

# La cryptographie RSA

Présenté par :

**Zineb Moufakkir**

Encadré par :

**Mr. Manessouri Abdelali**

# Sommaire

1. Introduction
2. Présentation du cryptage RSA :
  - Génération des clés
  - Description du Protocol
  - Démonstration mathématiques
3. Exemple d'utilisation de RSA
4. les contraintes du chiffrement RSA
5. Attaques sur RSA :
  - Cryptanalyse du RSA en connaissant  $\phi(n)$
  - Cryptanalyse de RSA lorsque  $m < n^{(1/e)}$
  - Cryptanalyse RSA ( même exposant et différents modules)
  - Cryptanalyse RSA ( même module et différents exposants)
6. Programme attaquant le RSA



# *Introduction :*

Depuis que les gens ont commencé à écrire les événements de leur vie, il y a eu un besoin de cryptographie. La cryptographie consiste à coder le texte de telle manière que les étrangers au code ne peuvent pas le comprendre, mais seul le lecteur souhaité est capable de décrypter le message. Surtout, en temps de guerre, il est essentiel que l'ennemi ne sache pas ce que vous et vos alliés complotez, car gagner ou perdre une guerre peut dépendre du secret des opérations afin de surprendre l'ennemi. Cependant, il y avait une mise en garde à tous les crypto systèmes avant RSA : ils étaient tous basés sur le fait que les deux parties du décodage et de l'encodage devaient connaître la méthode de cryptage et la clé pour décrypter le chiffre. En vérité, ce problème de distribution de clé est le même problème que les gens essayaient de résoudre lorsqu'ils ont inventé la cryptographie : les clés et la méthode de décryptage doivent être relayées à tous les lecteurs souhaités du message, mais comment relayer la clé en toute sécurité ?

## Clé asymétrique

Le problème a été résolu par Whitfield Diffie, en collaboration avec Martin Hellman. Diffie avait découvert un type de chiffrement révolutionnaire : son chiffrement incorporait une **clé asymétrique**. Dans tous les autres crypto systèmes, le déchiffrement est tout simplement le contraire du chiffrement ; ces systèmes utilisent une **clé symétrique**, car le déchiffrement et le chiffrement sont symétriques. Dans un chiffrement asymétrique, il existe deux clés distinctes : **les clés publiques et privées**. Bien que Diffie ait conçu un concept général de chiffrement asymétrique, il n'avait en fait pas de fonction unidirectionnelle spécifique répondant à ses besoins. Cependant, son article (publié en 1975) a montré qu'il existait effectivement une solution à la distribution des clés et il a suscité l'intérêt d'autres mathématiciens et scientifiques. Malgré tous ses efforts, Diffie et ses partenaires Hellman et Merkle n'ont pas pu découvrir un tel chiffre. Cette découverte a été faite par un autre trio de chercheurs : Rivest, Shamir et Adleman.

## L'équipe parfaite :

Rivest, Shamir et Adleman formaient une équipe parfaite;

-Rivest , Shamir : informaticiens .

-Adleman : mathématicien



Rivest et Shamir ont passé un an à trouver des idées, et Adleman a passé un an à les abattre avec des détection de failles.

*« En avril 1977, Rivest, Shamir et Adleman ont passé la Pâque chez un étudiant et ont consommé des quantités généreuses de vin avant de retourner dans leurs maisons respectives vers minuit. Rivest était incapable de dormir, alors il s'est allongé sur son canapé avec un manuel de mathématiques. Il se mit à réfléchir à la question qui le taraudait toute l'année : est-il possible de trouver une fonction unidirectionnelle qui ne peut être inversée que si le récepteur a des informations particulières ? Soudain, les brumes ont commencé à se dissiper et il a eu une révélation. Il passa le reste de la nuit à formaliser son idée, et à l'aube, il avait effectivement écrit un article mathématique complet. Rivest a fait une percée, mais elle n'aurait pas pu se produire sans l'aide de Shamir et Adleman. Le système a ensuite été surnommé RSA, pour Rivest, Shamir et Adleman » (Traduction)*

