RSA attacks : Code listings

Table des matières

1 Structure du code

- LICENCE
- README.md
- main.py
- modules
  - arithmetic.py
  - base.py
  - RSA_attacks.py
  - RSA.py
  - test_attacks.py
  - tests.py

2 Code

2.1 main.py

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

'''Main file running tests on the attacks'''

##-Import
from modules.test_attacks import *

from datetime import datetime as dt
from sys import argv
from sys import exit as sysexit

##-Run tests function
def run_tests(size=2048):
    '''Run the tests defined in the file `test_attacks.py`.'''

    passed = []

    t0 = dt.now()

    try:
        print('Launching tests...')
        print('-' * 16)

        print('1. Testing factorisation of the modulus with the private exponent :')
```

```python
        passed.append(test_mod_fact(size))
        print('-' * 16)

        print('2. Testing common modulus (finding the private exponent knowing a key set with the same exponent) :')
        passed.append(test_common_mod(size))
        print('-'*16)

        print('3. Testing multiplicative attack :')
        passed.append(test_multiplicative_attack(size))
        print('-'*16)

        print('4. Testing Hastad\'s attack (e = 5) :')
        passed.append(test_hastad(msg='Testing this attack with this message, because a message is needed.', size=size, e=5))
        print('-'*16)

        print('5. Testing Hastad\'s attack, testing the limit number of equations needed (e = 5, random message of length 100 characters) :')
        passed.append(test_hastad_message_size(size=size, e=5))
        print('-'*16)

        print('6. Testing Wiener\'s attack :')
        passed.append(test_wiener(size=size))
        print('-'*16)

        print('7. Testing Wiener\'s attack with a large private exponent :')
        passed.append(test_wiener(size=size, large=True))
        print('-'*16)

        print(f'\nDone in {dt.now() - t0}s.')

    except KeyboardInterrupt:
        print(f'\nStopped. Time elapsed : {dt.now() - t0}s.\nNumber of tests done : {len(passed)}')

    if not False in passed:
        print('\nAll tests passed correctly !')

    else:
        print('\nThe following tests failed :')

        for k, b in enumerate(passed):
            if not b:
                print(f'\t{k + 1}')

##-Run
if __name__ == '__main__':
    if len(argv) == 1:
        size = 2048

    else:
        try:
            size = int(argv[1])

        except:
            print(f'Wrong argument at position 1 : should be the RSA key size (in bits).\nExample : "{argv[0]} 2048".')
```

```
            sysexit()

    run_tests(size)
```

2.2 arithmetic.py

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

'''Useful arithmetic functions'''

##-Imports
from random import randint
from math import floor, ceil, sqrt, isqrt
from fractions import Fraction
from gmpy2 import is_square


##-Multiplicative inverse
def mult_inverse(a: int, n: int) -> int:
    '''
    Return the multiplicative inverse u of a modulo n.
    u*a = 1 modulo n
    '''

    (old_r, r) = (a, n)
    (old_u, u) = (1, 0)

    while r != 0:
        q = old_r // r
        (old_r, r) = (r, old_r - q*r)
        (old_u, u) = (u, old_u - q*u)

    if old_r > 1:
        raise ValueError(str(a) + ' is not inversible in the ring Z/' + str(n) + 'Z.')

    if old_u < 0:
        return old_u + n

    else:
        return old_u


##-Max parity
def max_parity(n):
    '''return (t, r) such that n = 2^t * r, where r is odd'''

    t = 0
    r = int(n)
    while r % 2 == 0 and r > 1:
        r //= 2
        t += 1

    return (t, r)


```

```python
##-Probabilistic prime test
def isSurelyPrime(n):
    '''Check if n is probably prime. Uses Miller Rabin test.'''

    if n == 2:
        return True

    elif n % 2 == 0:
        return False

    return miller_rabin(n, 15)


def miller_rabin_witness(a, d, s, n):
    '''
    Return True if a is a Miller-Rabin witness.

    - a : the base ;
    - d : odd integer verifying n - 1 = 2^s d ;
    - s : positive integer verifying n - 1 = 2^s d ;
    - n : the odd integer to test primality.
    '''

    r = pow(a, d, n)

    if r == 1 or r == n - 1:
        return False

    for k in range(s):
        r = r**2 % n

        if r == n - 1:
            return False

    return True


def miller_rabin(n, k=15) :
    '''
    Return the primality of n using Miller-Rabin probabilistic primality test.

    - n : odd integer to test the primality ;
    - k : number of tests (Error = 4^(-k)).
    '''

    if n in (0, 1):
        return False

    if n == 2:
        return True

    s, d = max_parity(n - 1)

    for i in range(k) :
        a = randint(2, n - 1)

        if miller_rabin_witness(a, d, s, n):
            return False
```

```

    return True


##-iroot
def iroot(n, k):
    '''
    Newton's method to find the integer k-th root of n.

    Return floor(n^(1/k))
    '''

    u, s = n, n + 1

    while u < s:
        s = u
        t = (k - 1) * s + n // pow(s, k - 1)
        u = t // k

    return s



##-Fermat factorisation
def fermat_factor(n):
    '''
    Try to factor n using Fermat's factorisation.
    For n = pq, works better if |q - p| is small, i.e if p and q
    are near sqrt(n).
    '''

    a = iroot(n, 2)

    while not is_square(pow(a, 2) - n):
        a += 1

        if pow(a, 2) - n <= 0:
            return False

    b = isqrt(pow(a, 2) - n)
    return (a - b, a + b)


##-Continued fractions
class ContinuedFraction:
    '''Class representing a continued fraction.'''

    def __init__(self, f):
        '''
        Initialize the class

        - f : the int array representing the continued fraction.
        '''

        if type(f) in (set, list):
            self.f = list(f)

        else:
```

```python
            raise ValueError('ContinuedFraction: error: `f` should be a list')

        if len(f) == 0:
            raise ValueError('ContinuedFraction: error: `f` should not be empty')

        for j, k in enumerate(f):
            if type(k) != int:
                raise ValueError(f'ContinuedFraction: error: `f` should be a list of int, but `{k}` found at position {j}')


    def __repr__(self):
        '''Return a pretty string representing the fraction.'''

        ret = f'{self.f[-1]}'

        for k in reversed(self.f[:-1]):
            ret = f'{k} + 1/(' + ret + ')'

        return ret


    def __eq__(self, other):
        '''Test the equality between self and the other.'''

        return self.f == other.f


    def eval_rec(self):
        '''Return the evaluation of self.f via a recursive function.'''

        return self._eval_rec(self.f)


    def _eval_rec(self, f_):
        '''The recursive function.'''

        if len(f_) == 1:
            return f_[0]

        return f_[0] + 1/(self._eval_rec(f_[1:]))


    def truncate(self, pos):
        '''
        Return a ContinuedFraction truncated at position `pos` from self.f.

        - pos : the position of the truncation. The element at position `pos` is kept in the result.
        '''

        return ContinuedFraction(self.f[:pos + 1])


    def get_convergents(self):
        '''
        Return two lists, p, q which represents the convergents :
```

```
        the n-th convergent is `p[n] / q[n]`.
        '''

        p = [0]*(len(self.f) + 2)
        q = [0]*(len(self.f) + 2)

        p[-1] = 1
        q[-2] = 1

        for k in range(0, len(self.f)):
            p[k] = self.f[k] * p[k - 1] + p[k - 2]
            q[k] = self.f[k] * q[k - 1] + q[k - 2]

        return p, q


    def eval_(self):
        '''Return the evaluation of self.f.'''

        p, q = self.get_convergents()

        return p[len(self.f) - 1] / q[len(self.f) - 1]


    def get_nth_convergent(self, n):
        '''Return the convergent at the index n.'''

        if n >= len(self.f):
            raise ValueError(f'ContinuedFraction: get_nth_convergent: n cannot
    be greater than {len(self.f) - 1}')

        p, q = self.get_convergents()

        return p[n] / q[n]



def get_continued_fraction(a, b):
    '''Return a ContinuedFraction object, the continued fraction of a/b.'''

    f = []
    d = Fraction(a, b)
    f.append(floor(d))

    while d - floor(d) != 0:
        d = 1/(d - floor(d))
        f.append(floor(d))

    return ContinuedFraction(f)


def get_continued_fraction_real(x):
    '''
    Return a ContinuedFraction object, the continued fraction of x.
    Note that there can be errors because of the float precision with this
    function.
    '''

