Projet de cryptographie n°3

Badea-Guéritée Maximilien
Tougard Enzo
06/09/2022 au 06/11/2022

Introduction :

Le sujet que nous avons choisi est celui sur la génération de clés RSA, VM et entropie. Ce sujet consiste à créer un programme nous permettant de générer un nombre important de clés RSA, afin de pouvoir ensuite les comparer entre elles et ainsi voir si obtenir un double d'une ou plusieurs clés est possible. Il faut ensuite utiliser un algorithme nommé Batch GCD afin de trouver soit p ou q commun à une clé entre elles.

Trouver un p ou q commun est important pour ce projet car nous savons que
n = p * q avec p et q étant générés de manière "aléatoire" cela voudrait idéalement dire que l'on ne pourrait pas avoir deux clés possédant le même p ou q or il n'est pas impossible que si l'on possède un nombre assez important de clés nous obtenions deux clés possédant le même p ou q du fait que l'aléatoire que nous utilisons n'est pas totalement aléatoire ou aussi si le générateur de nombres utilisé pour obtenir p et q est mal configuré. Trouvé alors plusieurs clés possédant le même p où q est alors possible et nous permet d'utiliser l'algorithme batch GCD pour trouver un nombre premier étant le plus grand diviseur commun entre ces clés.

Entropie et aléatoire :

L'entropie est un terme désignant la quantité de désordre d'un système. En informatique lorsque nous avons souvent besoin de générer des nombres aléatoires comme ici dans notre projet ou p et q doivent être générés de manière aléatoire or infortunément l'aléatoire en informatique ne l'est pas réellement, mais est le résultat d'une multitude de calculs en utilisant plusieurs sources tels que la date du système, l'état des disques durs ou encore la saisie clavier pour obtenir des informations et ainsi utiliser les valeurs récolter pour créer un nombre aléatoire.

Plus nous utilisons de source pour générer un nombre de manière "aléatoire" plus nous pourrons dire que l'entropie sera élevée. Malgré cela l'entropie possède une limite plus ou moins grande selon le système que l'on utilise car le système ne possède qu'un nombre limité de données bien que ce nombre de données puisse être conséquent il est possible de voir le nombre de sources d'entropie disponibles sur un système en utilisant la commande cat /proc/sys/kernel/random/entropy_avail sur Linux .

```
enzo@enzo-virtual-machine:/$ cat /proc/sys/kernel/random/entropy_avail
256
```

Démarche :

En premier lieu pour ce projet il nous fallait avoir des clés RSA. Pour obtenir des clés RSA il existe plusieurs solutions telles que le logiciel PuTTYgen, nous pouvons aussi utiliser ssh-keygen -t rsa dans le terminal pour générer ou encore nous avions la possibilité d'utiliser openssl avec la commande openssl -genrsa [taille de la clé]. Pour notre projet nous avons décidé d'utiliser openssl.

Voici un exemple de clé générée :

```
enzo@enzo-virtual-machine:~/Desktop$ openssl genrsa 512
-----BEGIN PRIVATE KEY-----
MIIBUwIBADANBgkqhkiG9w0BAQEFAASCAT0wggE5AgEAAkEAwaWzFNoTA+qLhWT6
uyhRuCE8Gujj//LoXWK26XVgTUZjFZ5HfSAQZS/Bdp/YLIzXPxfebD9wwrvmAC0W
rC+FQQIDAQABAkBrUIeo8518GxwM5cQpuhNnhkuH4i6QZi4tqooRYPHogDWLNUwK
soYffFz1hdE8jsTo6cvfaDYpihIELC5pehwBAiEA478ovsFYvL9VHVq/5UWE1vSo
YMw5HJXKwqKNIg/fsqECIQDZq5nVObvqjzeossftwRo39pgHbo6DBcyUdaqYSF9u
oQIgS/Vbv5N45yrhVnMIAY3YKmFbLz2t5qzG/Xd9ntiwgUECIEiUxNXpy3RaTyfs
fWgVFMYGbwlDXzfnTiFO3xSqRVAhAiBq8F/uzf2fLp3W0GEGNGfPN9WrHvvIiOC3
rQXKQR4Whg==
-----END PRIVATE KEY-----
```

Mais une clé toute seule n'est pas suffisante de ce fait il nous a fallu trouver un moyen d'utiliser cette commande depuis notre programme et de récupérer la sortie de la commande.

Nous avons donc utilisé : `key = os.popen('openssl genrsa 512').read()`. Cette commande permet de générer une clé rsa de 512 bits, pour obtenir une clé plus petite il suffit de changer le 512 en 256 ou si l'on en souhaite une plus grande en 1024 ou encore 2048.