# Présentation du cryptage RSA :

## Génération des clés :

Le RSA fonctionne à partir de deux nombres premiers, que l'on appellera  $p$  et  $q$ . Ces deux nombres doivent être très grands, car ils sont la clé de voûte de notre cryptage. Aujourd'hui, on utilise des clés de 128 à 1024 bits, ce qui représente des nombres décimaux allant de 38 à 308 chiffres !

### les étapes de génération des clés :

**Étape1:** Sélectionnez deux grands nombres premiers  $p$  et  $q$ .

**Étape2:** Calculez la valeur de  $n=p*q$

**Étape 3:** Calculer la fonction (fonction d'Euler)  $\phi(n) = (p-1)*(q-1)$ .

**Étape 4:** Sélectionnez n'importe quelle valeur de  $e$ , comprise entre 1 et  $(n)$ . tel que  $\text{PGCD}(e, \phi(n))=1$ .

**Étape 5:** Calculez la valeur de  $d$  telle que  $d*e \equiv 1 \pmod{\phi(n)}$  (l'inverse de  $e$  modulo  $\phi(n)$ )

On tire ainsi :

Clé publique : (n , e)

Clé privée : d

### Cryptage :

Pour crypter un message m il suffit de le mettre à la puissance de e :

$$M \equiv m^e \pmod{n}$$

### Décryptage:

Pour décrypter le message M il suffit de le mettre à la puissance d :

$$m \equiv M^d \pmod{n}$$

## Description du Protocole :

Le but de RSA est bien sûr de pouvoir transmettre un message codé, que seul le récepteur "officiel" puisse décrypter, c'est-à-dire qui ne puisse pas être décrypté par un tiers qui intercepterait le message. Nous appellerons Zineb la destinataire du message, et Othmane l'émettrice.

### Algorithme RSA:

**Input:** Othmane transmet un message  $m$

**Output:** Zineb reçoit le même message  $m$

**Begin**

Othmane calcule  $m^e \equiv M \pmod{n}$   
Othmane envoie  $M$  à Zineb  
Zineb calcule  $M^d \pmod{n}$  et retrouve  $m$  :  $M^d \equiv m \pmod{n}$

**End**



# Démonstration mathématiques :

Pourquoi  $M^d \equiv m \pmod{n}$  ?

Othmane a crypté son message par :  $m^e \equiv M \pmod{n}$

$$M^d \equiv m^{(e*d)} \pmod{n}$$

Or  $e*d \equiv 1 \pmod{\phi(n)}$

donc  $\exists k \in \mathbb{Z}$  tel que  $e*d - k*\phi(n) = 1$

donc  $M^d \equiv m^{(1+k*\phi(n))} \pmod{n}$

Petit théorème de Fermat

Si  $p$  est un nombre premier et  $a \in \mathbb{Z}$  alors :

$$a^{(p-1)} \equiv 1 \pmod{p}$$

donc  $m^{(p-1)} \equiv 1 \pmod{p}$

de même  $m^{(q-1)} \equiv 1 \pmod{q}$

d'où  $m^{(k(p-1)(q-1))} \equiv 1 \pmod{p*q}$

donc  $m^{(1+\phi(n))} \equiv m \pmod{n}$

On conclut ainsi que :  $M^d \equiv m \pmod{n}$

# Exemple d'utilisation de RSA, avec des petits nombres :

Jinping souhaiterait envoyer le message suivant à Poutine : « Protect Bachar el Assad». Malheureusement, Joe Biden les espionne, et pourrait intercepter ce message. Nos deux compères vont donc crypter leurs échanges avec la méthode RSA.