```

```python
    f = []

    d = x
    f.append(floor(x))

    while d - floor(d) != 0:
        d = 1/(d - floor(d))
        f.append(floor(d))

    return ContinuedFraction(f)


def get_continued_fraction_rec(a, b, f=[]):
    '''Return a ContinuedFraction object, the continued fraction of a/b. This
    is a recursive function.'''

    # euclidean division : a = bq + r
    q = a // b
    r = a % b

    if r == 0:
        return ContinuedFraction(f + [q])

    return get_continued_fraction_rec(b, r, f + [q])


##-Tests
if __name__ == '__main__':
    if False:
        n = int(input('number :\n>'))
        p, q = fermat_factor(n)
        print('Result : {}\np = {}'.format(p*q == n, p))
```

2.3 base.py

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

'''Miscellaneous and useful functions'''

##-Imports
import hashlib


##-Split function
def split(txt, size, pad_=None):
    '''
    Return a list representing txt by groups of size `size`.

    - txt   : the text to split ;
    - size  : the block size ;
    - pad_  : if not None, pad the last block with `pad_` to be `size`length (
    adding to the end).
    '''

    l = []

```

```
    for k in range(len(txt) // size + 1):
        p = txt[k*size : (k+1)*size]

        if p in ('', b''):
            break

        if pad_ != None:
            p = pad(p, size, pad_)

        l.append(p)

    return l


def pad(txt, size, pad=' ', end=True):
    '''
    Pad `txt` to make it `size` long.
    If len(txt) > size, it just returns `txt`.

    - txt  : the string to pad ;
    - size : the final wanted size ;
    - pad  : the character to use to pad ;
    - end  : if True, add to the end, otherwise add to the beginning.
    '''

    while len(txt) < size:
        if end:
            txt += pad

        else:
            txt = pad + txt

    return txt


##-Mask generation function
# From https://en.wikipedia.org/wiki/Mask_generation_function
def i2osp(integer: int, size: int = 4) -> str:
    return int.to_bytes(integer % 256**size, size, 'big')

def mgf1(input_str: bytes, length: int, hash_func=hashlib.sha256) -> str:
    '''Mask generation function.'''

    counter = 0
    output = b''
    while len(output) < length:
        C = i2osp(counter, 4)
        output += hash_func(input_str + C).digest()
        counter += 1

    return output[:length]


##-Xor
def xor(s1, s2):
    '''Return s1 xored with s2 bit per bit.'''

    if (len(s1) != len(s2)):
```

```
        raise ValueError('Strings are not of the same length.')

    if type(s1) != bytes:
        s1 = s1.encode()

    if type(s2) != bytes:
        s2 = s2.encode()

    l = [i ^ j for i, j in zip(list(s1), list(s2))]

    return bytes(l)


##-Int and bytes
def int_to_bytes(x: int) -> bytes:
    return x.to_bytes((x.bit_length() + 7) // 8, 'little')

def bytes_to_int(xbytes: bytes) -> int:
    return int.from_bytes(xbytes, 'little')


##-Other
def str_diff(s1, s2, verbose=True, max_len=80):
    '''
    Show difference between strings (or numbers) s1 and s2. Return s1 == s2.

    - s1      : input string to compare ;
    - s2      : output string to compare ;
    - verbose : if True, show input and output message and where they differ if so ;
    - max_len : don't show messages if their length is more than max_len. Default is 80. If negative, always show them.
    '''

    s1 = str(s1)
    s2 = str(s2)

    if verbose:
        if len(s1) <= max_len or max_len == -1:
            print(f'\nEntry message : {s1}')
            print(f'Output        : {s2}')

        for k in range(len(s1)):
            if s1[k] != s2[k]:
                if len(s1) <= max_len or max_len == -1:
                    print(' '*(len('Output        : ') + k) + '^')

                print('Input and output differ from position {}.'.format(k))

                return False

        print('Input and output are identical.')