La clé est donc à ce moment stockée dans la variable key, et nous avons ensuite sauvegardé cette clé dans un tableau `Clef.append(key)` avant de générer une nouvelle clé et ainsi répéter le même processus jusqu'à obtenir le nombre de clés que nous souhaitions.

Voici le code utiliser pour générer les clefs et vérifier s'il y a des doublons :

```python
for i in range(10): #boucle permettant de choisir le nombre de clés à crées
    tailleClef = len(Clef)
    key = os.popen('openssl genrsa 512').read() # génération et stockage d'une clé dans la variable key
    token = 0 # cette variable permets de savoir quand nous avons eu un doublon et si la clé actuel est la première génére
    if tailleClef == 0: # stockage de la première clé car le tableau est actuellement de taille 0
        Clef.append(key)
        tailleClef = len(Clef)
        token = 1 # empêche de stocker deux fois la première clef
    for j in range(tailleClef): #boucle permettant de parcourir tout le tableau Clef
        if key == Clef[j] and j != 0: #permet de voir si la clé est un doublon
            NbDoublon +=1
            Doublon.append(key)
            NbDoublon = len(Doublon)
            Clef.append(key)
            tailleClef = len(Clef) #actualisation de la valeur permettant de connaitre la taille du tableau Clef
            token = 1
    if token == 0: # stockage normal de la clef si elle n'est pas un doublon
        Clef.append(key)
        tailleClef = len(Clef)
    print(Clef[i])
```

Suite à la génération et vérification de la présence de doublons nous avons créé une fonction permettant de dire si des doublons ont été trouvés et si oui de les afficher.

```python
def affiche_doublon(): # dit si nous avons trouver un doublon ou pas
    if NbDoublon == 0:
        print("Pas de doublon")
    if NbDoublon !=0:
        print('Doublon Trouvé')
        for k in range(NbDoublon): # si un ou plusieurs doublon ont étaient trouver permets de les afficher
            print(Doublon[k])
```

Ensuite il nous a fallu changer le format de nos clés en enlevant le balisage de début et fin de clé.

Exemple de format de clé avec balisage :

```
-----BEGIN PRIVATE KEY-----
MIIBVgIBADANBgkqhkiG9w0BAQEFAASCAUAwggE8AgEAAkEAyAiOMnIawJ/1n7VL
byT6bG/3T79Vsm+ErM4macndq8K/rypq+9JHAn5bnjOg9yBJ3DJoMWzgit2qxMVP
Ye7k2wIDAQABAkBd+x8L9m45SC43zg7V9lKt9eEFF8kkn55/LhAALfa3BmEy+Sme
RbfsjztzdPfGrFINorlogpHZPa8rg3g9NmwRAiEA4qAZMopVcFqhHoWtKvQc/G5E
s8A3KWCpRdcOAGus0XkCIQDh9hSQ72b0QG1OO9kzflXdKgutSCWGYRLLIePfycfH
8wIhANr3SaFVm7vVNXviOTEZcLBOcr8i56lpJeS/J14sq2wxAiEAn+dvAmoIp4z4
TUbZHDGpPcvFW6kUtRX5ILso8XNv57UCIQCi5bdLsPJJZoy5hQ8MYLPwHMUTdmkO
mMw0SAx2YFgm4w==
-----END PRIVATE KEY-----
```

Exemple de format de clé sans balisage:

```
MIIBVgIBADANBgkqhkiG9w0BAQEFAASCAUAwggE8AgEAAkEAyAiOMnIawJ/1n7VL
byT6bG/3T79Vsm+ErM4macndq8K/rypq+9JHAn5bnjOg9yBJ3DJoMWzgit2qxMVP
Ye7k2wIDAQABAkBd+x8L9m45SC43zg7V9lKt9eEFF8kkn55/LhAALfa3BmEy+Sme
RbfsjztzdPfGrFINorlogpHZPa8rg3g9NmwRAiEA4qAZMopVcFqhHoWtKvQc/G5E
s8A3KWCpRdcOAGus0XkCIQDh9hSQ72b0QG1OO9kzflXdKgutSCWGYRLLIePfycfH
8wIhANr3SaFVm7vVNXviOTEZcLBOcr8i56lpJeS/J14sq2wxAiEAn+dvAmoIp4z4
TUbZHDGpPcvFW6kUtRX5ILso8XNv57UCIQCi5bdLsPJJZoy5hQ8MYLPwHMUTdmkO
mMw0SAx2YFgm4w==
```

Une fois le format de clé change nous pouvons alors convertir la clé. À ce moment dans le programme est stocké dans un tableau sous forme de chaîne de caractères nous avons alors converti la clé au format de base64 puis de base64 en hexadécimal et finalement l'hexadécimal en int.