Poutine a choisi  $p = 37$  et  $q = 43$ . Il en déduit  $n = 37 \times 43 = 1591$ , et  $\phi(n) = 36 \times 42 = 1512$ . Il choisit ensuite  $e = 19$ , qui est premier avec 1512. L'inverse de 19 modulo 1512 est  $d = 955$ . Poutine peut donc maintenant publier ses clés publiques, par exemple sur son site internet : ( $n = 1591$ ,  $e = 19$ )

Jinping va utiliser ces clés pour crypter son message, mais il doit avant tout convertir son texte en une suite de nombres. Comme Jinping veut envoyer le message sous forme d'un fichier informatique, le mieux est d'utiliser le code ASCII . Ce code attribue un nombre entre 0 et 255 pour chaque caractère de base utilisable sur un ordinateur. En ASCII, «Protect Bachar el Assad» devient :

<b>P</b>	<b>r</b>	<b>o</b>	<b>t</b>	<b>e</b>	<b>c</b>	<b>t</b>		<b>B</b>	<b>a</b>	<b>c</b>	<b>h</b>	<b>a</b>	<b>r</b>		<b>e</b>	<b>l</b>		<b>A</b>	<b>s</b>	<b>s</b>	<b>a</b>	<b>d</b>
80	82	79	84	69	67	84	32	66	65	67	72	65	82	32	69	76	32	65	83	83	65	68

Il suffit à Jinping de coder chaque nombre comme expliqué ci-dessus. Il obtient :  
80^19 [1591] = 1585 ; 82^19 [1591] = 197 ; etc...

80	82	79	84	69	67	84	32	66	65	67	72	65	82	32	69	76	32	65	83	83	65	68
1585	197	82	719	527	780	719	930	20	910	780	1232	910	197	930	527	730	930	910	368	368	910	933

Jinping envoie cette suite de nombres à Poutine, qui va le décrypter avec sa clé d. Il va pouvoir retrouver le message original :  
1585^955 [1591] = 80 ; 197^955 [1591] = 82 ; etc...

1585	197	82	719	527	780	719	930	20	910	780	1232	910	197	930	527	730	930	910	368	368	910	933
80	82	79	84	69	67	84	32	66	65	67	72	65	82	32	69	76	32	65	83	83	65	68
P	r	o	t	e	c	t		B	a	c	h	a	r		e	l		A	s	s	a	d

Bien sûr avec  $p=37$  et  $q=43$  l'opération ne prendra que quelques secondes pour que Joe Biden découvre le message mais si jamais Poutine avait utilisé des clés avec 1024 bits alors la tâche sera un peu plus difficile pour Biden et il devra attendre des mois pour le décoder .

On en déduit ainsi la condition nécessaire pour assurer l'efficacité du cryptage RSA :

Après avoir décomposer en facteurs premiers un nombre de 155 chiffres en 1999 , la société RSA DATA SECURITY recommande des nombres de 309 voir 617 chiffres , les deux nombres premiers dépassent donc les 100 chiffres donc il serait quasiment impossible de les trouver à partir de la clé publique  $n$

```
puissance.py - C:/Users/Zineb/Downloads/puissance.py (3.8.2)
File Edit Format Run Options Window Help
x = float(input("veuillez saisir la valeur de x : "))
y = float(input("veuillez saisir la valeur de y : "))
p = x ** y
print (p%1591)|
```

```
Python 3.8.2 Shell
File Edit Shell Debug Options Window Help
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/Zineb/Downloads/puissance.py =====
veuillez saisir la valeur de x : 80
veuillez saisir la valeur de y : 19
1585.0
>>>
===== RESTART: C:/Users/Zineb/Downloads/puissance.py =====
veuillez saisir la valeur de x : 82
veuillez saisir la valeur de y : 19
197.0
>>>
===== RESTART: C:/Users/Zineb/Downloads/puissance.py =====
veuillez saisir la valeur de x : 79
veuillez saisir la valeur de y : 19
82.0
>>> |
```