    return s1 == s2


##-Testing
if __name__ == '__main__':
```

```python
    msg = input('msg\n>').encode()

    print(mgf1(msg, 10).hex())
    print(xor('test', 'abcd'))
```

2.4 RSA.py

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

'''Implementation of RSA cipher and key management'''

##-Imports
try:
    from arithmetic import *
    from base import *

except ModuleNotFoundError:
    from modules.arithmetic import *
    from modules.base import *

from secrets import randbits
from random import randint, randbytes
import math

import base64

##-RsaKeys
class RsaKey:
    '''RSA key object'''

    def __init__(self, e=None, d=None, n=None, phi=None, p=None, q=None):
        '''
        - e : public exponent
        - d : private exponent
        - n : modulus
        - p, q : primes that verify pq = n
        - phi = (p - 1)(q - 1)
        '''

        self.e = e
        self.d = d
        self.n = n
        self.phi = phi
        self.p = p
        self.q = q

        self.is_private = self.d != None

        if self.is_private:
            if self.q < self.q:
                self.p = q
                self.q = p

        self.pb = (e, n)
        if self.is_private:
            self.pv = (d, n)
```

```python

        self.size = None

    def __repr__(self):
        if self.is_private:
            return f'RsaKey private key :\n\tsize : {self.size}\n\te : {self.e}\n\td : {self.d}\n\tn : {self.n}\n\tphi : {self.phi}\n\tp : {self.p}\n\tq : {self.q}'

        else:
            return f'RsaKey public key :\n\tsize : {self.size}\n\te : {self.e}\n\tn : {self.n}'


    def __eq__(self, other):
        '''Return True if the key are of the same type (public / private) and have the same values.'''

        ret = self.is_private == other.is_private

        if not ret:
            return False

        if self.is_private:
            ret = ret and (
                self.e == other.e and
                self.d == other.d and
                self.n == other.n and
                self.phi == other.phi
            )

            ret = ret and ((self.p == other.p and self.q == other.q) or (self.q == other.p and self.p == other.q))

        else:
            ret = ret and (
                self.e == other.e and
                self.n == other.d
            )

        return ret


    def public(self):
        '''Return the public key associated to self in an other RsaKey object.'''

        k = RsaKey(e=self.e, n=self.n)
        k.size = self.size
        return k


    def _gen_nb(self, size=2048, wiener=False):
        '''
        Generates p, q, and set attributes p, q, phi, n, size.

        - size   : the bit size of n ;
        - wiener : If True, generates p, q prime such that q < p < 2q.
```

```
        '''

        self.p, self.q = 1, 1

        while not isSurelyPrime(self.q):
            self.q = randbits(size // 2)

        while not (isSurelyPrime(self.p) and ((wiener and self.q < self.p < 2 * self.q) or (not wiener))):
            self.p = randbits(size // 2)

        self.phi = (self.p - 1) * (self.q - 1)
        self.n = self.p * self.q

        self.size = size


    def new(self, size=2048):
        '''
        Generate RSA keys of size `size` bits.
        If self.e != None, it keeps it (and ensures that gcd(phi, e) = 1).

        - size : the key size, in bits.
        '''

        self._gen_nb(size)

        while self.e != None and math.gcd(self.e, self.phi) != 1:
            self._gen_nb(size)

        if self.e == None:
            self.e = 0
            while math.gcd(self.e, self.phi) != 1:
                self.e = randint(max(self.p, self.q), self.phi)

        elif math.gcd(self.e, self.phi) != 1: #Not possible !
            raise ValueError('RsaKey: new: error: gcd(self.e, self.phi) != 1')

        self.d = mult_inverse(self.e, self.phi)

        self.is_private = True

        self.pb = (self.e, self.n)
        self.pv = (self.d, self.n)

        self.size = size


    def new_wiener(self, size=2048):
        '''
        Generate RSA keys of size `size` bits.
        If self.e != None, it does NOT keeps it.
        These key are generated so that the Wiener's attack is possible on them.

        - size : the key size, in bits.
        '''

```

```
        self._gen_nb(size, wiener=True)

        self.d = 0
        while math.gcd(self.d, self.phi) != 1:
            self.d = randint(1, math.floor(isqrt(isqrt(self.n))/3))

        self.e = mult_inverse(self.d, self.phi)

        self.is_private = True

        self.pb = (self.e, self.n)
        self.pv = (self.d, self.n)

        self.size = size


    def new_wiener_large(self, size=2048, only_large=True):
        '''
        Same as `self.new_wiener`, but `d` can be very large.

        - size        : the RSA key size ;
        - only_large : if False, d can be small, or large, and otherwise, d is large.
        '''

        self._gen_nb(size, wiener=True)

        self.d = 0
        while math.gcd(self.d, self.phi) != 1:
            if only_large:
                #ceil(sqrt(6)) = 3
                self.d = randint(int(self.phi - iroot(self.n, 4) // 3), self.phi)

            else:
                self.d = randint(1, self.phi)
                if iroot(self.n, 4) / 3 < self.d or self.d < self.phi - iroot(self.n, 4) / math.sqrt(6):
                    self.d = 0 #go to the next iteration

        self.e = mult_inverse(self.d, self.phi)
        self.is_private = True
        self.pb = (self.e, self.n)
        self.pv = (self.d, self.n)

        self.size = size



##-Padding
class OAEP:
    '''Class implementing OAEP padding'''

    def __init__(self, block_size, k0=None, k1=0):
        '''
        Initiate OAEP class.

        - block_size   :    the bit size of each block ;
```

```
        - k0            :     integer (number of bits in the random part). If
    None, it is set to block_size // 8 ;
        - k1            :     integer such that len(block) + k0 + k1 = block_size
    . Default is 0.
        '''

        self.block_size = block_size #n

        if k0 == None:
            k0 = block_size // 8

        self.k0 = k0
        self.k1 = k1


    def _encode_block(self, block):
        '''
        Encode a block.

        - block : an n - k0 - k1 long bytes string.
        '''

        #---Add k1 \0 to block
        block += (b'\0')*self.k1

        #---Generate r, a k0 bits random string
        r = randbytes(self.k0)

        X = xor(block, mgf1(r, self.block_size - self.k0))

        Y = xor(r, mgf1(X, self.k0))

        return X + Y


    def encode(self, txt):
        '''
        Encode txt

        Entry :
            - txt : the string text to encode.

        Output :
            bytes list
        '''

        if type(txt) != bytes:
            txt = txt.encode()

        #---Cut message in blocks of size n - k0 - k1
        blocks = []
        l = self.block_size - self.k0 - self.k1

        blocks = split(txt, l, pad_=b'\0')

        #---Encode blocks
        enc = []
        for k in blocks:
```

```
            enc.append(self._encode_block(k))

        return enc


    def _decode_block(self, block):
        '''Decode a block encoded with self._encode_block.'''

        X = block[:self.block_size - self.k0]
        Y = block[-self.k0:]

        r = xor(Y, mgf1(X, self.k0))

        txt = xor(X, mgf1(r, self.block_size - self.k0))

        while txt[-1] == 0: #Remove padding
            txt = txt[:-1]

        return txt


    def decode(self, enc):
        '''
        Decode a text encoded with self.encode.

        - enc : a list of bytes encoded blocks.
        '''

        txt = b''

        for k in enc:
            txt += self._decode_block(k)

        return txt



#-RSA
class RSA:
    '''RSA cipher'''

    def __init__(self, key, padding, block_size=None):
        '''
        - key        : a RsaKey object ;
        - padding    : the padding to use. Possible values are :
            'int' : msg is an int, return an int ;
            'raw' : msg is a string, simply cut it in blocks ;
            'oaep' : OAEP padding ;
        - block_size : the size of encryption blocks. If None, it is set to `key.size // 8 - 1`.
        '''

        self.pb = key.pb
        if key.is_private:
            self.pv = key.pv

        self.is_private = key.is_private

