```
from IPython.core.display import display, HTML
```

Exercice 1

```
from Crypto.Hash import SHA256
SHA256.new('abc'.encode('utf-8')).hexdigest()
```

```
'ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
```

```
SHA256.new('bbc'.encode('utf-8')).hexdigest()
```

```
'77e6f78af45f649c5f3b8ebe484a91a144eb203a34a89c8dc5b1c4ca87bc6f71'
```

```
def init_password():
    pwd = input("Entrez votre mot de passe ")
    return SHA256.new(pwd.encode()).hexdigest()
display(HTML("<h3>Inscription</h3>"))
hash_pwd = init_password()
hash_pwd
```

Inscription

```
Entrez votre mot de passe  test
```

```
'9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08'
```

```
def check_password(clear_password, password_hash):
    return SHA256.new(clear_password.encode()).hexdigest() ==
password_hash
display(HTML("<h3>Vérification</h3>"))
```

```
pwd = input("Entrer votre mot de passe ")
check_password(pwd, hash_pwd)
```

Vérification

```
Entrer votre mot de passe  tutu




False
```

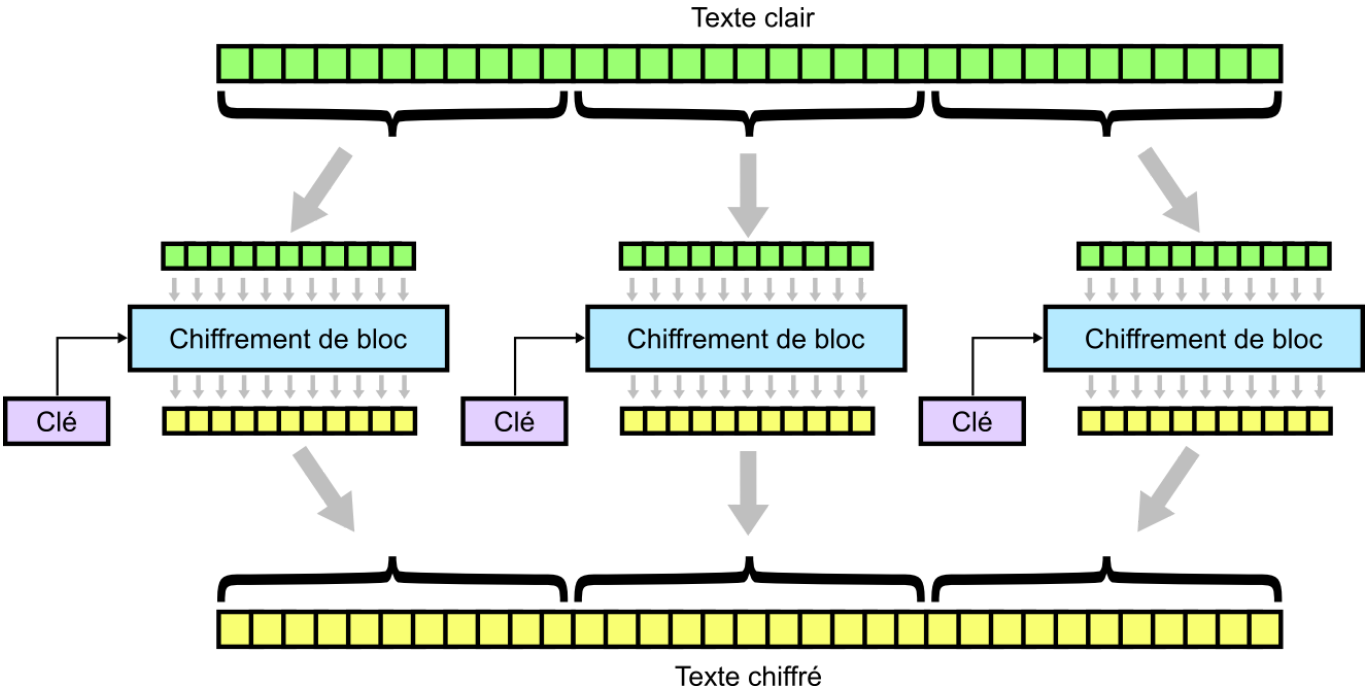
Les fonction de hashage:

- Collision : deux entrèes dont la fonction de hash retourne la même valeur
- Fonction de hash:
  - crc32 - Checksum (rapide) -> beaucoup de collisions
  - md5 - rapide mais CASSÉ
  - sha1 - un peu sécure (pas trop)
  - sha256 - un peu plus sécure (bof)
  - sha512 - un peu mieux
  - Bcrypt: défini le temps d'exécution + ajout d'un sel -> la rendre plus lente et donc éviter les attaques Brute force
- Les attaques :
  - Attaque brute-force (on essaye tout et n'importe quoi)
  - Attaque par dictionnaire (On essaye de termes probables)
  - Attaque par Rainbow Table

Exercice 2

Q.1

Le mode « Electronic Code Book » (ECB)



> Source : Wikipedia

Faille:

- Deux bloc contenant la même chaîne de caractère auront la même valeur chiffrée. On peut déduire la valeur de ces blocs en utilisant la fréquence d'apparition des mots ...

```
from Crypto.Cipher import DES
key = '01234567'
des = DES.new(key, DES.MODE_ECB)
text = 'abcdefgh'.encode('utf-8')
cipher_text = des.encrypt(text)
print(cipher_text)
des.decrypt(cipher_text)
```

```
b'\xec\xc2\x9e\xd9] a\xd0'




b'abcdefgh'
```

Que se passe-t-il si `text='test'` ?

```
text = 'test'
cipher_text = des.encrypt(text)
```

```
---------------------------------------------------------------------------
-

ValueError                                Traceback (most recent call
last)

<ipython-input-9-d83400233fee> in <module>
      1 text = 'test'
----> 2 cipher_text = des.encrypt(text)


~/.pyenv/versions/ENSEM_kernel/lib/python3.7/site-
packages/Crypto/Cipher/blockalgo.py in encrypt(self, plaintext)
    242             return res
    243
--> 244         return self._cipher.encrypt(plaintext)
    245
    246     def decrypt(self, ciphertext):


ValueError: Input strings must be a multiple of 8 in length
```

Remarque :

- La taille du texte en entrée doit êter un multiple de 8 (taille d'un bloc)
- Pour contourner ce problème on utilise la méthode de padding (remplissage du dernier bloc incomplet)

> Source pour les fonctions `pad()` et `unpad` : https://gist.github.com/crmccreary/5610068

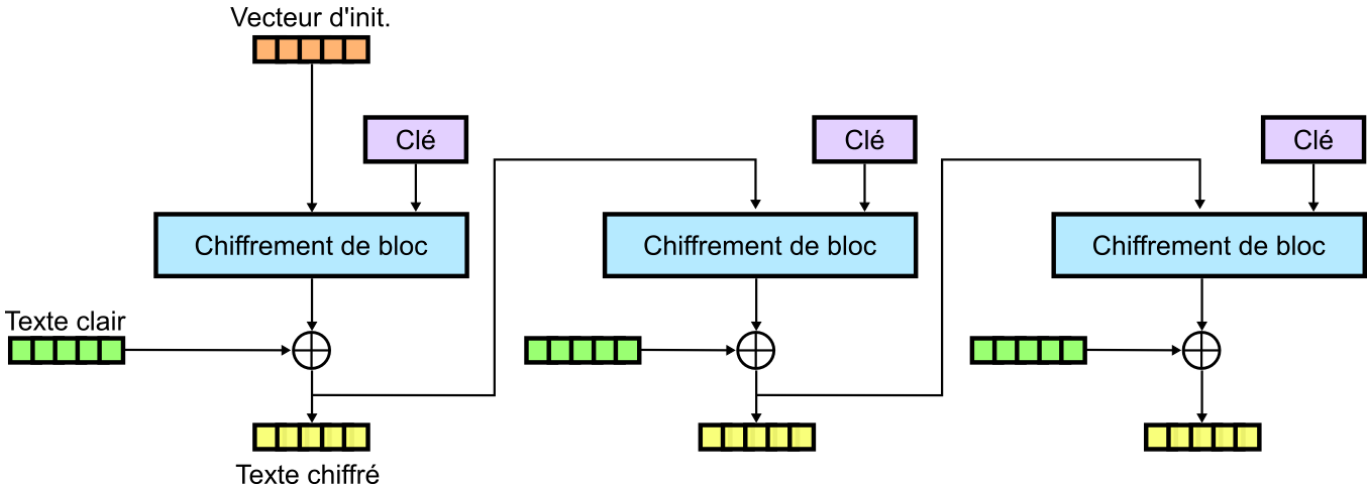
```
BS = 8
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(s[-1])]
cypher_pad = des.encrypt(pad('test'))
```

```
unpad(des.decrypt(cypher_pad).decode('utf-8'))
```

```
'test'
```

