Application de RSA Standard et RSA Unidirectionnel

Introduction :

Pour réussir cette phase nous allons utiliser les outils et technologies nécessaires, ci-dessous, à l'implémentation de notre application.

- **Python**

Figure 12 – Logo Python

Python est un langage de programmation open source interprété côté serveur et non compilé. Créé par Guido van Rossum, il est utilisé pour le développement web, de jeux vidéo et autres logiciels ainsi que pour les interfaces utilisateurs graphiques. Il a notamment été utilisé dans la création d'Instagram, de YouTube et de Spotify, et est l'un des langages de programmation officiels de Google. L'interpréteur Python peut être facilement étendu par de nouvelles fonctions et types de données implémentés en C ou C++ (ou tout autre langage appelable depuis le C). Python est également adapté comme langage d'extension pour personnaliser des applications.

- **Django**

Figure 13 – Logo de django

Django est un framework python open-source consacré au développement web 2.0 . Les concepteurs de Django lui ont attribué le slogan suivant : " Le framework web pour les perfectionnistes sous pression ". Il est donc clairement orienté pour les développeurs ayant comme besoin de produire un projet solide rapidement et sans surprise ... c'est à dire à tous les développeurs !

Comme il est toujours compliqué de partir de rien, Django nous propose une base de projet solide. Django est donc une belle boîte à outils qui aide et oriente le développeur dans la construction de ses projets.

Django a été créer en 2003 et a été publié sous licence BSD en juillet 2005.

• **Pycharm**

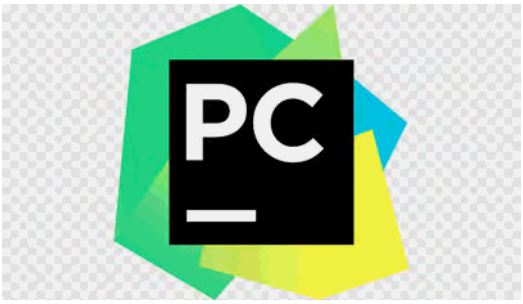


Figure 14 – Logo pycharm

PyCharm est un environnement de développement intégré dédié aux langages Python et Django. Ce logiciel existe en deux versions disponibles en téléchargement, une version libre et gratuite appelée Community Edition et une version complète payante Professional Edition.

nous disposerons d'une foule de fonctionnalités d'édition comme la coloration syntaxique, la complétion automatique du code ainsi que sa factorisation. Le logiciel s'avère utile pour analyser et réparer automatiquement le code source Python et Django.

3.1 Implémentation de RSA Standard

Montrons comment les algorithmes RSA fonctionnent en Python. Le code ci-dessous générera une paire de clés RSA aléatoire, cryptera un message qu'on veut chiffré et le décryptera à sa forme d'origine, en utilisant le schéma de remplissage RSA-OAEP.

Tout d'abord, installons le package pycryptodome, qui est une puissante bibliothèque Python de primitives cryptographiques de bas niveau (hachages, codes MAC, dérivation de clé, chiffrements symétriques et asymétriques, signatures numériques) :

```
(myenv) mamy@mamy-ThinkPad-X240:~/Documents/appliRSA$ pip install pycryptodome
```

Figure 15 – Installation du bibliothèque pycryptodome

Importation des packages pour l'algorithme RSA en python

```
from pathlib import Path
from django.shortcuts import render
from django import template
from django.http import HttpResponse

from django.views.decorators.csrf import csrf_exempt
import json
import zlib
import base64
import ast
register = template.Library()
from .models import *
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
```

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.Cipher import PKCS1_v1_5
import binascii
from hashlib import sha512
```

Figure 16 – Importation des packages

3.1.1 Chiffrement/déchiffrement

Maintenant, écrivons le code Python. Tout d'abord, générons les clés RSA (3072 bits) et récuperons la clé publique dans un fichier qu'on nomme **"public.pem"** et la clé privée dans un fichier appelé **"private.pem"** au format PEM

```
def index(request):
    new_key = RSA.generate(3072)
    private_key = new_key.exportKey("PEM")
    public_key = new_key.publickey().exportKey("PEM")
    private_key_path = Path('private.pem')
    private_key_path.touch(mode=0o600)
    private_key_path.write_bytes(private_key)
    public_key_path = Path('public.pem')
    public_key_path.touch(mode=0o664)
    public_key_path.write_bytes(public_key)
```

```
private_key = Path('private.pem')
public_key = Path('public.pem')
rsa_public_key = RSA.importKey(public_key.read_bytes())
rsa_public_key = PKCS1_OAEP.new(rsa_public_key)
rsa_private_key = RSA.importKey(private_key.read_bytes())
rsa_private_key = PKCS1_OAEP.new(rsa_private_key)

print(f"Public key:  (n={hex(new_key.n)}, e={hex(new_key.e)})")
print(f"Private key: (n={hex(new_key.n)}, d={hex(new_key.d)})")
```

Figure 17 – Génération des clés

Voici la clé publique dans le fichier **"public.pem"**