Voici le code utilisé pour le formatage et la conversion :

```
26  def Format_Clef():
27      for g in range(tailleClef):
28          NEW_key = Clef[g].replace("-----BEGIN PRIVATE KEY-----", "") #cette ligne et la suivante permette d'enlever deux lignes dans la clef
29          NEW_key = NEW_key.replace("-----END PRIVATE KEY-----", "")   #pour permettre ensuite de convertir la clef
30          Clef[g] = NEW_key # change la clef la ou elle est stocker
31          print(Clef[g])
32          #Clef[g] = int.from_bytes((base64.b64decode(Clef[g])), byteorder="big",)
33          Clef[g] = int(base64.b64decode(Clef[g]).hex(),base = 16)#nous permets de convertir la clef de string en base64
34          #puis de base64 en hexadecimal et finalement d'hexadecimal en int
35          Clef_Int.append(Clef[g]) # stockage de la nouvelle valeur de la clef
```

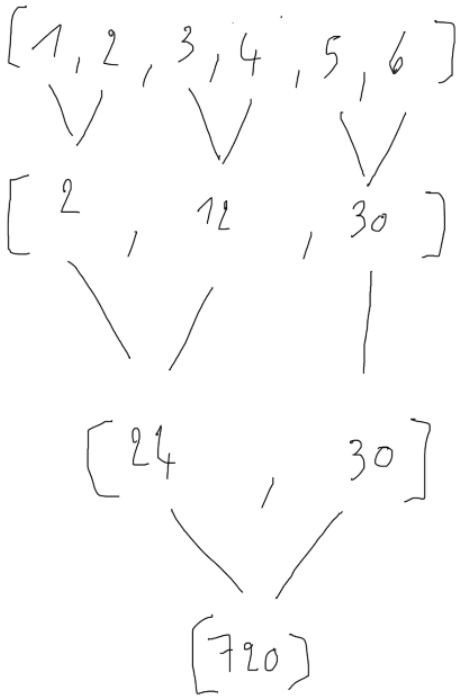
Une fois le format changé et la conversion effectuée voici à quoi ressemble une clé :

```
13193497510381761293788346135367768219866795008526384877294943291709677615566767123475840928923254644853668348480898078448854715986848433099250357891079708758045732041085113097351261128340510062848203458098260368169758839360456920571552478435539288217517924804600292715257155988482417405004020282248337127864778356933838045687107180429475976281883314034051418040427203621332851187774561805799612051913879067794940676040513611137031085772457013034519888944127802920711218209642603129351080567135044147282494487986403689778289710115897039082214325320281040449670224267842788325337308817929592964475323125137981422049465779944237853826980899418421913552192010820211082205931592335990584050186543051292825904897545650771399305482457192441064415152624176267234415242153523239055208394109373685935841253424204427083674
```

Une fois que la clé a changé de format et a été convertie en int, nous avons pu utiliser l’algorithme de BatchGCD sur la clé.

L’algorithme commence par utiliser un arbre de produit.
L’arbre de produit va multiplier les éléments entre eux deux par deux jusqu'à diviser le nombre d’éléments par deux puis répéter le processus jusqu'à ce qu’il ne reste plus qu’un seul nombre.

Exemple d'exécution d’un arbre de produit:



Ensuite faire le carré des nœuds de cet arbre de produit avant de finalement faire le gcd.

Résultat de la génération de clé :

Le temps pris pour générer un certain nombre de clés dépendra beaucoup des ressources allouées à la vm sur laquelle on travaille et du nombre de processus qui tourne en arrière-plan sur la vm aussi, nous avons constaté grâce à la commande top que la génération de clés prenait entre 95 et 100 % du processeur lorsqu’on les fait tourner.

Une autre notion intéressante par rapport à la génération des clés et que la probabilité d’obtenir des doublons augmentera plus on aura généré de clé mais elle sera moindre si on en génère pas beaucoup plusieurs fois à la suite mais cela augmentera largement le temps pris pour la génération des clés avec les différentes vérifications entre toutes les clés

-Génération de clé simple non extrapolée:

Taille de clé	2000	4000	8000	10000	100000	500 000	1 millions
256	41 second	1 min 24 sec	2 min 55	n/a	n/a	n/a	n/a
512	15 sec	29 sec	57 sec	1 min 20 sec	13 min 44sec	1h16 min 25sec	2h44min
1024	24 sec	49 sec	1 min 38 sec	2 min 12 sec	23 min 40 sec	n/a	n/a
2048	5 min 58 sec	12 min 7sec	19 min 16 sec	n/a	n/a	n/a	n/a

De ces résultats nous pouvons déterminer une moyenne de temps nécessaire à la génération d'une clé.

