

1 Blockchain

La primera parte de la práctica consistirá en implementar una [Blockchain](#) sencilla para entender su funcionamiento.

Como función *hash* usaremos SHA256 y RSA para validar las transacciones.

Cada bloque tendrá la estructura siguiente:

```
class block:
    def __init__(self):
        self.block_hash
        self.previous_block_hash
        self.transaction
        self.seed
```

siendo

- `block_hash` el SHA256 del bloque actual representado por un entero,
- `previous_block_hash` el SHA256 del bloque anterior representado por un entero,
- `transaction` una transacción válida,
- `seed` un entero.

Cada transacción será de la forma:

```
class transaction:
    def __init__(self, message=0, RSAkey=0):
        self.public_key
        self.message
        self.signature
```

siendo

- `public_key` la clave pública RSA correspondiente a `RSAkey`,
- `RSAkey` la clave RSA con la que se firma la transacción,
- `message` el documento que se firma en la transacción representado por un entero,
- `signature` es la firma de `message` hecha con la clave `RSAkey` representada por un entero.

Las claves privadas y públicas RSA serán de la forma:

```

class rsa_key:
    def __init__(self, bits_modulo=2048, e=2**16+1):
        self.publicExponent
        self.privateExponent
        self.modulus
        self.primeP
        self.primeQ
        self.privateExponentModulusPhiP
        self.privateExponentModulusPhiQ
        self.inverseQModulusP

```

siendo

- `publicExponent`, `privateExponent`, `modulus`, `primeP`, `primeQ` están representados por enteros,
- `privateExponentModulusPhiP` es congruente con `privateExponent` módulo `primeP-1` representado por un entero,
- `privateExponentModulusPhiQ` es congruente con `privateExponent` módulo `primeQ-1` representado por un entero,
- `inverseQModulusP` es el inverso de `primeQ` módulo `primeP` representado por un entero,

y

```

class rsa_public_key:
    def __init__(self, publicExponent=1, modulus=1):
        self.publicExponent = publicExponent
        self.modulus = modulus

```

siendo

- `publicExponent` el exponente público de la clave `rsa_key`,
- `modulus` el módulo de la clave `rsa_key`.

Una transacción es válida si

$$\text{signature}^{\text{publicExponent}} \equiv \text{message} \pmod{\text{modulus}}$$

Un bloque es válido si la transacción es válida y su *hash* h satisface la condición $h < 2^{256-d}$ siendo d un parámetro que indica el *proof of work* necesario par generar un bloque válido. Para esta práctica **d=16**.

Para calcular el *hash* h de un bloque haremos lo siguiente:

```

entrada=str(previous_block_hash)
entrada=entrada+str(transaction.public_key.publicExponent)
entrada=entrada+str(transaction.public_key.modulus)
entrada=entrada+str(transaction.message)
entrada=entrada+str(transaction.signature)
entrada=entrada+str(seed)
h=int(hashlib.sha256(entrada.encode()).hexdigest(),16)

```

Definid en **Python 3.x** las siguientes clases con los métodos descritos¹ (podéis definir otros si lo consideráis necesario):

```
• class rsa_key:
    def __init__(self, bits_modulo=2048, e=2**16+1):
        '''
        genera una clave RSA (de 2048 bits y exponente público 2**16+1 por defecto)
        '''
        self.publicExponent
        self.privateExponent
        self.modulus
        self.primeP
        self.primeQ
        self.privateExponentModulusPhiP
        self.privateExponentModulusPhiQ
        self.inverseQModulusP

    def __repr__(self):
        return str(self.__dict__)

    def sign(self, message):
        '''
        Salida: un entero que es la firma de "message" hecha con la clave RSA usando el TCR
        '''

    def sign_slow(self, message):
        '''
        Salida: un entero que es la firma de "message" hecha con la clave RSA sin usar el TCR
        '''

• class rsa_public_key:
    def __init__(self, publicExponent=1, modulus=1):
        '''
        genera la clave pública RSA asociada a la clave RSA "rsa_key"
        '''
        self.publicExponent
        self.modulus

    def __repr__(self):
        return str(self.__dict__)

    def verify(self, message, signature):
        '''
        Salida: el booleano True si "signature" se corresponde con la
                firma de "message" hecha con la clave RSA asociada a la clave
                pública RSA;
                el booleano False en cualquier otro caso.
        '''
```