```
-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEA54do46Z16k3mupDADaxU
QcYwFi4eKPr5W3MGQwD0hy1TMV0Ph++QZgND0lEZvQvtIejfUwokNu3y4q59dzwX
mzUR6JSlwukMjwYGZjge4ZNF1mm0nNKgDjPEmdVUOnWATIxONJsQVjAy9Jfrh5/7
ee43lqUrLWFg5tahAwdvuVoHXmR24DK6blF825xGAYLc6OH+LhBdy7CbAB5+aSjK
SRd7MANIK6t4nPZ+bgnqL7pUvApalyALyZIDCBUwczCFCpLjJ/nCiEmVJoIlg1EW
egvora36Is6CkQ0ic5PKrzXWNWYdiusxRVbDtdjo13XlKc5+b8FPRDaX+rFGX4fh
IQBse/BN9H6e1KCPdB1wHwpieLeo8gldgs+i9wcy9/Xq0R1b7yjtxR17I0z/Ama5
tnox++3ba7COhllVNx1GiTQ7EQV8zM7WhpVtOVXQc6DTYJ/NI7jNeBo6p/kf2OS8
Vw3cvnTbbxVf2Nr551UK8dwe2t2+z/RWIfwDedrFqzxTAgMBAAE=
-----END PUBLIC KEY-----
```

Figure 18 – La clé publique

La clé privée est stockée dans le fichier **"private.pem"**

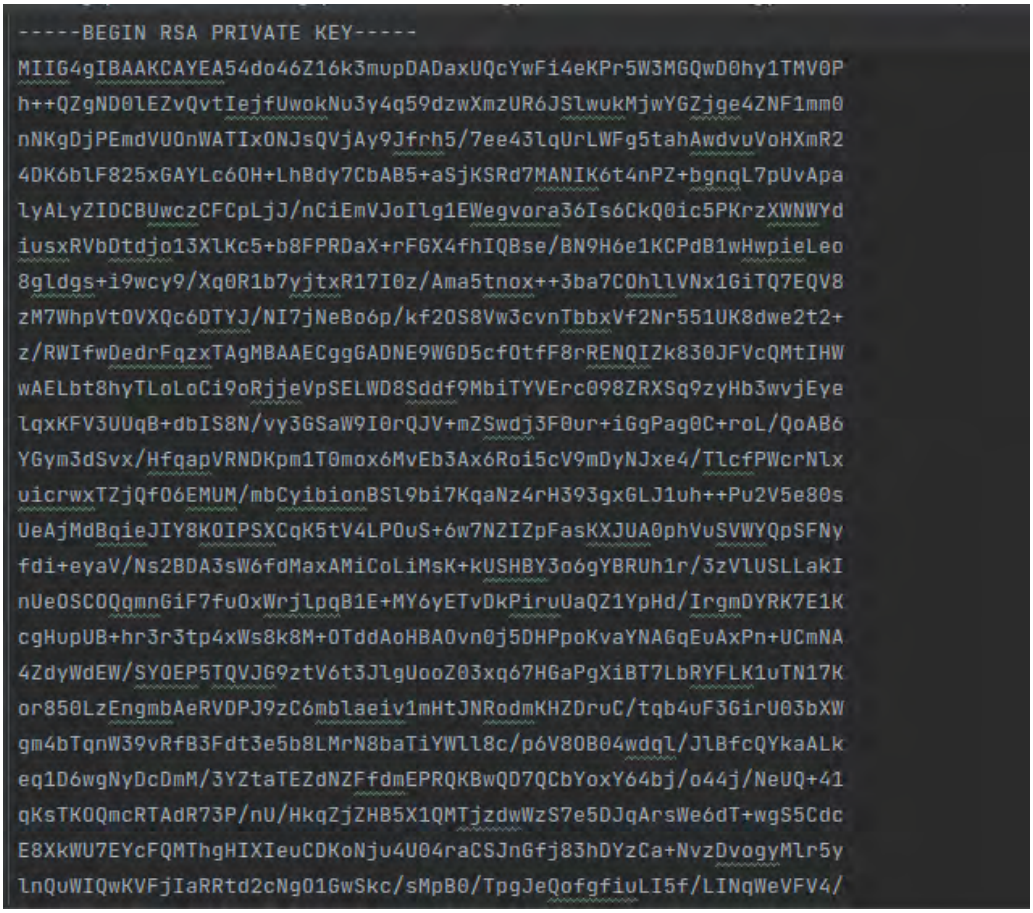


Figure 19 – La clé Privée

On récupére les nombres (n, e) qui forment la clé publique

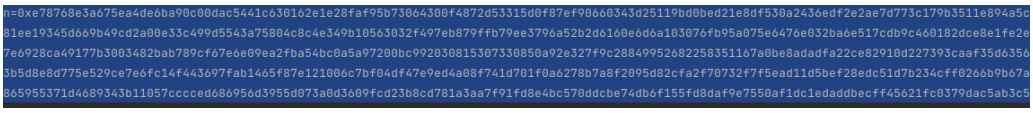


Figure 20 – Le nombre n

Figure 21 – Le nombre e

Et les nombres (n, d) qui forment la clé privée.

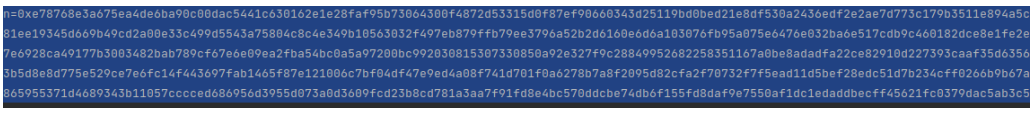


Figure 22 – Le nombre n

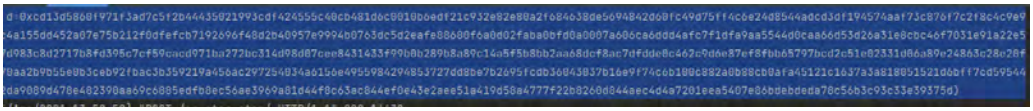


Figure 23 – Le nombre d

Ensuite, on crypte le message au moyen de la clé publique RSA (ici supposée être disponible localement dans le fichier **"public.pem"**) à l'aide du schéma de cryptage RSA-OAEP (RSA avec remplissage OAEP PKCS1) et le text chiffré est stocké dans le fichier **"encrypt.txt"**.

```
if request.method == 'POST' and request.FILES:
    if request.POST.get("form_type") == 'form_encrypt':
        myfile = request.FILES['input_file_message']
        msg = myfile.read()
        encrypted = rsa_public_key.encrypt(msg)
        encrypted_message = binascii.hexlify(encrypted)
        f = open('encrypt.txt', 'wb')
        f.write(encrypted)
        f.close()
```

Figure 24 – Le chiffrement du text en claire

Enfin, on décrypte le message en utilisant RSA-OAEP avec la clé privée RSA qui est stockée dans le fichier **"private.pem"**, le message décrypté se trouve dans le fichier **decrypt.txt**.