# Génération de e :

La clé publique  $e$  doit impérativement être première avec  $\phi(n)$ . Nous devons donc construire un programme python capable de déterminer si deux nombres sont premiers entre eux, c'est à dire que  $\text{PGCD}(a,b) = 1$ . La méthode la plus simple et la plus rapide pour déterminer un PGCD reste sans conteste l'algorithme d'Euclide, que nous pouvons programmer de la manière suivante :

\*premiers.py - C:/Users/Zineb/Downloads/premiers.py (3.8.2)\*

File Edit Format Run Options Window Help

```
def pgcd ( a,b) :  
    c=a%b  
    if c==0 :  
        return b  
    else :  
        while c !=0 :  
            a=b  
            b=c  
            c=a%b  
        return b
```

Python 3.8.2 Shell

File Edit Shell Debug Options Window Help

Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

===== RESTART: C:/Users/Zineb/Downloads/premiers.py =====

>>> pgcd (2,3)

1

>>> pgcd (155,345)

5

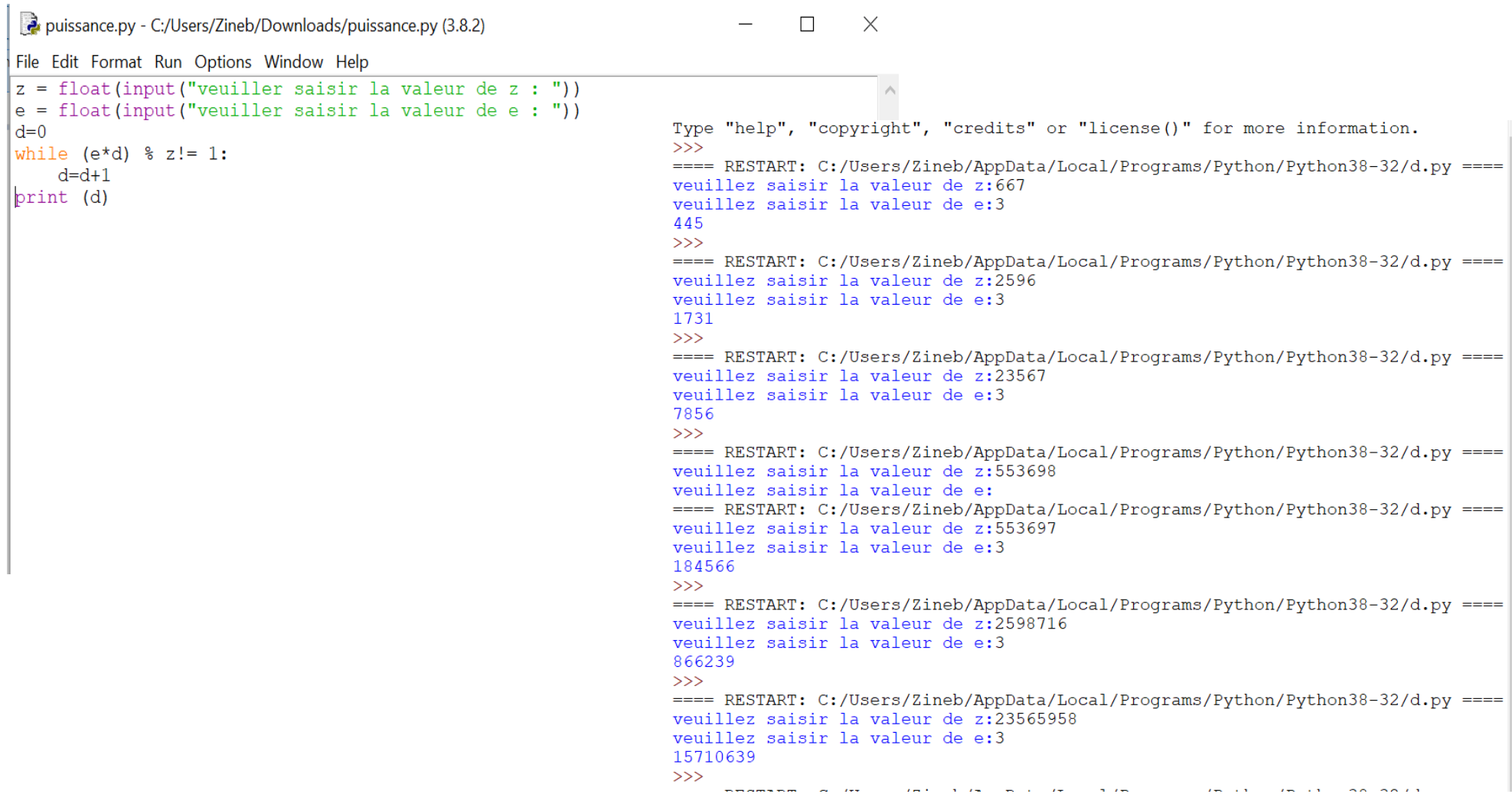
>>> pgcd (177,987)