Q.2

Le mode « Cipher Feedback » (CFB)



> Source Wikipedia

- Agit comme un chiffrement par flux
- La clé est obtenue en chiffrant le bloc précédemment chiffré, il faut donc un vecteur qui initialise la première clé

Génration du vecteur d'initialisation

- Ce vecteur peut-être partagé (ce n'est pas un secret), mais n'utilisez pas le même vecteur à chaque fois ...
- Bonne pratique : générer aléatoirement ce vecteur

```
from Crypto import Random
iv = Random.new().read(DES.block_size)
iv
```

```
b'\xf2\x8dZ\x11\xf3\xb4\xb5d'
```

Chiffrement

```
des_cfb_1 = DES.new(key, DES.MODE_CFB, iv)
des_cfb_2 = DES.new(key, DES.MODE_CFB, iv)
cipher_text = des_cfb_1.encrypt(text)
cipher_text
```

```
b'\xde\xa9\xef\x99'
```

Déchiffrement

```
decoded_text = des_cfb_2.decrypt(cipher_text)
decoded_text
```

```
b'test'
```

Algorithme ARC4

```
from Crypto.Cipher import ARC4
obj1 = ARC4.new('01234567')
obj2 = ARC4.new('01234567')
text = 'abcdefghijklmnop'
cipher_text = obj1.encrypt(text)
cipher_text
```

```
b'\xf0\xb7\x90{#ABXY9\xd06\x9f\xc0\x8c '
```

```
obj2.decrypt(cipher_text)
```

```
b'abcdefghijklmnop'
```

La valeur de la clé (*symétrique*) ?

`01234567`

Exercice 3

```
from Crypto.PublicKey import RSA
random_generator = Random.new().read
key = RSA.generate(1024, random_generatortor)
key
```

```
<_RSAobj @0x10f85d2e8 n(1024),e,d,p,q,u,private>
```

```
public_key = key.publickey()
enc_data = public_key.encrypt('abcdefgh'.encode(),32)
enc_data
```

```
(b"\xb8\x9ey\xceZ\xc5\x11~\x8c[QQ\x02\x94\x05\xf9\x1f^\x05\xd9}\x1e\xd8\x8
b\xe0\xdcV\xf7\xa1\n~\x10v\xbe\xb8\x1f\xd4\xa2\xd8\x8a\x1cl\xaf\xd0]xd'\xa
f\x91\x89\x98\xf0\x93\xc3%'\xa4QA'\x98\x8c\nt\xb9x;\x8e\xad7\x90\x10\xd9t\
xe1\xc3\xe9~\x06r\x7f\x93\xe0BB\x8c\xebU\xfa\x8d{6C\x180g\xa4\xb3\xf7\x9b\
xf1hWXT\xfc\xd3\x02f\xf0\x14Yf-\xf5\xb5!%\xa0%\x14\xdd\xcb\xd0\xdc\xcbk",)
```

```
key.decrypt(enc_data)
```

```
b'abcdefgh'
```