```
elif request.POST.get("form_type") == 'form_decrypt':
    myfile = request.FILES['input_file_encrypt']
    fenc = myfile.read()
    print("fenc", fenc)
    try:
        decrypted = rsa_private_key.decrypt(fenc)
        decrypted_message = binascii.hexlify(decrypted)
        f = open('decrypt.txt', 'wb')
        f.write(str(decrypted))
        f.close()
    except ValueError as err:
        print("errorr ", err)
```

Figure 25 – Le déchiffrement du message chiffré

3.1.2 Signature/Vérification

Maintenant, signons un message, en utilisant la clé privée RSA (n, d). Calculons son hachage et augmentons le hachage à la puissance d modulo n (chiffrons le hachage avec la clé privée). Nous utiliserons le hachage SHA-512, il s'adaptera à la taille de clé RSA actuelle (3072). En Python, nous avons l'exponentiation modulaire comme fonction intégrée pow (x, y, n).

La signature numérique obtenue est un entier compris entre zéro et la longueur de la clé RSA [0 ... n] qui est stocké dans le fichier **"sign.txt**, comme illustré ci-dessous

```
elif request.POST.get("form_type") == 'form_sign':
    myfile = request.FILES['input_file_sign']
    message = myfile.read()
    print("message", message)
    hash = int.from_bytes(sha512(message).digest(), byteorder='big')
    signature = pow(hash, new_key.d, new_key.n)
    f = open('sign.txt', 'wb')
    f.write(str(signature))
    f.close()
```

Figure 26 – La signature du message en claire

Ensuite, vérifions la signature en déchiffrant la signature à l'aide de la clé publique (élevons la signature à e modulo n) et comparons le hachage obtenu de la signature(hastFromSignature) au hachage du message signé à l'origine(hash)

```
    elif request.POST.get("form_type") == 'form_verify':
        f = open('sign.txt', 'r')
        signature = f.read()
        myfile = request.FILES['input_file_verify']
        msg = myfile.read()
        hash = int.from_bytes(sha512(msg).digest(), byteorder='big')
        hashFromSignature = pow(int(signature), new_key.e, new_key.n)
        valid = hash == hashFromSignature

return render(request, 'rsa_standard.html', locals())
```

Figure 27 – La vérification de la signature

3.2 Implémentation de RSA Unidirectionnel

3.2.1 Chiffrement/Déchiffrement

Pour l'implémentation de RSA unidirectionnel on crée d'abord deux paires de clés de taille 3072 et 4096. chaque clé est stockée dans un fichier nommé respectivement par "**publicuni1.pem**", "**privateuni1.pem**", "**publicuni2.pem**",et "**privateuni2.pem**". On a donc deux clés publiques et deux clés privées

```
def rsa_uni(request):
    new_key = RSA.generate(3072)
    private_key_uni1 = new_key.exportKey("PEM")
    public_key_uni1 = new_key.publickey().exportKey("PEM")
    private_key_uni1_path = Path('privateuni1.pem')
    private_key_uni1_path.touch(mode=0o600)
    private_key_uni1_path.write_bytes(private_key_uni1)
    public_key_uni1_path = Path('publicuni1.pem')
    public_key_uni1_path.touch(mode=0o664)
    public_key_uni1_path.write_bytes(public_key_uni1)

    private_key_uni1 = Path('privateuni1.pem')
    public_key_uni1 = Path('publicuni1.pem')
    rsa_public_key_uni1 = RSA.importKey(public_key_uni1.read_bytes())
    rsa_public_key_uni1 = PKCS1_OAEP.new(rsa_public_key_uni1)
    rsa_private_key_uni1 = RSA.importKey(private_key_uni1.read_bytes())
    rsa_private_key_uni1 = PKCS1_OAEP.new(rsa_private_key_uni1)

    print(f"Public key:  (n={hex(new_key.n)}, e={hex(new_key.e)})")
    print(f"Private key: (n={hex(new_key.n)}, d={hex(new_key.d)})")
```

```
keypair= RSA.generate(4096)
private_key_uni2 = keypair.exportKey("PEM")
public_key_uni2 = keypair.publickey().exportKey("PEM")
private_key_uni2_path = Path('privateuni2.pem')
private_key_uni2_path.touch(mode=0o600)
private_key_uni2_path.write_bytes(private_key_uni2)
public_key_uni2_path = Path('publicuni2.pem')
public_key_uni2_path.touch(mode=0o664)
public_key_uni2_path.write_bytes(public_key_uni2)

private_key_uni2 = Path('privateuni2.pem')
public_key_uni2 = Path('publicuni2.pem')
rsa_public_key_uni2 = RSA.importKey(public_key_uni2.read_bytes())
rsa_public_key_uni2 = PKCS1_OAEP.new(rsa_public_key_uni2)
rsa_private_key_uni2 = RSA.importKey(private_key_uni2.read_bytes())
rsa_private_key_uni2 = PKCS1_OAEP.new(rsa_private_key_uni2)

print(f"Public key:  (n={hex(new_key.n)}, e={hex(new_key.e)})")
print(f"Private key: (n={hex(new_key.n)}, d={hex(new_key.d)})")
```

Figure 28 – Génération des clés

On récupére les nombres n, e et d pour chaque clé
• Pour les clés de tailles 3072 on a :
- les nombres (n,e) qui forment la clé publique ci-dessous