3

>>> |

# Génération de d :

Le calcul de d, la clé privée, est l'opération la plus lourde. Rappelons tout d'abord que d l'inverse de e modulo  $\phi(n)$



The image shows a screenshot of a Python script named 'puissance.py' and its execution output. The script is located at 'C:/Users/Zineb/Downloads/puissance.py (3.8.2)'. The script's menu bar includes 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The script code is as follows:

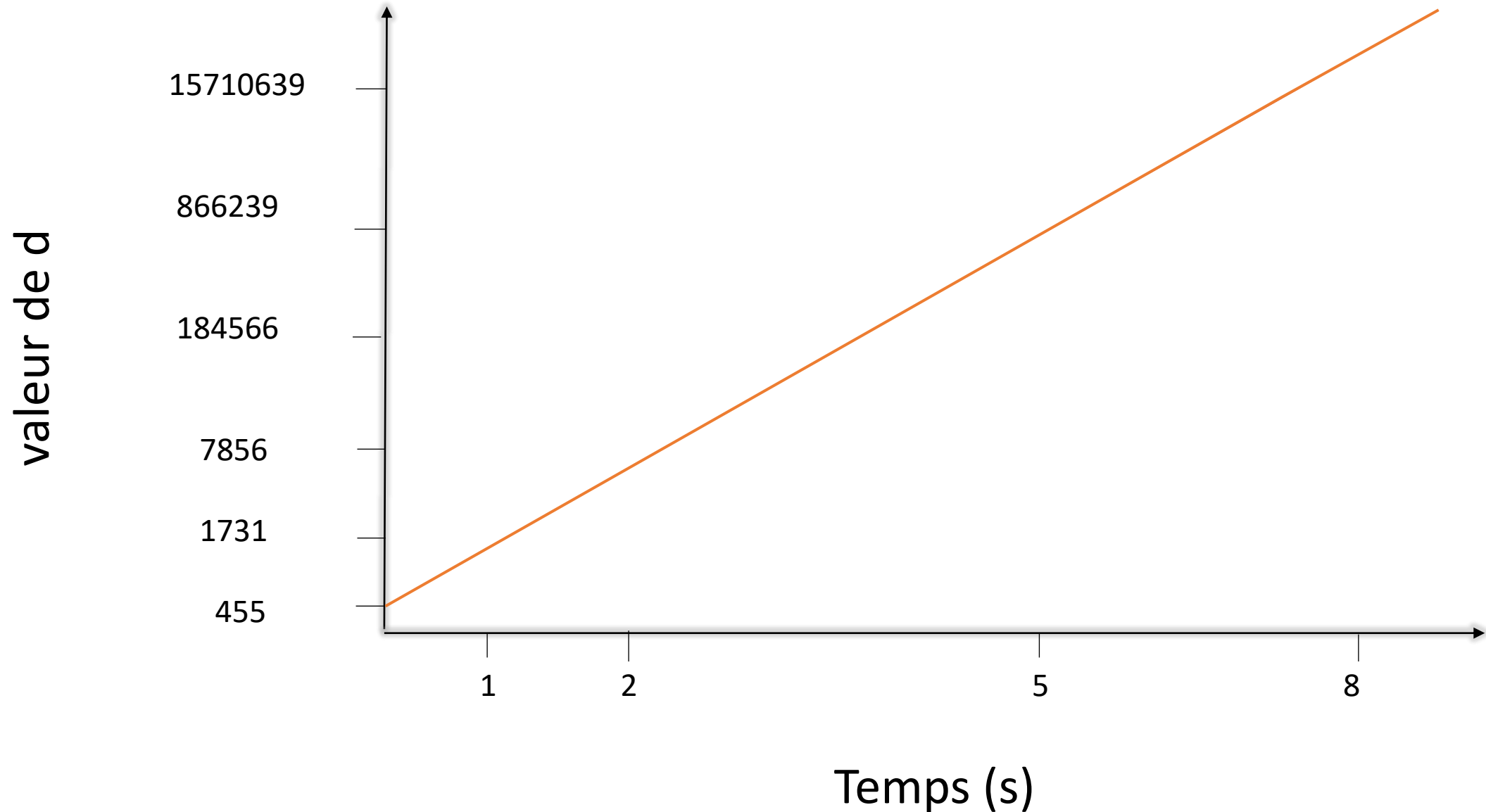
```
z = float(input("veuillez saisir la valeur de z : "))
e = float(input("veuillez saisir la valeur de e : "))
d=0
while (e*d) % z!= 1:
    d=d+1
print (d)
```