Q. 1

Clé privée au format PEM

```
key.exportKey()
```

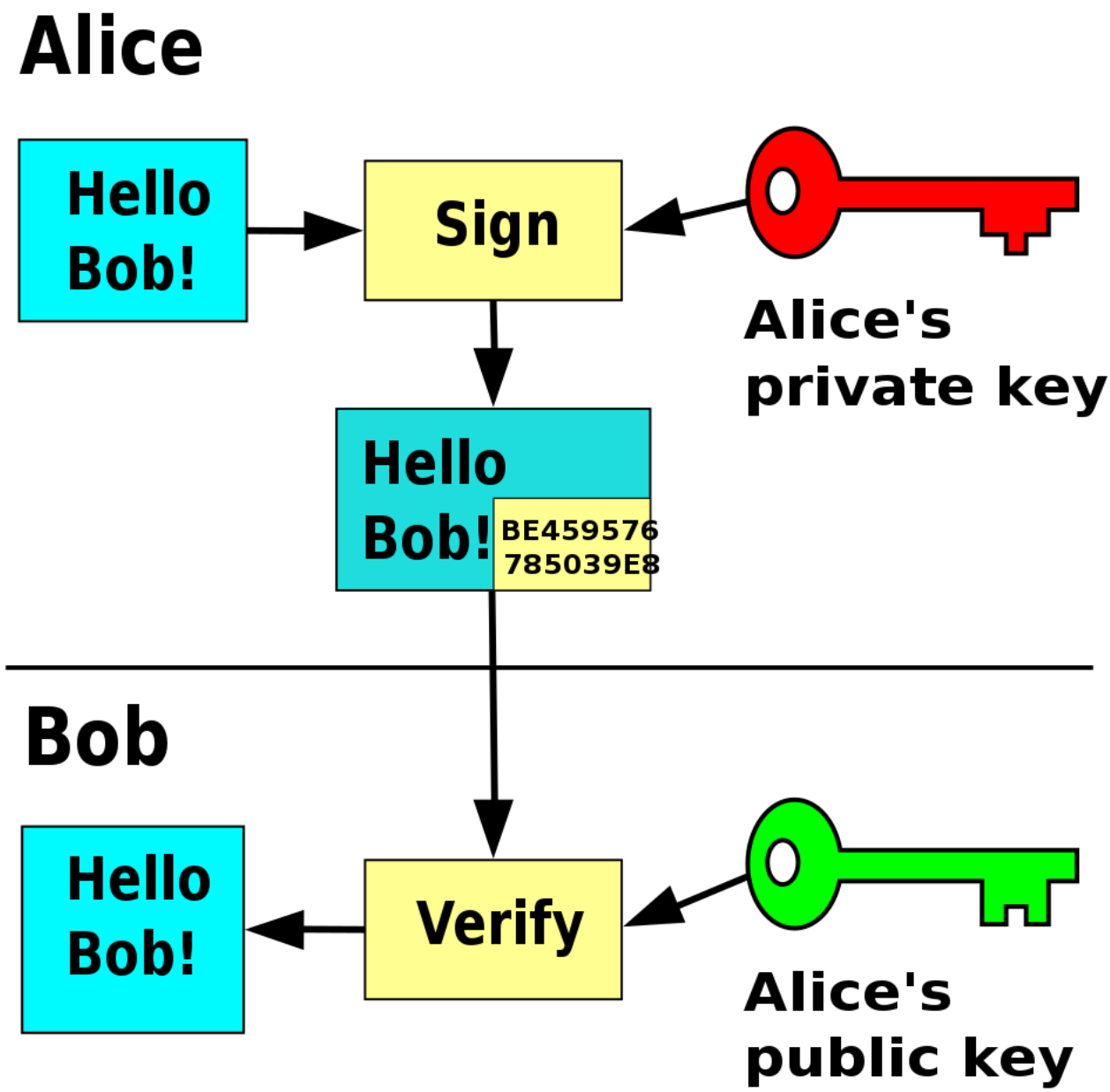
```
b'-----BEGIN RSA PRIVATE KEY-----
\nMIICXAIBAAKBgQDWm4GKAiA6PNcchdufsj3RfkwiXqJQTZduNGJkMXvmV1Csf33N\nUp7hVv
PkDb/5JeIThFlCzhKmYnI5a6IR2Nc7PYFbipIjFNqfyeGmrjyh4SpoHRwH\nG5ReIeE/3pnduI
WiH4bBno0fqzNdlJzrVCS8HUrki2cpS/LuoVlITnXanwIDAQAB\nAoGASCEuEb+TK1/FgQ2wem
VDB8VmO2sJLOpXSjALHdfP5+RS9hdHphnt0pkSEgVZ\nD5le+eZSxfRTKl8OOyRk7ZCvDdmJIb
f9k3HU1UbGuzky/0quSGmcoLOw50z4jmFx\npoKsCxcygHcciLuk2F2CZEPfu23taqeGzps5O8
CGZPJ4FEECQQDkLAYWNafxDQmJ\nfDtL8ZA8vHadsmBGSDLzuEkHWf1i40xxul+uFXSj8IbitA
2XpnkU6UmR7PJp8Ete\nFhkbSeGhAkEA8Mf4o5cNJMC7S7RSaKvULT/xJnfilx+22DRieRwybe
UGadNQMmZO\ndM3loA+X9zs42BfeNYMFu438rKNo7a7UPwJAKv2QeAbMCcLnFRxvH7P39TmdL8
E4\nGoXn87u5JsVx1Hv3H0Z6WJZ6T5k2EOjT1srq7WG3FuOKGSPJT4TiXh3hgQJBANA6\n0E+B
oW9IcnHNF3vIJi2f38h54274kWM3InZEqWQarco1YmTDz1P6gHxFIBJCmC1h\nrt0uW3XRHFSc
2XC8QncCQAGNutocjgMVHyTil9fTAqcqmklgM1Y6CWXdjQhYI1Gl\nTDqb+/6kwQTZIhyGs6eX
POETRaVmkBc+e9Hb6YJ8jU4=\n-----END RSA PRIVATE KEY-----'
```