```
(n=0xecc76886c8c7742d202e11ab8967e6b430a6c2f18cc72be5abd024d8c49b2b1adeb981ea77b860dd6490aa8b8f25f365ac52856c035d8832f36433515981f268337d16e
58e213c82f4c82dd5d4b84bb549ba98a7bda5d8bb63006c6446bba61739f34fdb5cdd54dc2ee1a24e629995266f9032a479fc04b1d88155f1adf9f2ba076f0d41c893b0f46242
3364b1ccc5ebdf915c335b92596c66ee67c76bdb64bc69db9e893ffdcf2fbbcb46bd0e27df01b3b82d82f78bce8908385a8e12dd868a5da6cf85e374dd56e3d5121dab62e3e91
c809428372c7ed7e5c8033be87372a86808ac7ae5cb07276cb909642a86e8370adb14cabec5ad570f7a714207bcd32500bcf5f022dfc17d02b5ed32e348450b5a36989a115aaf
67dd38da9844a6c0439c58954a0884bd91df5f4139901a6c091c28184c012e76b68b5cfaa6fcf9c2a032d3185c31a70f78269ff2af121844eb4d14fc5e6c34f48aca1c2015ebf
```

Figure 29 – Le nombre n

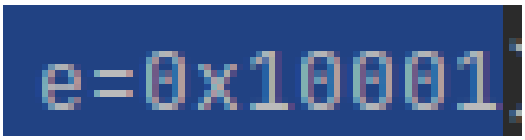


Figure 30 – Le nombre e

La clé publique

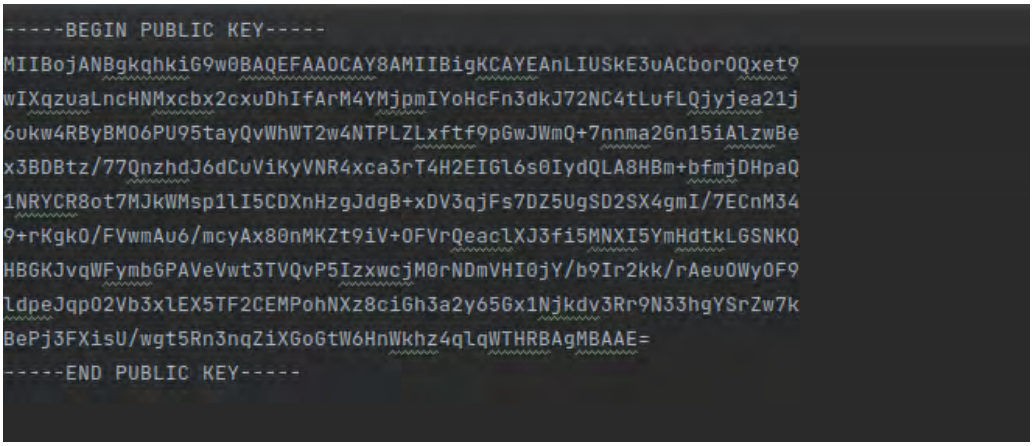


Figure 31 – La clé publique 1

- Les nombres (n,d) qui forment la clé privée ci-dessous

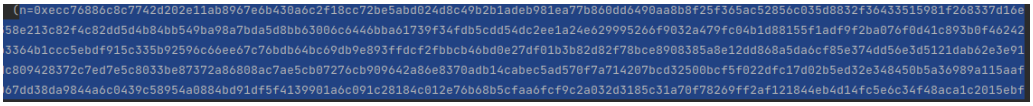
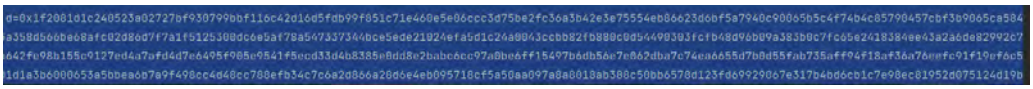


Figure 33 – Le nombre n



La clé privée

Figure 34 – La clé privée 1

• Pour les clés de tailles 4092 on a :
- les nombres (n,e) qui forment la clé publique ci-dessous

Figure 35 – Le nombre n

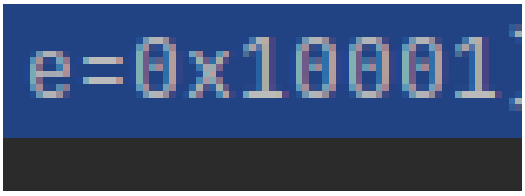


Figure 36 – Le nombre e

La clé publique

Figure 37 – La clé publique 2

- Les nombres (n,d) qui forment la clé privée ci-dessous

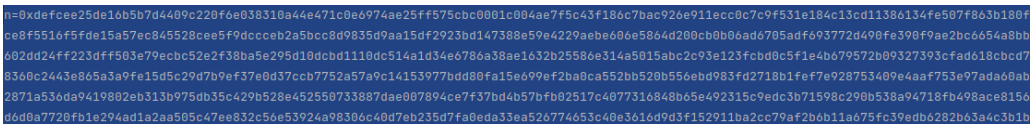


Figure 38 – Le nombre n

Figure 39 – Le nombre d

La clé privée

Figure 40 – La clé privée 2

Ensuite on crypte le message en claire avec la première clé publique *rsa_public_key_uni*1 qui est stockée dans le fichier **publicuni1.pem** et on obtient un message chiffré nommé "encrypted" se trouvant dans le fichier "encryptuni1.txt". Ce dernier sera transformé avec la deuxiéme clé publique **rsa_public_key** qui se trouve dans le fichier **publicuni2.pem** on obtient un second chiffré nommé rsa_encrypted stocké dans le fichier "encryptuni2.txt"

```
if request.method == 'POST' and request.FILES:
  if request.POST.get("form_type") == 'form_encrypt':
    myfile = request.FILES['input_file_message']
    msg = myfile.read()
    encrypted = rsa_public_key_uni1.encrypt(msg)
    encrypted_message = binascii.hexlify(encrypted)
    f = open('encryptuni1.txt', 'wb')
    f.write(encrypted)
    f.close()

    msg1 = encrypted
    rsa_encrypted = rsa_public_key_uni2.encrypt(msg1)
    # print("uni", rsa_encrypted)
    f = open('encryptuni2.txt', 'wb')
    f.write(str(rsa_encrypted))
    f.close()
```

Figure 41 – Le chiffrement des messages

En fin on déchiffre le message "rsa_encrypted" avec la clé privée $rsa\_private\_key\_uni2$ on obtient un déchiffré nommé "decrypted" qu'on stocke dans le fichier "decrypt1.txt". On decrypt le message "decrypted" avec la clé privée $rsa\_private\_key\_uni1$, on obtient "rsa_decrypted" se trouvant dans le fichier "decrypt2.txt" qui contient le message en claire.

```
elif request.POST.get("form_type") == 'form_decrypt':
    myfile = request.FILES['input_file_encrypt']
    msg = myfile.read()
    encrypted = rsa_public_key_uni1.encrypt(msg)
    msg1 = encrypted
    rsa_encrypted = rsa_public_key_uni2.encrypt(msg1)
    decrypted = rsa_private_key_uni2.decrypt(rsa_encrypted)
    print('Decrypted:', decrypted)
    f = open('decrytp1.txt', 'wb')
    f.write(str(decrypted))
    f.close()