The execution output shows the script being restarted multiple times, each time with different input values for 'z' and 'e'. The output for each restart is as follows:

```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:667
veuillez saisir la valeur de e:3
445
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:2596
veuillez saisir la valeur de e:3
1731
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:23567
veuillez saisir la valeur de e:3
7856
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:553698
veuillez saisir la valeur de e:
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:553697
veuillez saisir la valeur de e:3
184566
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:2598716
veuillez saisir la valeur de e:3
866239
>>>
==== RESTART: C:/Users/Zineb/AppData/Local/Programs/Python/Python38-32/d.py ====
veuillez saisir la valeur de z:23565958
veuillez saisir la valeur de e:3
15710639
>>>
```

## Les contraintes auxquelles nous pouvons être confrontés dans le RSA :

en fait, le calcul de  $e$  ne prend que quelques instants. Cependant le calcul le plus difficile et celui de  $d$



# Attaques sur RSA

## Cryptanalyse du RSA en connaissant $\phi(n)$ :

si on connaît  $\phi(n)$  alors on peut factoriser  $n$

Preuve :

$$\begin{cases} p \cdot q = n \\ p + q = n + 1 - \phi(n) \end{cases}$$

donc

$$p^2 - (n + 1 - \phi(n))p + n = 0$$

on tire ainsi :

$$p = \frac{n + 1 - \phi(n) + ((n + 1 - \phi(n))^2 - 4n)^{1/2}}{2}$$

$$q = \frac{n + 1 - \phi(n) - ((n + 1 - \phi(n))^2 - 4n)^{1/2}}{2}$$

## Cryptanalyse de RSA lorsque $m < n^{1/e}$ :

si le message chiffré est court de tel façon qu'il sera inférieur à la racine  $e$  ième de  $n$  ( ça arrive trop si  $e$  est très petit devant  $\phi(n)$  par exemple  $e=3$  ). Comme  $c = m^e \bmod(n)$  , et donc on peut efficacement décrypter  $c$  en extrayant une racine  $e$  ième dans  $Z$ . Ainsi, il n'y a absolument aucune sécurité si l'on chiffre une clef de session 128 bits au moyen de RSA 1024 d'exposant 3 avec la fonction de chiffrement de base. Là encore, cet exemple montre qu'il faut randomiser les messages.