Clé publique au format PEM

```
key.publickey().exportKey()
```

```
b'-----BEGIN PUBLIC KEY-----
\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDWm4GKAiA6PNcchdufsj3Rfkwi\nXqJQTZ
duNGJkMXvmV1Csf33NUp7hVvPkDb/5JeIThFlCzhKmYnI5a6IR2Nc7PYFb\nipIjFNqfyeGmrj
yh4SpoHRwHG5ReIeE/3pnduIWiH4bBno0fqzNdlJzrVCS8HUrk\ni2cpS/LuoVlITnXanwIDAQ
AB\n-----END PUBLIC KEY-----'
```

Signature numérique



> Source Wikipedia

- Alice signe le message en ajoutant une version du message (ou son hash) chiffré avec sa clé privée
- Bob déchiffre la signature avec la clé publique d'Alice et compare le résultat avec le message en clair reçu

Emetteur

```
text = 'abcdefgh'
hash_text = SHA256.new(text.encode()).hexdigest()
hash_text
```

```
'9c56cc51b374c3ba189210d5b6d4bf57790d351c96c47c02190ecf1e430635ab'
```

```
signature = key.sign(hash_text.encode(), '')
signature
```

```
(18808299963020434294727396364242714647704321685759802988427467491607460980514731372193822935669523122732482817127254951773775441469989055796793749706474333656445096209798803364615100423351066826934773721661188701223871060809271717141598593347811647620654423778492120371919962790717638217873005457924346527859,)
```

Récepteur

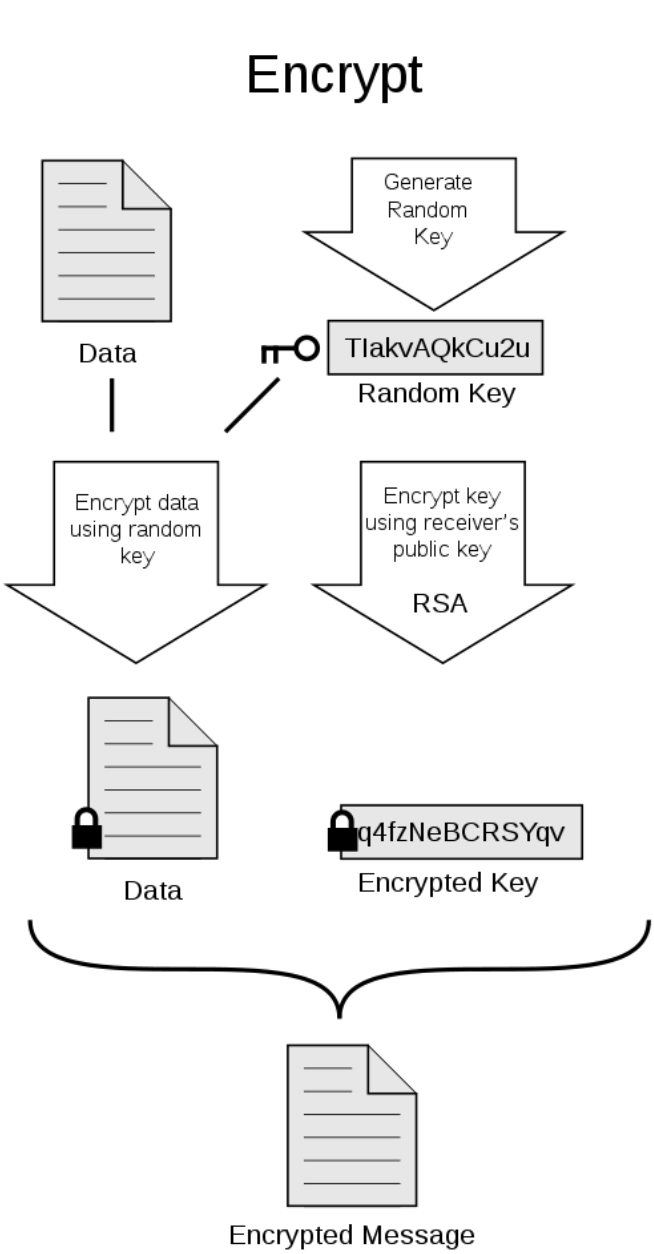
```
text = 'abcdefgh'
hash_text = SHA256.new(text.encode()).hexdigest()
public_key.verify(hash_text.encode(), signature)
```