    rsa_decrypted = rsa_private_key_uni1.decrypt(decrypted)
    f = open('decrypt2.txt', 'wb')
    f.write(str(rsa_decrypted))
    f.close()
```

Figure 42 – Le déchiffrement des messages chiffrés

3.3 Test et Validation

3.3.1 RSA standard

a. **Page de gérération des clés**

On clic sur le bouton "Générer la paire de clé" et les clés sont générées dans les fichiers **public.pem** et **private.pem** ci-dessus.

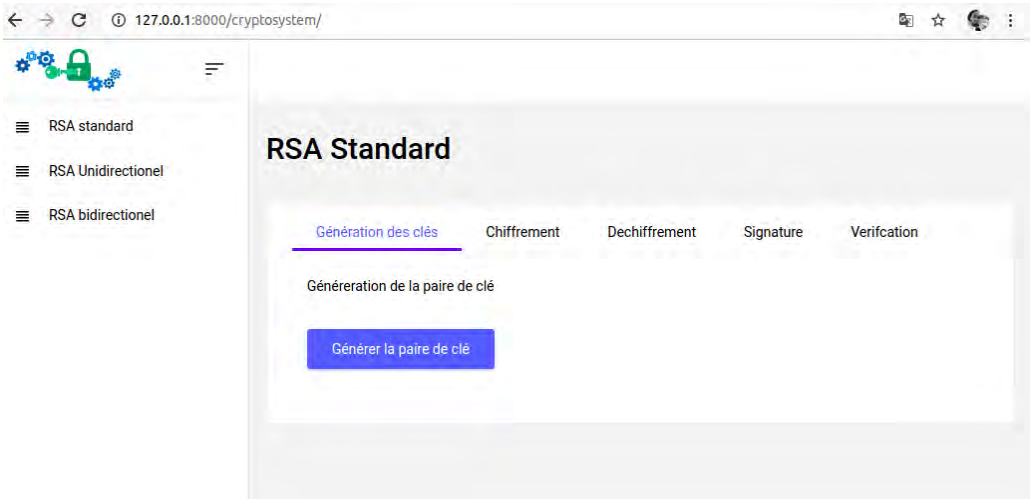


Figure 43 – Page de génération des clés

b. **Page de chiffrement du message**

On veut chiffrer le message stocké dans le fichier "plaintext.txt" ci-dessous avec la clé public générée dans le fichier "public.pem"

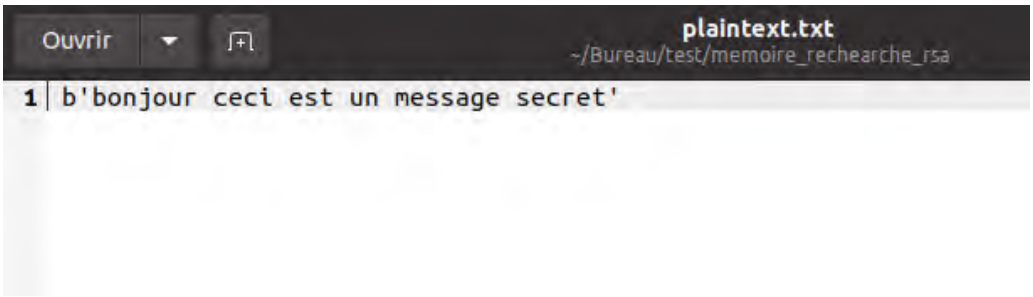


Figure 44 – Le fichier plaintext.txt

D'abord on clic sur le bouton "Upload" et on choisit le fichier "plaintext.txt" contenant le text a chiffré. Ensuite on clic sur "Chiffrer" et le message est chiffré et gardé dans le fichier "encrypt.txt" comme illustré ci-dessous.

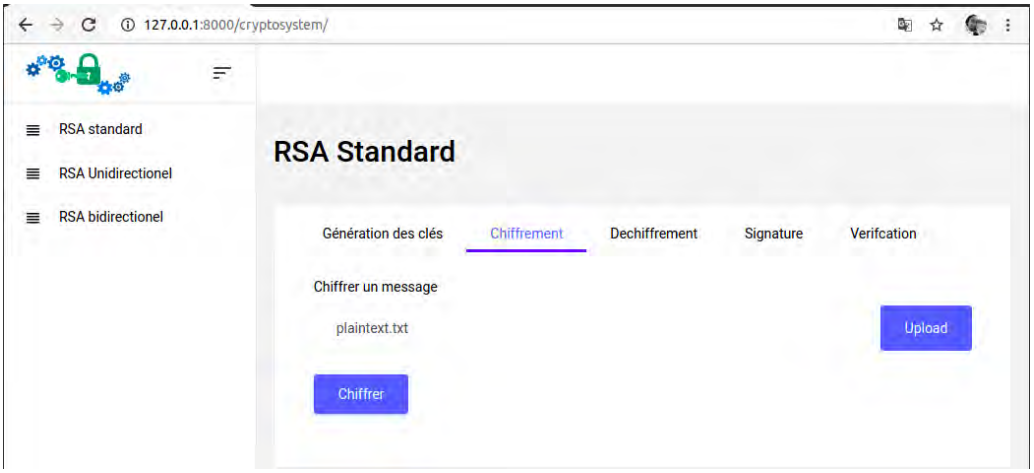


Figure 45 – Page de chiffrement du message

Figure 46 – Le fichier encrypt.txt

c. **Page de déchiffrement du message**

On clic sur le bouton "Upload", ensuite on choisit le fichier contenant le chiffré "encrypt.txt" et enfin on clic sur le bouton "Déchiffrer". Le message est déchiffré à l'aide de la clé privée stockée dans "private.pem" et gardée dans "decrypt.txt".