## Cryptanalyse RSA lors de l'envoi du même message à deux personnes différentes qui ont le même exposant $e$ et différents modules dans leur clé :

Cette attaque est basée sur le théorème des restes chinois on a :

$$\left\{ \begin{array}{l} m^e = c \bmod(n) \\ m^e = c' \bmod(n') \\ \dots \\ m^e = c''' \bmod(n''') \end{array} \right.$$



## Théorème des restes chinois

Prenons  $N_1, N_2, \dots, N_p$  des entiers supérieurs à 2 deux à deux premiers entre eux, et  $C_1, C_2, \dots, C_p$  des entiers

Le système d'équations :

$$\left\{ \begin{array}{l} x = C_1 \bmod (N_1) \\ x = C_1 \bmod (N_2) \\ \dots \\ x = C_p \bmod (N_p) \end{array} \right.$$

Admet une unique solution modulo  $N = N_1 * N_2 * N_3 * \dots * N_p$  donnée par la formule :

$$x = C_1 * N_1' * y_1 + \dots + C_p * N_p' * y_p$$

où  $N_i' = N / N_i$  et  $y_i = 1 / (N_i') \bmod (N_i)$  pour  $i$  compris entre 1 et  $n$

Cryptanalyse RSA lors de l'envoi du même message à deux personnes différentes qui ont le même numéro  $n$  dans leur clé :

Si jamais Jinping avait envoyé le même fameux message «Protect Bachar el Assad» à Hassan Rohani et se dernier possède comme clé ( $n=1591$  ,  $e'=3$ )

on a  $\text{pgcd}(e=19, e'=3) = 1$

d'après le théorème de Bezout  $\exists k, k' \in \mathbb{Z}$  tel que  $e*k + k'*e' = 1$

on a  $m \equiv m^{(k*e + k'*e')} \pmod{n}$

$$\equiv m^{(k*e)} * m^{(k'*e')} \pmod{n}$$

$$\equiv c^k * c'^{k'} \pmod{n}$$

$$\text{Car } c = m^e \pmod{n}$$

$$c' = m^{e'} \pmod{n}$$

En déterminant  $k = -4$  et  $k' = 25$  , Biden peut déchiffrer simplement le message bloc par bloc

# Programme attaquant le RSA

```
root@kali:/home/juniper# echo "Protect Bachar el Assad" | openssl enc -base64 >> msg1.b64
root@kali:/home/juniper# cat msg1.b64
UHJvdGVjdCBCYWNoYXZlZWwgQXNzYWQK
```

```
root@kali:/home/juniper# ./rsa-cm.py -c1 msg1.b64 -c2 msg2.b64 -k1 key1.pub.pem -k2 key2.pub.pem
[+] Recovered message:
UHJvdGVjdCBCYWNoYXZlZWwgQXNzYWQK[...]
[+] Recovered bytes:
Protect Bachar el Assad[...]'
'''
```

```
root@kali:/home/juniper# ./rsa-cm.py -h
usage: rsa-cm.py [-h] [-c1 ciphertext1] [-c2 ciphertext2] [-k1 pubkey1] [-k2 pubkey2]
                 [-o outfile]
35
36 '''bash
A simple script to perform RSA common modulus attacks.
37 [-c1 ciphertext1] The first ciphered message
38 [+] Recovered message:
optional arguments:
39 -h, --help            show this help message and exit
40 -c1 ciphertext1       The first ciphered message
41 -c2 ciphertext2       The second ciphered message
42 -k1 pubkey1           The first public key
43 -k2 pubkey2           The second public key
44 -o outfile            Output file
45 And finally the help message.
More info at contact Zineb Moufakkir
'''bash
```