```

```
        if padding.lower() not in ('int', 'raw', 'oaep'):
            raise ValueError('RSA: padding not recognized.')

        self.pad = padding.lower()

        if block_size == None:
            self.block_size = key.size // 8 - 1

        else:
            self.block_size = block_size


    def encrypt(self, msg):
        '''
        Encrypt `msg` using the key given in init.
        Redirect toward the right method (using the good padding).

        - msg      : The string to encrypt.
        '''

        if self.pad == 'int':
            return self._encrypt_int(msg)

        elif self.pad == 'raw':
            return self._encrypt_raw(msg)

        else:
            return self._encrypt_oaep(msg)


    def decrypt(self, msg):
        '''
        Decrypt `msg` using the key given in init, if it is a private one.
    Otherwise raise a TypeError.
        Redirect toward the right method (using the good padding).
        '''

        if not self.is_private:
            raise TypeError('Can not decrypt using a public key.')

        if self.pad == 'int':
            return self._decrypt_int(msg)

        elif self.pad == 'raw':
            return self._decrypt_raw(msg)

        else:
            return self._decrypt_oaep(msg)


    def _encrypt_int(self, msg):
        '''
        RSA encryption in its simplest form.

        - msg : an integer to encrypt.
        '''

        e, n = self.pb
```

```
        return pow(msg, e, n)


    def _decrypt_int(self, msg):
        '''
        RSA decryption in its simplest form.
        Decrypt `msg` using the key given in init if possible, using the 'int' padding.

        - msg : an integer.
        '''

        d, n = self.pv

        return pow(msg, d, n)


    def _encrypt_raw(self, msg):
        '''
        Encrypt `msg` using the key given in init, using the 'raw' padding.

        - msg : The string to encrypt
        '''

        e, n = self.pb

        #---Encode msg
        if type(msg) != bytes:
            msg = msg.encode()

        #---Cut message in blocks
        m_lst = split(msg, self.block_size)

        #---Encrypt message
        enc_lst = []
        for k in m_lst:
            enc_lst.append(pow(bytes_to_int(k), e, n))

        return b' '.join([base64.b64encode(int_to_bytes(k)) for k in enc_lst])


    def _decrypt_raw(self, msg):
        '''Decrypt `msg` using the key given in init if possible, using the 'raw' padding'''

        d, n = self.pv

        enc_lst = [base64.b64decode(k) for k in msg.split(b' ')]

        c_lst = []
        for k in enc_lst:
            c_lst.append(pow(bytes_to_int(k), d, n))

        txt = b''
        for k in c_lst:
            txt += int_to_bytes(k)

```

```
            return txt.decode()


    def _encrypt_oaep(self, msg):
        '''Encrypt `msg` using the key given in init, using the 'oaep' padding.
    '''

        e, n = self.pb

        if type(msg) != bytes:
            msg = msg.encode()

        #---Padding
        E = OAEP(self.block_size)
        m_lst = E.encode(msg)

        #---Encrypt message
        enc_lst = []
        for k in m_lst:
            enc_lst.append(pow(bytes_to_int(k), e, n))

        return b' '.join([base64.b64encode(int_to_bytes(k)) for k in enc_lst])


    def _decrypt_oaep(self, msg):
        '''Decrypt `msg` using the key given in init if possible, using the '
    oaep' padding.'''

        d, n = self.pv

        #---Decrypt
        enc_lst = [base64.b64decode(k) for k in msg.split(b' ')]
        c_lst = []

        for k in enc_lst:
            c_lst.append(pow(bytes_to_int(k), d, n))

        #---Decode
        encoded_lst = []
        for k in c_lst:
            encoded_lst.append(pad(int_to_bytes(k), self.block_size, b'\0'))

        E = OAEP(self.block_size)

        return E.decode(encoded_lst)


##-Testing
if __name__ == '__main__':
    from tests import test_OAEP, test_RSA, dt
    from sys import argv, exit as sysexit

    if len(argv) == 1:
        size = 2048

    else:
        try:
            size = int(argv[1])
```

```

        except:
            print(f'Wrong argument at position 1 : should be the RSA key size (
    in bits).\nExample : "{argv[0]} 2048".')
            sysexit()

    t0 = dt.now()
    print('Generating a key (for all the tests) ...')
    k = RsaKey()
    k.new(size)
    print('Done.')

    test_OAEP(size // 8 - 1)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_RSA(k, 'int', size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_RSA(k, 'raw', size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_RSA(k, 'oaep', size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
```

2.5 RSA_attacks.py

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

'''Implementation of RSA attacks'''

##-Imports
try:
    from arithmetic import *
    import RSA

except ModuleNotFoundError:
    from modules.arithmetic import *
    import modules.RSA as RSA

import math
from random import randint

from datetime import datetime as dt


##-Elementary attacks
#------Elementary attacks
#---Factor modulus with private key
def factor_with_private(e, d, n, max_tries=None):
    '''
    Factor modulus n using public and private exponent e and d.