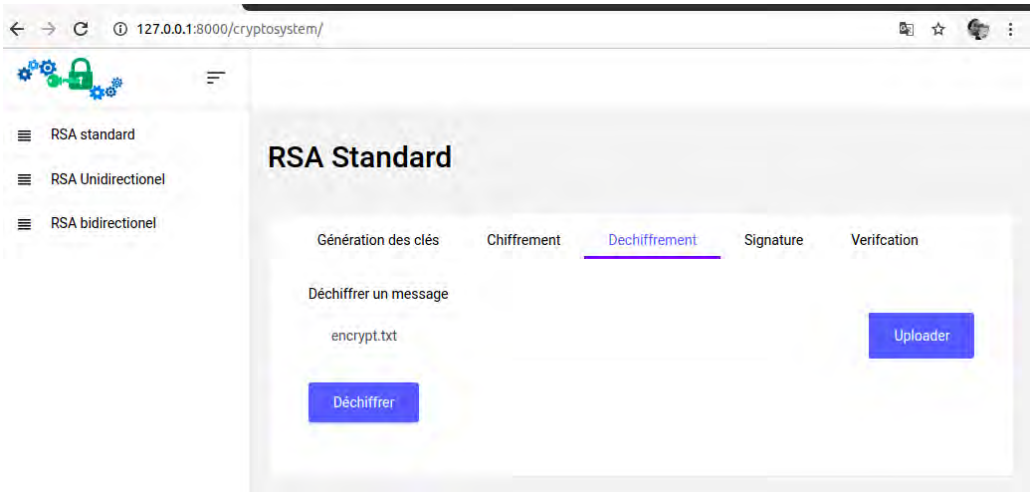


Figure 47 – Page de déchiffrement du message

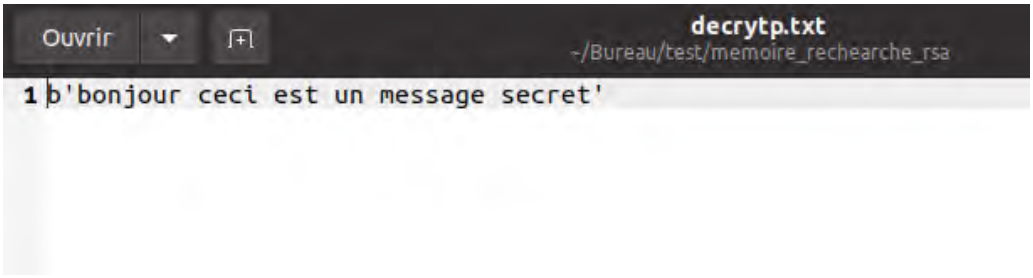


Figure 48 – Le fichier decrypt.txt

d. **Page de signature du message**

on veut signer le message en claire, se trouvant dans le fichier "plaintext.txt", avec la clé privée.

on clic sur "Uplaod" on choisit le fichier "plaintext.txt" et on clic sur le bouton "Signer". Le message signé est gardé dans le fichier **sign.txt**.

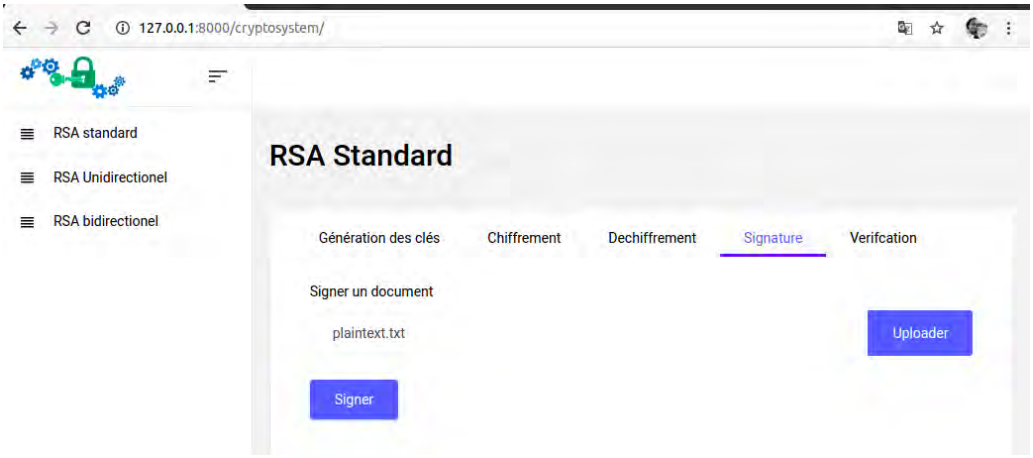


Figure 49 – Page Signature du message

Figure 50 – Le fichier sign.txt

e. **Page de verification de la signature**

Maintenant on va vérifier la signature, on clic le bouton "Uplaod" on choisit le fichier **sign.txt**. Ensuite on clic sur le bouton "Verifier" et on obtient le resultat ci-dessous :

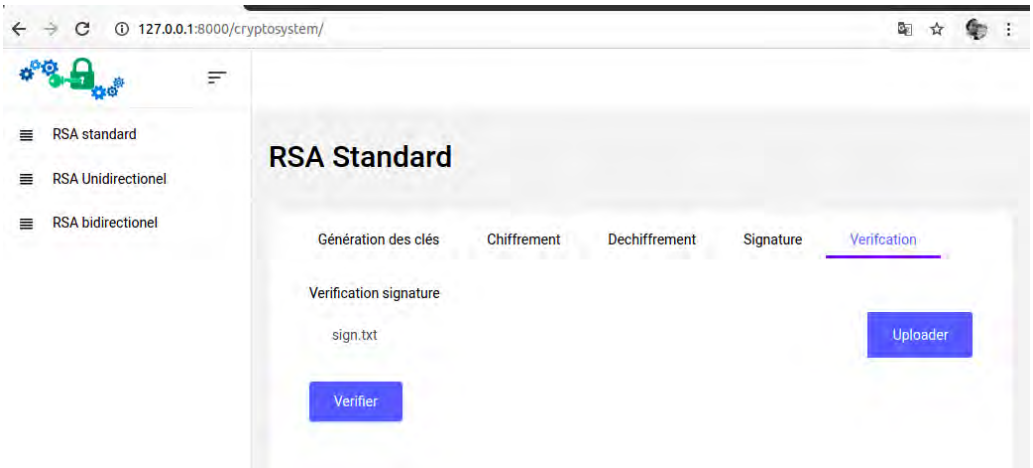


Figure 51 – Page Vérification de la signature

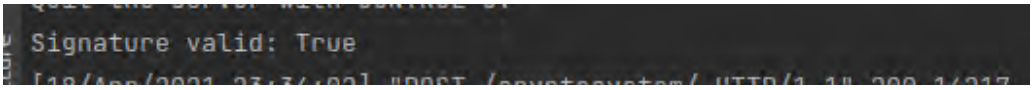


Figure 52 – Résultat de la signature

La signature est valide et retourne **true** car le message reçu est effectivement celui qui a été signé. Dans le cas contraire le résultat serait **false** .

3.3.2 RSA Unidirectionnel

a. **Page de gérération des clés**