# Annexe

```
#!/bin/env python3
# -*- coding: utf-8 -*-

__author__ = "Zineb"

from Crypto.PublicKey import RSA
from Crypto.Util.number import (
    long_to_bytes,
    bytes_to_long,
    GCD
)
import gmpy2
from base64 import b64decode

import argparse
import sys

def parse_args():
    parser = argparse.ArgumentParser(description="A simple script to perform RSA common modulus attack",
    epilog="More info at contact Zineb Moufakkir")
    parser.add_argument("-c1", type=argparse.FileType("r"), metavar="ciphertext1", required=True,
    help="The first ciphered message")
    parser.add_argument("-c2", type=argparse.FileType("r"), metavar="ciphertext2", required=True,
    help="The second ciphered message")
    parser.add_argument("-k1", type=argparse.FileType("rb"), metavar="pubkey1", required=True,
    help="The first public key")
    parser.add_argument("-k2", type=argparse.FileType("rb"), metavar="pubkey2", required=True,
    help="The second public key")
    parser.add_argument("-o", type=argparse.FileType("wb"), metavar="outfile", required=False,
    help="Output file")
    args = parser.parse_args()
    return args

# Source: https://crypto.stackexchange.com/a/60404
def bytes_to_integer(data):
    output = 0
    size = len(data)
    for index in range(size):
        output |= data[index] << (8 * (size - 1 - index))
    return output

def integer_to_bytes(integer, _bytes):
    output = bytearray()
    for byte in range(_bytes):
        output.append((integer >> (8 * (_bytes - 1 - byte))) & 255)
    return output

def egcd(a, b):
    if (a == 0):
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

# Calculates a^{b} mod n when b is negative
def neg_pow(a, b, n):
    assert b < 0
    assert GCD(a, n) == 1
    res = int(gmpy2.invert(a, n))
    res = pow(res, b*(-1), n)
    return res
```

```
# e1 --> Public Key exponent used to encrypt message m and get ciphertext c1
# e2 --> Public Key exponent used to encrypt message m and get ciphertext c2
# n --> Modulus
# The following attack works only when m^{GCD(e1, e2)} < n
def common_modulus(e1, e2, n, c1, c2):
    g, a, b = egcd(e1, e2)
    if a < 0:
        c1 = neg_pow(c1, a, n)
    else:
        c1 = pow(c1, a, n)
    if b < 0:
        c2 = neg_pow(c2, b, n)
    else:
        c2 = pow(c2, b, n)
    ct = c1*c2 % n
    m = int(gmpy2.iroot(ct, g)[0])
    return m
```

```
def main(args):
    pubkey1 = RSA.import_key(args.k1.read())
    pubkey2 = RSA.import_key(args.k2.read())
    c1 = b64decode(args.c1.read())
    c1 = bytes_to_long(c1)
    c2 = b64decode(args.c2.read())
    c2 = bytes_to_long(c2)
```

```
# We first check that the modulus N of both public keys are equal
if pubkey1.n != pubkey2.n:
    sys.stderr.write("[ERROR] The modulus of both public keys must be the same\n")
    sys.exit(1)
if GCD(pubkey1.e, pubkey2.e) != 1:
    sys.stderr.write("[ERROR] The greatest common denominator between the exponent of each keys is 1\n")
    sys.exit(2)
deciphered_message = common_modulus(
    pubkey1.e,
    pubkey2.e,
    pubkey1.n,
    c1,
    c2
)
deciphered_bytes = long_to_bytes(deciphered_message)

print("[+] Recovered message:")
print(deciphered_message)
print("[+] Recovered bytes:")
print(deciphered_bytes)

if args.o:
    args.o.write(deciphered_bytes)
```

```
if __name__ == '__main__':
    args = parse_args()
    main(args)
```

**Fin de la  
présentation**



**Merci pour  
votre attention**



**Vos questions  
sont les  
bienvenues ...**