Pour des clés de 256 : ((41/2000) + (84/4000) + (175/8000))/3 = 0.021125s
Pour des clés de 512 : ((15/2000) + (29/4000) + (57/8000))/3 = 0.007291s
Pour des clés de 1024 : ((24/2000) + (49/4000) + (98/8000))/3 = 0.01216s
Pour des clés de 2048:((358/2000)+(727/4000) + (1156/8000))/3 = 0.1684166667s

Extrapolation pour 1 millions de clé:
Pour des clés de 256:(((0.021125 * 1000000)/60)/60) = 5,868055556 heures
Pour des clés de 512:(((0.007291 * 1000000)/60)/60) = 2.025277778 heures
Pour des clés de 1024: (((0.01216 * 1000000)/60)/60) = 3.377777778 heures
Pour de clés de 2048 :(((0.1684166667 * 1000000)/60)/60) = 46.78240741 heures

-Génération de clé avec vérification de doublons

Taille de clé	2000	4000	8000	10000	100 000
256	42 sec	1 min 27 sec	2 min 59 sec	n/a	n/a
512	16 sec	31 sec	1 min 2 sec	1 min 22s	15 min 48 sec
1024	25 sec	51 sec	1 min 42 sec	2 min 56 sec	29 min 34 sec

De ces résultats nous pouvons déterminer une moyenne de temps nécessaire à la génération d'une clé.

Pour des clés de 256 : ((42/2000) + (87/4000) + (179/8000))/3 = 0.02170s
Pour des clés de 512 : ((16/2000) + (31/4000) + (62/8000))/3 = 0.00783s
Pour des clés de 1024 : ((25/2000) + (51/4000) + (102/8000))/3 = 0.01266s

-Génération de clé avec vérification des doublons et batchgcd

Taille de clé	2000	4000	8000
256	1 min 8 sec	4 min 39 sec	18 min 39 sec
512	1 min 38 sec	5 min 52 sec	22 min 34 sec
1024	6 min 37 sec	20 min 22 sec	1h 07 min 42 sec

De ces résultats nous pouvons déterminer une moyenne de temps nécessaire à la génération d'une clé.

Pour des clés de 256 : ((68/2000) + (279/4000) + (1119/8000))/3 = 0.08120s
Pour des clés de 512 : ((98/2000) + (352/4000) + (1354/8000))/3 = 0.10208s
Pour des clés de 1024 : ((397/2000) + (1222/4000) + (4062/8000))/3 = 0.33725s

Grâce à ces résultats nous pouvons déduire que le temps pour générer un certain nombre de clés augmente en fonction de plusieurs paramètres tels que la taille d'une clé, si la clé est stockée, si nous vérifions s'il existe des doublons et finalement si nous utilisons batch GCD.

Problème rencontrés et solutions :

Lors de ce projet nous avons rencontré plusieurs problèmes.

Le premier problème que nous avons rencontré s'est avéré être la génération de clé rsa 256 bits malgré le fait que la génération de clés de 512,1024 et 2048 bits fonctionnait. La solution que nous avons trouvée a été de changer de version d'Ubuntu pour la 14.04.

Le deuxième problème que nous avons eu a été la conversion de la clé de string à int car lorsque nous utilisation la conversion de base64 à int avec la commande :

```
Clef[g] = int.from_bytes((base64.b64decode(Clef[g])), sys.byteorder)
```

La valeur ainsi obtenue n'était pas correct il nous a fallu d'abord convertir la clé de base64 en hexadécimal puis l'hexadécimal en int a l'aide de la commande :

```
Clef[g] = int(base64.b64decode(Clef[g]).hex(),base = 16)
```

Le dernier problème que nous avons rencontré a été sur l'algorithme de Batch GCD car les valeurs retourner ne corresponde pas à celle attendu.

Conclusion :

Le matériel a une importance capitale dans la génération de clés ce qui voudrait dire que lorsque les ordinateurs quantiques seront créé il n'y aura plus aucune sécurité possible car ce qui freine les personnes malveillantes est le nombre de clés qu'il faut générer pour trouver ne serait-ce qu'un doublon, ce qui tourne autour du milliard de clés.
Surtout que le temps pour les générer s'allonge par rapport à la taille de la clé ce qui explique le fait que les clés de 256 bytes ne soient plus utilisées et que les clés de 512 ne sont plus trop utilisées car elles ne demandent pas beaucoup de temps avec des bons ordinateurs actuellement.