On clic sur le bouton "Générer la paire de clé" deux fois et les clés sont générée dans les fichiers **publicuni1.pem**, **privateuni1.pem**, **publicuni2.pem** et **privateuni2.pem** ci-dessus.

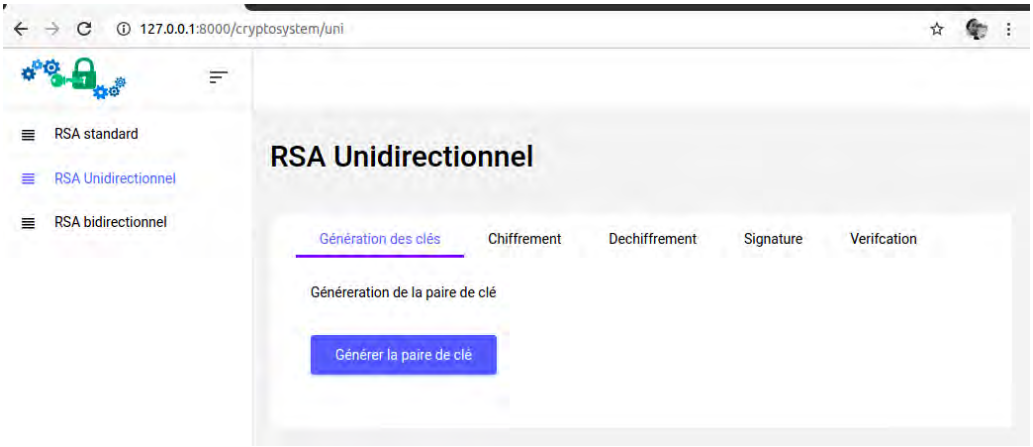


Figure 53 – Page de génération des clés

b. **Page de chiffrement du message**

On veut chiffrer le message se trouvant dans le fichier "plaintext.txt" avec les deux clés publiques

Maintenant on choisit le fichier "plaintext.txt" en cliquant sur le bouton "Uplaod", on clic sur le bouton "Chiffrer" et le message est chiffré et gardé dans le fichier "encrypt1.txt" ci-dessous avec la première clé public qui se trouve dans le fichier **publicuni1.pem**

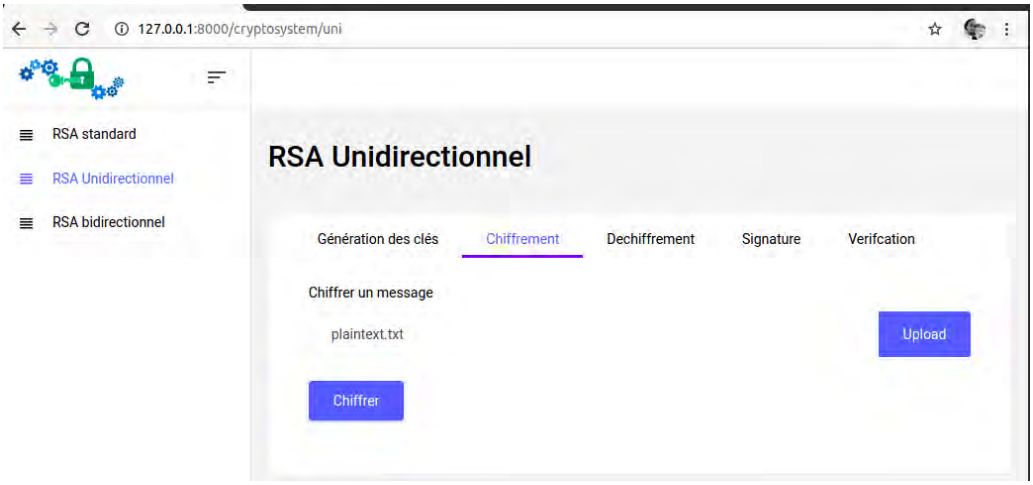


Figure 54 – Page de chiffrement du message stocké dans plaintext.txt

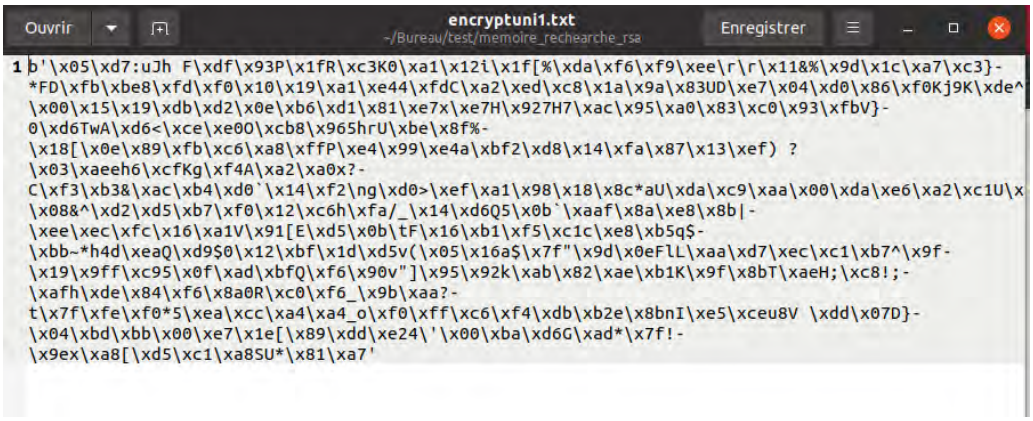


Figure 55 – Le fichier encryptuni1.txt

Ensuite on clic sur le bouton "Upload" et on choisit le fichier "encryptuni1.txt", on clic sur "Chiffrer" et le contenu du fichier est chiffré de nouveau avec la deuxième clé public stocké dans "publicuni2.txt" et est gardé dans le fichier "encryptuni2.txt".

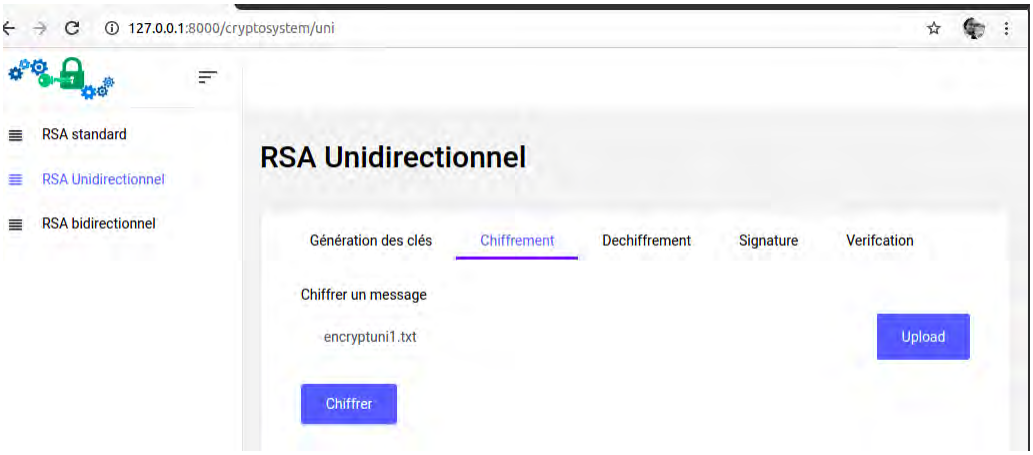


Figure 56 – Page de chiffrement du message stocké dans encryptuni1.txt

Figure 57 – Le fichier encryptuni2.txt

