levelup.gitconnected.com/how-do-computers-generate-random-numbers-a72be65877f6

https://www.techno-science.net/definition/6162.html

https://electricalfundablog.com/random-number-generator-types-works-architecture/

Analysis for Anomalies in Cryptographic RNG and Industrial Applications :

https://www.theses.fr/2022GRALM013.pdf