    - max_tries : stop after `max_tries` tries if not found before. If None,
    don't stop until found.
    '''

    k = e*d - 1
    t, r = max_parity(k) # k = 2^t * r, r is odd.

```

```
    i = 0
    while True:
        g = 0
        while math.gcd(g, n) != 1: # find a g in (Z/nZ)^*
            g = randint(2, n - 1)

        for j in range(t, 1, -1): # Try with g^(k / 2^j)
            x = pow(g, k // (2**j), n)
            y = math.gcd(x - 1, n)

            if n % y == 0 and (y not in (1, n)):
                return y, n//y

        if max_tries != None:
            i += 1
            if i >= max_tries:
                return None


#---Common modulus
def common_modulus(N, e, d, e1):
    '''
    Entry :
        - N  : the common modulus ;
        - e  : the known public exponent ;
        - d  : the known private exponent ;
        - e1 : public exponent associated to the wanted private exponent.

    Calculate d1 the private exponent associated to e1.
    '''

    p, q = factor_with_private(e, d, N)
    phi = (p - 1) * (q - 1)

    return mult_inverse(e1, phi)


#---Multiplicative attack
def multiplicative_attack(m_, r, n):
    '''
    Uses the fact that the product of two ciphertexts is equal to the
    ciphertext of the product.

    We have c = m^e [n] and we want m.
    We ask for the decryption of c_ = c * r^e [n] (m_).

    - m_ : the decryption of c_ = c * r^e [n] ;
    - r  : the number used to obfuscate the initial message ;
    - n  : the modulus.
    '''

    inv_r = mult_inverse(r, n)

    return (m_ * inv_r) % n


#------Large message (close to n)
def large_message(c, e, n):
```

```
    '''
    Return the decryption of c using the method from Hinek's paper (cacr2004).
    In order for this attack to work, we need to have
        n - n^(1/e) < m < n
    Then we have :
        m = n - (-c % n)^(1/e).

    Arguments :
        - c : the encryption of m : c = m^e [n] ;
        - e : the public exponent ;
        - n : the RSA modulus.
    '''

    return n - iroot(-c % n, e)


##-Hastad
#---Hastad (same message)
def _hastad(e, enc_msg_lst, mod_lst):
    '''
    Return (me, e, M). The decrypted message is `iroot(me, e)` or `large_message(me, e, M)` (if the message was very long).

    - e           : the common public exponent ;
    - enc_msg_lst : the list of the encrypted messages ;
    - mod_lst     : the list of modulus.

    The lists `enc_msg_lst` and `mod_lst` should have the same length.
    '''

    M = 1
    for k in mod_lst:
        M *= k

    me = sum(enc_msg_lst[k] * (M // mod_lst[k]) * mult_inverse(M // mod_lst[k], mod_lst[k]) for k in range(len(mod_lst))) % M

    return (me, e, M)


def hastad(e, enc_msg_lst, mod_lst):
    '''
    Return the decrypted message.

    - e           : the common public exponent ;
    - enc_msg_lst : the list of the encrypted messages ;
    - mod_lst     : the list of modulus.

    The lists `enc_msg_lst` and `mod_lst` should have the same length.
    '''

    me, e = _hastad(e, enc_msg_lst, mod_lst)[:-1]

    return iroot(me, e)


def hastad_large_message(e, enc_msg_lst, mod_lst):
    '''
```

```
    Return the decrypted message.

    - e           : the common public exponent ;
    - enc_msg_lst : the list of the encrypted messages ;
    - mod_lst     : the list of modulus.

    The lists `enc_msg_lst` and `mod_lst` should have the same length.
    '''

    me, e, M = _hastad(e, enc_msg_lst, mod_lst)

    return large_message(me, e, M)


##-Wiener's attack
def factor_with_phi(n, phi):
    '''
    Return (p, q) such that n = pq, if possible. Otherwise, raise a ValueError

    - n   : the RSA modulus ;
    - phi : the Euler totien of n : phi = (p - 1)(q - 1).

    It solve the quadratic
        x^2 - (n - phi + 1)x + n = 0
    '''

    delta = (n - phi + 1)**2 - 4*n

    if delta < 0:
        raise ValueError('Wrong modulus or wrong phi.')

    p = (n - phi + 1 - isqrt(delta)) // 2
    q = (n - phi + 1 + isqrt(delta)) // 2

    if p * q != n:
        raise ValueError('Wrong modulus or wrong phi.')

    return p, q


def wiener(e, n):
    '''
    Run Wiener's attack on the public key (e, n).
    Return a private RsaKey object.

    Can factor the key if the private exponent d is such that
        1 < d < n^(1/4) / 3
        or
        phi - n^(1/4)/sqrt(6) < d < phi

    - e : the public exponent ;
    - n : the modulus.
    '''

    #---Calculate the continued fraction of e/n
    e_n_frac = get_continued_fraction(e, n)

    #---Calculate the convergents
```

```python
    k_, d_ = e_n_frac.get_convergents()

    #---Compute phi to check correctness
    for i in range(1, len(k_) - 2):
        phi = (e * d_[i] - 1) // k_[i]
        phi2 = (e * d_[i] + 1) // k_[i] #With large private exponent.

        try:
            p, q = factor_with_phi(n, phi)

        except ValueError:
            try:
                p2, q2 = factor_with_phi(n, phi2)

            except ValueError:
                continue

            else: #Correct factorisation with p2, q2
                key = RSA.RsaKey(e, phi2 - d_[i], n, phi2, p2, q2)
                return key

        else: #Correct factorisation with p, q
            key = RSA.RsaKey(e, d_[i], n, phi, p, q)
            return key