c. **Page de déchiffrement du message stocké dans "encryptuni2.txt"**

On clic sur le bouton "Upload", ensuite on choisit le fichier "encryptuni2.txt" et on clic sur le bouton "Déchiffrer" pour le premier déchiffrement. Le contenu du fichier est déchiffré avec la deuxième clé privée stockée dans "privateuni2.pem" et le chiffré est gardé dans le fichier "decrypt1.txt"

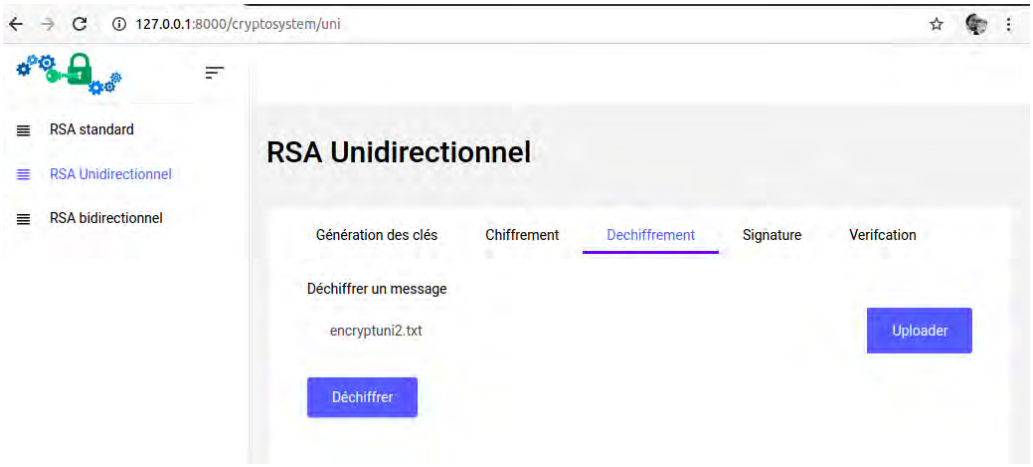


Figure 58 – Page de déchiffrement du message stocké dans encryptuni2.txt

Figure 59 – Le fichier decrypt1.txt

d. **Page de déchiffrement du message stocké dans decrypt1.txt**

on clic sur "Uplaod", on choisit le fichier "décrypt1.txt et on clic sur le bouton "Déchiffrer". Le contenu du fichier est déchiffré avec la premiere clé(privateuni1.pem) et est gardé dans le fichier **decrypt2.txt**

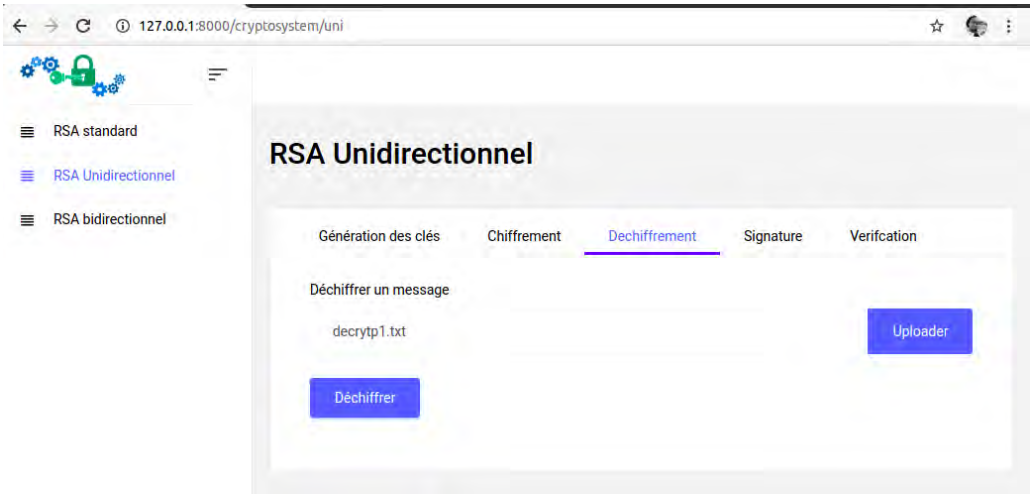


Figure 60 – Page de déchiffrement du message stocké dans "décrypt1.txt"

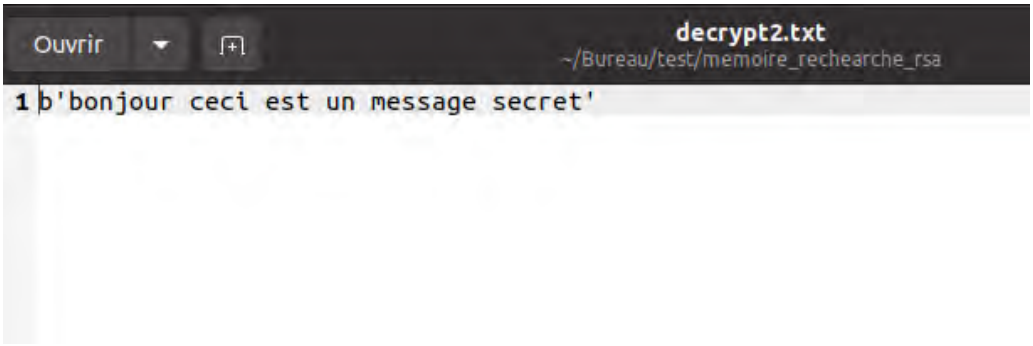


Figure 61 – Le fichier decrypt2.txt

Conclusion :

Dans ce chapitre nous avons utilisé le framework Django et le langage python avec l'algorithme RSA afin de développer une application web pour le RSA standard et RSA unidirectionnel qui nous permet de faire le chiffrement, le déchiffrement, la signature et la vérification des données.

.