¹Los métodos `__repr__(self)` ya están completamente definidos, no es necesario añadir nada, se usarán para guardar los objetos como diccionarios.

```

• class transaction:
    def __init__(self, message=0, RSAkey=0):
        '''
        genera una transaccion firmando "message" con la clave "RSAkey"
        '''
        self.public_key
        self.message
        self.signature

    def __repr__(self):
        return str(self.__dict__)

    def verify(self):
        '''
        Salida: el booleano True si "signature" se corresponde con la
                firma de "message" hecha con la clave RSA asociada a la clave
                pública RSA;
                el booleano False en cualquier otro caso.
        '''

• class block:
    def __init__(self):
        '''
        crea un bloque (no necesariamente válido)
        '''
        self.block_hash
        self.previous_block_hash
        self.transaction
        self.seed

    def __repr__(self):
        return str(self.__dict__)

    def genesis(self, transaction):
        '''
        genera el primer bloque de una cadena con la transacción "transaction"
        que se caracteriza por:
            - previous_block_hash=0
            - ser válido
        '''

    def next_block(self, transaction):
        '''
        genera un bloque válido siguiente al actual con la transacción "transaction"
        '''

```

```

def verify_block(self):
    '''
    Verifica si un bloque es válido:
    -Comprueba que el hash del bloque anterior cumple las condiciones exigidas
    -Comprueba que la transacción del bloque es válida
    -Comprueba que el hash del bloque cumple las condiciones exigidas

    Salida: el booleano True si todas las comprobaciones son correctas;
            el booleano False en cualquier otro caso.
    '''

```

```

• class block_chain:
    def __init__(self, transaction=0):
        '''
        genera una cadena de bloques que es una lista de bloques,
        el primer bloque es un bloque "genesis" generado con la transacción "transaction"
        '''
        self.list_of_blocks

    def __repr__(self):
        return str(self.__dict__)

    def add_block(self, transaction):
        '''
        añade a la cadena un nuevo bloque válido generado con la transacción "transaction"
        '''

    def verify(self):
        '''
        verifica si la cadena de bloques es válida:
        - Comprueba que todos los bloques son válidos
        - Comprueba que el primer bloque es un bloque "genesis"
        - Comprueba que para cada bloque de la cadena el siguiente es correcto

        Salida: el booleano True si todas las comprobaciones son correctas;
                en cualquier otro caso, el booleano False y un entero
                correspondiente al último bloque válido
        '''

```

2 RSA

2.1 Ron was wrong, Whit is right

Se recomienda encarecidamente leer el artículo:

”Ron was wrong, Whit is right”, <https://eprint.iacr.org/2012/064.pdf>.

En *Atenea* encontraréis directorio RSA_RW donde hay una serie de ficheros del tipo:

- *nombre.apellido_AES.enc* que es el resultado de cifrar un fichero determinado con la clave K
- *nombre.apellido_RSA_RW.enc* que es el resultado de cifrar la clave K con la clave pública RSA que se encuentra en el fichero *nombre.apellido_pubkeyRSA_RW.pem*.

El fichero cifrado se ha obtenido usando el comando:

```
openssl enc -e -aes-128-cbc -pbkdf2 -kfile fichero.key -in fichero.txt -out fichero.enc
```

El fichero *fichero.key* que contiene la clave se ha cifrado con el comando:

```
openssl pkeyutl -encrypt -inkey pubkeyRSA.pem -pubin -in fichero.txt -out fichero.enc
```

openssl está disponible en <https://www.openssl.org>. Se instala por defecto en la mayoría de las distribuciones de Linux, por ejemplo en la imagen Linux de la FIB.