    raise ValueError('The attack failed with this key')
```

2.6 test_attacks.py

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

'''Tests for RSA attacks'''

##-imports
try:
    from RSA_attacks import *
    from base import str_diff, int_to_bytes, bytes_to_int

except ModuleNotFoundError:
    from modules.RSA_attacks import *
    from modules.base import str_diff, int_to_bytes, bytes_to_int

from secrets import randbits
from random import randint

##-Fermat factorisation
def test_fermat_factor(size=2048, dist=512):
    '''
    Tests the Fermat factorisation : generates two primes p, q and test the algorithm on it.

    - size : the size of the modulus to generate in bits, i.e of p*q ;
    - dist : the bit size of |p - q| ;
    '''

    print('Prime generation ...')
```

```
    t0 = dt.now()

    p = 1
    while not isSurelyPrime(p):
        p = randbits(size // 2)

    q = p + 2**dist
    while not isSurelyPrime(q):
        q +=1

    print(f'Generation done in {dt.now() - t0}s.\np : {round(math.log2(p), 2)} bits\nq : {round(math.log2(q), 2)} bits\n|p - q| : {round(math.log2(q - p), 2)} bits\n2 * |p - q|^(1/4) : {round(math.log2(2 * iroot(p * q, 4)), 2)}')

    b = q - p <= 2 * iroot(p * q, 4)

    print('\nFactorisation ...')
    t1 = dt.now()
    a, b = fermat_factor(p * q)
    print(f'Factorisation done in {dt.now() - t1}s.')

    if a * b != p * q:
        print('Factorisation failed : the product of the result is not p * q.')
        return False

    if not (p in (a, b) and q in (a, b)):
        print('Factorisation failed : p or q not in the result.')
        return False

    print('Good factorisation.')

    return True


##-Modulus factorisation
def test_mod_fact(size=2048):
    print('Key generation ...')
    t0 = dt.now()
    key = RSA.RsaKey()
    key.new(size)
    print(f'Generation done in {dt.now() - t0}s.')

    t1 = dt.now()
    try:
        p, q = factor_with_private(key.e, key.d, key.n)

    except TypeError:
        print('not found !')
        return False

    else:
        print('Found in {}.\nCorrect : n == pq : {}, key.p in (p, q) : {}'.format(dt.now() - t1, key.n == p*q, key.p in (p, q)))
        return True

        # for k in (key.p, key.q, p, q):
        #     print(k)