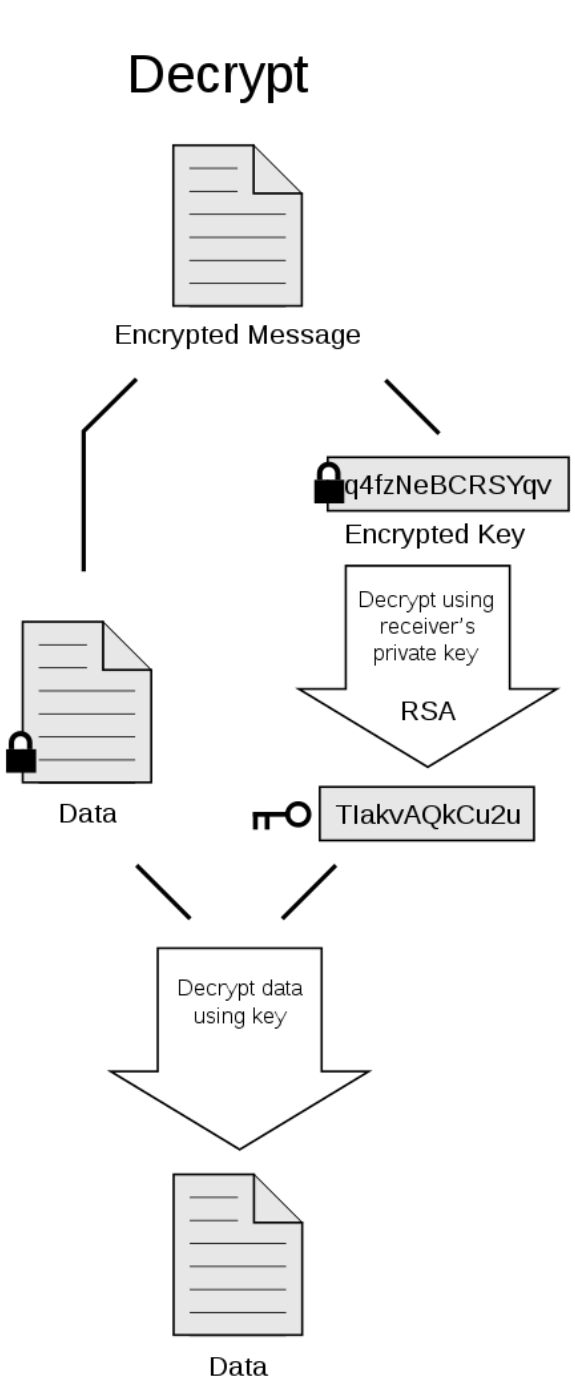
```
True
```

Exemples d'utilisation de RSA

Logiciel PGP (Pretty Good Privacy)

> Chiffrement et signature de messages (email)





> Source Wikipedia

Solution hybride qui combine à la fois :

- Un algorithme de chiffrement symétrique pour chiffrer le message ...
- ... un algorithme asymétrique pour chiffrer la clé symétrique (avec la clé publique) et la déchiffrer (avec la clé privée)

Authentification avec clés pour SSH

- Le serveur détient votre clé publique
- Il peut vérifier que vous êtes en possession de la clé privée correspondante (sans que vous ayez à la communiquer)