Del fichero *nombre.apellido_pubkeyRSA_RW.pem* hay que extraer la clave pública (*openssl* puede ayudar), factorizar el módulo, calcular la clave privada, escribirla en un fichero en formato PEM (puede ser útil la biblioteca *Crypto.PublicKey.RSA* de *python*) aunque podéis encontrar otras para cualquier lenguaje de vuestra preferencia) y, para acabar, descifrar el fichero usando *openssl*.

2.2 Pseudo RSA

En *Atenea* también encontraréis el directorio RSA_pseudo donde hay una serie de ficheros semejantes a los anteriores.

Ahora el módulo público es un entero $n = pq$ con p y q tales que si en p es la concatenación de r y s de exactamente la mitad de bits de p , entonces q es, en binario, la concatenación de s y r . O sea que si $p = r||s$, entonces $q = s||r$ con $\#bits(r) = \#bits(s) = \frac{1}{2}\#bits(p) = \frac{1}{2}\#bits(q)$.

Del fichero *nombre.apellido_pubkeyRSA_pseudo.pem* hay que extraer la clave pública, factorizar el módulo, calcular la clave privada, escribirla en un fichero en format PEM y descifrar el fichero usando *openssl*.

3 Entrega

Un único fichero `zip`, `tar`,... con:

- **BlockChain:**
 - Un fichero que contenga todas las clases implementadas.
El nombre del fichero será `BlockChain_nombre1.apellido1_nombre2.apellido2.py` substituyendo `nombreX.apellidoX` por los nombres de los componentes del grupo.
 - Una tabla comparativa con el tiempo necesario para firmar, usando el TCR y sin usarlo, 100 mensajes diferentes con claves de 512, 1024, 2048 y 4096 bits. La tabla debe venir con una explicación de lo que se observa.
 - Un fichero, `BlockChain_nombre1.apellido1_nombre2.apellido2.block`, con una cadena válida de 100 bloques.
 - Un fichero, `FalseBlockChain_nombre1.apellido1_nombre2.apellido2.block`, con una cadena de 100 bloques que sólo sea válida hasta el bloque 33.

Los ficheros con las cadenas de bloques se han de generar con el siguiente código:

```
import json
with open(fichero_de_salida, 'w') as f:
    f.write(json.dumps(repr(cadena_de_bloques)))
```

Si queréis leer una cadena de bloques (ver Apéndice) contenida en el fichero anterior:

```
with open(fichero_de_salida, 'r') as f:
    cadena_de_bloques_diccionario = eval(json.loads(f.read()))
cadena_de_bloques = block_chain()
cadena_de_bloques.from_dictionary(cadena_de_bloques_diccionario)
```

- **RSA:**
 - Los ficheros descifrados.
 - Los ficheros descifrados con las claves secretas que se han usado para el AES.
 - Los ficheros en formato PEM con las claves privadas RSA.

Referencias

Sympy: Number Theory

[SymPy](#) is a Python library for symbolic mathematics.

[Welcome to SymPy's documentation!](#)

[Number Theory](#)

Sage

<http://www.sagemath.org>

SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.

<http://sagecell.sagemath.org>

<http://wiki.sagemath.org/quickref>

<http://wiki.sagemath.org/quickref?action=AttachFile&do=get&target=quickref-nt.pdf>

Algunos métodos para importar una cadena de bloques:

[illegible]

```
def from_dictionary(self, bloque):
    '''
    bloque = {
        'block_hash': 611227525515763553892040764593246705224095844323849655584941894507859918,
        'previous_block_hash': 860009111636437550099323966792787928396638877763118311905514989098990,
        'transaction': {
            'public_key': {
                'publicExponent': 65537,
```


[illegible]