```

```python
##-Common modulus
def test_common_mod(size=2048):
    print('Key generation ...')
    t0 = dt.now()
    key = RSA.RsaKey()
    key.new(size)
    print(f'Generation done in {dt.now() - t0}s.')

    t1 = dt.now()
    e1 = 0
    while math.gcd(e1, key.phi) != 1:
        e1 = randint(max(key.p, key.q), key.phi)

    print(f'Generation of e1 done in {dt.now() - t1}s.')

    t2 = dt.now()
    d1 = mult_inverse(e1, key.phi)
    if common_modulus(key.n, key.e, key.d, e1) == d1:
        print(f'Attack succeeded : private exposant recovered.\nDone in {dt.now() - t2}s.')
        return True
    else:
        print(f'Attack failed : private exposant NOT recovered.\nDone in {dt.now() - t2}s.')
        return False


##-Test Multiplicative attack
def test_multiplicative_attack_one_block(m=None, size=2048):
    '''
    Test multiplicative_attack.

    - m    : the message (int). If None, generates a random one ;
    - size : the RSA key size.
    '''

    t0 = dt.now()
    print('Key generation ...')
    key = RSA.RsaKey()
    key.new(size)

    n = key.n
    e = key.e
    d = key.d

    print(f'Done in {dt.now() - t0}s')

    if m == None:
        m = randint(1, n - 1)

    c = pow(m, e, n)

    t1 = dt.now()
    print('Running the attack ...')
    r = randint(2, n - 1)

    if math.gcd(r, n) != 1: # To ensure that r is inversible modulo n
        p = math.gcd(r, n)
```

```
            q = n // p
            print(f'We accidentally factorized n ...\nn = {n}\np = {p}\nq = {q}.\nn == p*q : {n == p * q}.')
            return n == p * q

    c_ = (c * pow(r, e, n)) % n #obfuscated encrypted message
    m_ = pow(c_, d, n) #The inoffensive looking message (obfuscated) gently decrypted by Alice

    recov_m = multiplicative_attack(m_, r, n)

    print(f'Attack done in {dt.now() - t1}s.')

    if recov_m == m:
        print('Attack successful')
        return True

    else:
        print('Attack failed')
        return False


def test_multiplicative_attack(m=None, size=2048):
    '''
    Test multiplicative_attack.

    - m    : the message (int). If None, generates a random one ;
    - size : the RSA key size.
    '''

    t0 = dt.now()
    print('Key generation ...')
    key = RSA.RsaKey()
    key.new(size)

    n = key.n
    e = key.e
    d = key.d

    print(f'Done in {dt.now() - t0}s')

    if m == None:
        m = randint(1, n - 1)

    E = RSA.OAEP(key.size // 8 - 1)
    m_e = [bytes_to_int(k) for k in E.encode(int_to_bytes(m))] #message encoded in blocks
    enc_lst = [pow(k, e, n) for k in m_e] #The ciphertexts

    t1 = dt.now()
    print('Running the attack ...')
    r_lst = [randint(2, n - 1) for k in range(len(m_e))] #choose one r per block

    for r in r_lst:
        if math.gcd(r, n) != 1: # To ensure that all r are inversible modulo n
            p = math.gcd(r, n)
            q = n // p
```

```python

                print(f'We accidentally factorized n ...\nn = {n}\np = {p}\nq = {q}.\nn == p*q : {n == p * q}.')

                return n == p * q

        enc_lst_r = [(c_k * pow(r_k, e, n)) % n for (c_k, r_k) in zip(enc_lst, r_lst)] #List of obfuscated encrypted messages

        dec_lst = [pow(k, d, n) for k in enc_lst_r] #The inoffensive looking messages (obfuscated) gently decrypted by Alice

        recov_lst = [multiplicative_attack(m_k, r_k, n) for (m_k, r_k) in zip(dec_lst, r_lst)]

        decoded = E.decode([int_to_bytes(k) for k in recov_lst])

        print(f'Attack done in {dt.now() - t1}s.')

        if bytes_to_int(decoded) == m:
            print('Attack successful')
            return True

        else:
            print('Attack failed')
            return False


##-Test large positive numbers
def test_large_message(e=3, size=2048, verbose=False):
    '''
    Cf cacr2004 (Hinek) paper.
    Generates an RSA key, and a message m such that n - n^(1/e) < m < n
    Then encrypt it : c = m^e [n]
    It is possible to recover the message :
        m = n - (-c % n)^(1/e)
    '''

    print('Generating RSA key ...')
    t0 = dt.now()

    k = RSA.RsaKey(e = e)
    k.new(size)
    print(f'Generation done in {dt.now() - t0}s.')

    print('Generating message and encrypting it ...')
    t1 = dt.now()
    m = randint(k.n - iroot(k.n, e), k.n)
    c = pow(m, e, k.n)
    print(f'Done in {dt.now() - t1}s.')

    print('Recovering the message ...')
    t2 = dt.now()
    m_recov = large_message(c, k.e, k.n)
    print(f'Message recovered in {dt.now() - t2}s.')

    if str_diff(str(m), str(m_recov), verbose=verbose, max_len=-1):
        print(f'Attack successful. Done in {dt.now() - t0}s.')
```

```
        return True

    else:
        print(f'Attack failed. Time elapsed : {dt.now() - t0}s.')
        return False


##-Hastad
def test_hastad(msg = 'testing', e=3, size=2048, nb_eq=None, try_large=False):
    '''
    Tests the `hastad` function.

    - msg       : the message that will be encrypted with RSA and be recovered
    ;
    - e         : the public exponent used for all the keys ;
    - size      : the size of the modulus ;
    - nb_eq     : the number of equations. If None, calculate the right number
    using the message ;
    - try_large : bool indicating if trying to break the message using
    hastad_large_message.
    '''

    msg = int(''.join(format(ord(k), '03') for k in msg)) #testing ->
    116101115116105110103

    n = math.ceil(e * math.log2(msg) / size)
    print(f'Number of equations actually needed to recover the message : {n}.')

    if nb_eq == None:
        nb_eq = n

    keys = [RSA.RsaKey(e=e) for k in range(nb_eq)]

    print(f'\nKey generation for Hastad\'s attack ({size} bits, {nb_eq} keys)
    ...')
    t0 = dt.now()
    for k in range(nb_eq):
        t1 = dt.now()
        keys[k].new(size)
        print(f'{k + 1}/{nb_eq} generated in {dt.now() - t1}s.')

    print(f'Done in {dt.now() - t0}s.')

    mod_lst = [keys[k].n for k in range(nb_eq)]
    ciphers = [RSA.RSA(keys[k], 'int') for k in range(nb_eq)]
    enc_lst = [ciphers[k].encrypt(msg) for k in range(nb_eq)]

    print('\nHastad attack ...')
    t2 = dt.now()
    ret = hastad(e, enc_lst, mod_lst)
    print(f'Attack done in {dt.now() - t2}s.')

    dec_out = ''.join([chr(int(str(ret)[3*k : 3*k + 3])) for k in range(len(str
    (ret)) // 3)])

    if msg != ret and try_large:
        # This can't work because no message can fit in [M - M^(1/e) ; M] :
    they would need to have exactly len(str(M))/k = len(str(M - iroot(M, e)))/k
```

```
    characters (where k is defined with the encoding used (here it is k = 3)) so
     we need that k divide len(str(M)) (thus that way it is possible to find an
    int of this length that will thus maybe correspond to an encoded message).

        print('Attack failed, trying to use the large number way ...')

        M = 1
        for k in range(nb_eq):
            M *= keys[k].n

        print(f'Is the condition good for large number attack ? : {M - iroot(M,
     e) <= msg <= M}')
        if M - iroot(M, e) > msg:
            print('Message is too small for the large message attack.')

        elif msg > M:
            print('Message is too large for the large message attack.')

        t3 = dt.now()
        ret2 = hastad_large_message(e, enc_lst, mod_lst)
        print(f'Attack done in {dt.now() - t3}s.')

        return str_diff(msg, ret2)

    return str_diff(msg, ret)

    #print(f'\nDecoded output :\n{dec_out}')

#-Test message size limit
def test_hastad_message_size(msg_size=100, e=3, size=2048):
    '''
    Test the number size with the number of equations

    - msg_size : the length of the message ;
    - e        : the public exponent used for all the keys ;
    - size     : the size of the modulus.
    '''

    msg = ''.join([chr(randint(65, 122)) for k in range(msg_size)]) #Random
    chars
    msg = int(''.join(format(ord(k), '03') for k in msg)) #Encoding the message

    n = math.ceil(e * math.log2(msg) / size)
    print(f'Number of equations theoretically needed to recover the message : {
    n}.')

    keys = [RSA.RsaKey(e=e) for k in range(n)]

    print(f'\nKey generation for Hastad\'s attack ({size} bits) ...')
    t0 = dt.now()
    for k in range(n):
        t1 = dt.now()
        keys[k].new(size)
        print(f'{k + 1}/{n} generated in {dt.now() - t1}s.')

    print(f'Done in {dt.now() - t0}s.')

    mod_lst = [keys[k].n for k in range(n)]
```

```
    ciphers = [RSA.RSA(keys[k], 'int') for k in range(n)]
    enc_lst = [ciphers[k].encrypt(msg) for k in range(n)]

    print(f'\nHastad attack with {n} equations ...')
    t2 = dt.now()
    ret1 = hastad(e, enc_lst, mod_lst)
    print(f'Attack done in {dt.now() - t2}s.')

    if msg == ret1:
        print('Attack succeeded : message correctly recovered.')

    else:
        print('Attack failed : message NOT correctly recovered.')
        return False

    if n - 1 == 0:
        print('\nNot trying to with less equations than one.')
        return True

    print(f'\nHastad attack with {n - 1} equations ...')
    t3 = dt.now()
    ret2 = hastad(e, enc_lst[:-1], mod_lst[:-1])
    print(f'Attack done in {dt.now() - t3}s.')

    if msg == ret2:
        print('Attack succeeded : message correctly recovered. So the limit is NOT correct.')
        return False

    else:
        print('Attack failed : message not correctly recovered. So the limit is correct.')
        return True


def test_hastad_large_message(e=3, size=2048, less=0):
    '''
    Tests the `hastad` function, with large message (see Hinek's paper).

    - e    : the public exponent used for all the keys ;
    - size : the size of the modulus ;
    - less : the number of equations to remove.

    But the problem with this is that it generates the message after having M, which is not how it would be in real life.
    '''

    keys = [RSA.RsaKey(e=e) for k in range(e)]

    print(f'\nKey generation for Hastad\'s attack ({size} bits) ...')
    t0 = dt.now()
    for k in range(e):
        t1 = dt.now()
        keys[k].new(size)
        print(f'{k + 1}/{e} generated in {dt.now() - t1}s.')

    print(f'Done in {dt.now() - t0}s.')