Librairie :

Pour le code nous avons utilisé les librairies:
-os
-base64
-sys
-numpy
-math
-collections.abc
-openssl

Documentation :

/index.html (openssl.org)
os — Miscellaneous operating system interfaces — Python 3.11.0 documentation
base64 — Base16, Base32, Base64, Base85 Data Encodings — Python 3.11.0 documentation
sys — System-specific parameters and functions — Python 3.11.0 documentation
collections.abc — Abstract Base Classes for Containers — Python 3.11.0 documentation
https://libraries.io/pypi/batch-gcd
https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_quick_guide.htm
https://stackoverflow.com/questions/13262125/how-to-convert-from-base64-to-string-python-3-2
https://stackoverflow.com/questions/24608302/the-integer-part-of-rsa-public-key
https://research.kudelskisecurity.com/2018/08/16/breaking-and-reaping-keys-updated-slides-and-resources/
https://github.com/StanGirard/HackRSA/blob/master/src/MultiplyCerts.ipynb

Annexes

Code:

```
import os
import base64
import sys
import numpy as np
import math
from math import prod, floor, gcd

clear = 'clear'
Clef = []
Clef_Int = []
Doublon = []
NbDoublon = len(Doublon)
resultat_gcd = []
i = 0
j = 0

def affiche_doublon(): # dit si nous avons trouver un doublon ou pas
    if NbDoublon == 0:
        print("Pas de doublon")
    if NbDoublon !=0:
        print('Doublon Trouvé')
        for k in range(NbDoublon): # si un ou plusieurs doublon ont étaient trouver permets de les afficher
            print(Doublon[k])

def Format_Clef():
    for g in range(tailleClef):
            NEW_key = Clef[g].replace("-----BEGIN PRIVATE KEY-----", "") #cette ligne et la suivante permette d'enlever deux lignes dans la clef pour permettre ensuite de convertir la clef
        NEW_key = NEW_key.replace("-----END PRIVATE KEY-----", "")
        Clef[g] = NEW_key # change la clef la ou elle est stocker
        print(Clef[g])
        #Clef[g] = int.from_bytes((base64.b64decode(Clef[g])), byteorder="big",)
```

```
            Clef[g] = int(base64.b64decode(Clef[g]).hex(),base = 16)#nous permets de convertir la clef de string en base64 puis de base64 en hexadecimal et finalement d'hexadécimal en int
        Clef_Int.append(Clef[g]) # stockage de la nouvelle valeur de la clef

def producttree(X): # cette fonction permets de multiplier par pairs et ensuite les pairs obtenue entre elles
    result = [X]
    while len(X) > 1:
        X = [prod(X[int(i*2): int((i+1)*2)]) for i in range((len(X)+1)//2)]
        result.append(X)
    return result

def batchgcd_faster(X): #effectuer le carre du noeud actuelle du productree actuel
    prods = producttree(X)
    R = prods.pop()
    while prods:
        X = prods.pop()
        R = [R[floor(i/2)] % X[i]**2 for i in range(len(X))]
    return [gcd(r//n,n) for r,n in zip(R,X)]

file_out = open("rsa_key.txt", "w")
for i in range(250000): #boucle permettant de choisir le nombre de clés à crées
    tailleClef = len(Clef)
     key = os.popen('openssl genrsa 512').read()# génération et stockage d'une clé dans la variable key
    file_out.write(key)
     token = 0 # cette variable permets de savoir quand nous avons eu un doublon et si la clé actuel est la première génére
      if tailleClef == 0: # stockage de la première clé car le tableau est actuellement de taille 0
        Clef.append(key)
        tailleClef = len(Clef)
        token = 1 # empêche de stocker deux fois la première clef
    for j in range(tailleClef): #boucle permettant de parcourir tout le tableau Clef
        if key == Clef[j] and j != 0: #permet de voir si la clé est un doublon
            NbDoublon +=1
            Doublon.append(key)
            NbDoublon = len(Doublon)
            Clef.append(key)
```

```
            tailleClef = len(Clef) #actualisation de la valeur permettant de connaitre la taille du tableau Clef
            token = 1
    if token == 0: # stockage normal de la clef si elle n'est pas un doublon
        Clef.append(key)
        tailleClef = len(Clef)
    print(Clef[i])

#os.system(clear)

affiche_doublon()
Format_Clef() # permet de changer le format des clefs et ensuite de les convertir de string à int
resultat_gcd = batchgcd_faster(Clef_Int) #appelle de fct permettant d'effectuer le batchGCD
print(resultat_gcd)
file_out.close()
```