```

```
    M = 1
    for k in range(e):
        M *= keys[k].n

    msg = randint(M - iroot(M, e), M)
    print('len(str(msg)) :', len(str(msg)), 'log2(M) :', math.log2(M))
    print(f'Number of equations actually needed to recover the message (without
     large message idea) : {math.ceil(e * math.log2(msg) / size)}.')
    #print(f'msg : {msg}')

    mod_lst = [keys[k].n for k in range(e - less)]
    ciphers = [RSA.RSA(keys[k], 'int') for k in range(e - less)]
    enc_lst = [ciphers[k].encrypt(msg) for k in range(e - less)]

    print('\nHastad attack ...')
    t2 = dt.now()
    ret = hastad_large_message(e, enc_lst, mod_lst)
    print(f'Attack done in {dt.now() - t2}s.')

    if str_diff(str(msg), str(ret)):
        return True

    else:
        return False


##-Wiener
def test_wiener(size=2048, large=False, not_in_good_condition=False):
    '''
    Test Wiener's attack.

    - size                  : The RSA key size ;
    - large                 : if True, generates a large private exponent ;
    - not_in_good_condition : Do not try to generate a key that is breakable
   with this attack.
    '''

    key = RSA.RsaKey()

    print(f'Key generation for Wiener\'s attack ({size} bits) ...')
    t0 = dt.now()
    if not_in_good_condition:
        key.new()

    elif large:
        key.new_wiener_large(size)

    else:
        key.new_wiener(size)

    print(f'Key generated in {dt.now() - t0}s.')

    pb = key.public()

    t1 = dt.now()
    try:
        recovered_key = wiener(pb.e, pb.n)

```

```
    except ValueError as err:
        print(f'Wiener\'s attack finished in {dt.now() - t1}s.')
        print(err)
        return False

    print(f'Wiener\'s attack finished in {dt.now() - t1}s.')

    if recovered_key == key:
        print('Correct result !')
        return True

    else:
        print('Incorrect result !')
        return False

if __name__ == '__main__':
    # test_wiener(large=True)
    test_multiplicative_attack()
```

2.7 tests.py

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

'''Tests'''

##-Import
try:
    from base import *
    from arithmetic import *
    from RSA_attacks import *
    from RSA import *
    import test_attacks

except ModuleNotFoundError:
    from modules.base import *
    from modules.arithmetic import *
    from modules.RSA_attacks import *
    from modules.RSA import *
    import modules.test_attacks as test_attacks

from datetime import datetime as dt


##-Test function
def tester(func_name, assertion):
    '''Print what is tested and fail if the assertion failed.'''

    if assertion:
        print(f'Testing {func_name}: passed')
        return True

    else:
        print(f'Testing {func_name}: failed')
        raise AssertionError


```

```python
##-Base
def test_base():
    tester(
        'base: split',
        split('azertyuiopqsdfghjklmwxcvbn', 3) == ['aze', 'rty', 'uio', 'pqs', 'dfg', 'hjk', 'lmw', 'xcv', 'bn']
    )
    tester(
        'base: split',
        split('azertyuiopqsdfghjklmwxcvbn', 3, '0') == ['aze', 'rty', 'uio', 'pqs', 'dfg', 'hjk', 'lmw', 'xcv', 'bn0']
    )


##-Arithmetic
def test_arith(size=2048):
    tester(
        'arithmetic: mult_inverse',
        [mult_inverse(k, 7) for k in range(1, 7)] == [1, 4, 5, 2, 3, 6]
    )
    tester(
        'arithmetic: max_parity',
        max_parity(256) == (8, 1) and max_parity(123) == (0, 123) and max_parity(8 * 5) == (3, 5)
    )
    tester(
        'arithmetic: isSurelyPrime',
        (not isSurelyPrime(1)) and isSurelyPrime(2) and isSurelyPrime(11) and isSurelyPrime(97) and (not isSurelyPrime(561))
    )
    tester(
        'arithmetic: iroot',
        iroot(2, 2) == 1 and iroot(27, 3) == 3
    )
    print('Testing fermat_factor :')
    tester(
        'arithmetic: fermat_factor',
        test_attacks.test_fermat_factor(size, size // 4)
    )


##-RSA
def test_OAEP(size=2048):
    '''
    Test the OAEP padding scheme.

    - size : the RSA key's size. The OAEP size is `size // 8 - 1`.
    '''

    # Using the LICENCE file as test file
    try:
        with open('LICENCE') as f:
            m = f.read()

    except FileNotFoundError:
        with open('../LICENCE') as f:
            m = f.read()

```

```python
    C = OAEP(size // 8 - 1)
    e = C.encode(m)

    tester('RSA: OAEP', m.encode() == C.decode(e))


def test_RSA(k=None, pad='raw', size=2048):
    '''Test RSA encryption / decryption'''

    print(f'Testing RSA (padding : {pad}).')

    if k is None:
        print('Generating a key ...', end=' ')
        k = RsaKey()
        k.new(size=size)
        print('Done.')

    else:
        size = k.size

    C = RSA(k, pad)

    if pad.lower() == 'int':
        m = randint(0, k.n - 1)

    else:
        print('Reading file ...', end=' ')
        # Using the LICENCE file as test file
        try:
            with open('LICENCE') as f:
                m = f.read()

        except FileNotFoundError:
            with open('../LICENCE') as f:
                m = f.read()

        print('Done.')

    print('Encrypting ...', end=' ')
    enc = C.encrypt(m)
    print('Done.\nDecrypting ...', end=' ')
    dec = C.decrypt(enc)

    if pad.lower() == 'oaep':
        # print(dec)
        dec = dec.decode()

    print('Done.')

    tester(f'RSA: RSA (padding : {pad})', dec == m)


##-Run tests function
def run_tests(size=2048):
    '''Run all the tests'''

    t0 = dt.now()
    test_base()
```

```
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_arith(size=size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')

    test_OAEP(size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')

    test_RSA(pad='int', size=size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_RSA(pad='raw', size=size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')
    test_RSA(pad='oaep', size=size)
    print(f'\n--- {dt.now() - t0}s elapsed.\n')

    print('All tests passed.') #Otherwise the function `tester` in the tests would have raised an AssertionError.


##-Main
if __name__ == '__main__':
    from sys import argv
    from sys import exit as sysexit

    if len(argv) == 1:
        size = 2048

    else:
        try:
            size = int(argv[1])

        except:
            print(f'Wrong argument at position 1 : should be the RSA key size (in bits).\nExample : "{argv[0]} 2048".')
            sysexit()

    run_tests(size=size)
```