WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

RSA Encryption implementation from scratch in python

ECRYPT PROJECT

AIMIN Charles
GADRAT Alessandro

2022

Contents

1 The RSA Crytposystem

1.1 Introduction

RSA encryption (named after the initials of its three inventors) is an asymmetric cryptographic algorithm, widely used in electronic commerce, and more generally for exchanging confidential data on the Internet. The algorithm was described in 1977 by Ronald Rivest, Adi Shamir and Leonard Adleman. RSA was patented1 by the Massachusetts Institute of Technology (MIT) in 1983 in the USA.

The RSA encryption system is an easy-to-use asymmetric encryption method that is very popular in many areas that require data transfer over the Internet. It consists of two RSA encryption keys, one public and one private. While the public key is used for encryption, the private key is used to decrypt the data. Since no algorithm is able to decode the private key from the public key, this method is seen as a secure process. In addition to encryption, the RSA encryption system also allows the generation of its own digital signatures.

1.2 How to encrypt in RSA

RSA encryption is asymmetric: it uses a key pair (integers) consisting of a public key to encrypt and a private key to decrypt confidential data. Both keys are created by a person, often conventionally named Alice, who wants confidential data sent to her. Alice makes the public key available. This key is used by her correspondents (Bob, etc.) to encrypt the data sent to her. The private key is reserved for Alice, and allows her to decrypt the data. The private key can also be used by Alice to sign a piece of data she sends, the public key allowing any of her correspondents to verify the signature.

A necessary condition is that it is "computationally impossible" to decrypt using the public key alone, in particular to reconstruct the private key from the public key, i.e. that the computational means available and the methods known at the time of the exchange (and the time the secret must be kept) do not allow it.

RSA encryption is often used to communicate a symmetric encryption key, which then allows the exchange to continue confidentially: Bob sends Alice a symmetric encryption key which can then be used by Alice and Bob to exchange data.

1.2.1 Creating the keys

The key creation stage is the responsibility of Alice. It does not occur at each encryption because the keys can be reused. The main difficulty, which is not solved by encryption, is that Bob must be sure that the public key he holds is Alice's. The renewal of keys only occurs if the private key is compromised, or as a precaution after a certain period of time (which may be years).

- 1 : We choose two very large distinct primes p and q.
- 2 : Calculate their product $n = p * q$, called the cipher modulus.
- 3 : Calculate $\phi(n) = (p-1)(q-1)$ (this is the value of the Euler indicator in n)
- 4 : Choose a natural number e that is prime with $\phi(n)$ and strictly less than $\phi(n)$, Called the encryption exponent, such that $GCD(e, \phi(n)) = 1$

- 5 : Calculate the natural number d, the modular inverse of e for multiplication modulo $\phi(n)$ and strictly less than $\phi(n)$, called the decryption exponent; d can be efficiently calculated by the extended Euclid algorithm such that $d * e \equiv 1 \pmod{\phi(n)}$

The pair (n, e) or (e, n) is the public key of the cipher, while its private key is the number d, since the decryption operation requires only the private key d and the integer n, known by the public key (the private key is sometimes also defined as the pair (d, n) or the triplet (p, q, d))

1.2.2 Message encryption

If M is a natural number strictly less than n representing a message, then the encrypted message will be represented by

$$C \equiv M^e \pmod{n} \tag{1}$$

the natural number C being chosen strictly less than n.

1.2.3 Message decryption

To decrypt C, we use d, the inverse of e modulo $(p-1)(q-1)$, and we find the clear message M by

$$M \equiv C^d \pmod{n} \tag{2}$$

1.2.4 Example

An example with small prime numbers (in practice you need very large prime numbers):

- we choose two primes $p = 3$, $q = 11$
- their product $n = 3 * 11 = 33$ is the cipher modulus
- $\phi(n) = (3-1)(11-1) = 210 = 20$
- we choose $e = 3$ (first with 20) as the encryption exponent
- the decryption exponent is $d = 7$, the inverse of 3 modulo 20 (indeed $e*d = 3*7 \equiv 1 \pmod{20}$).

Alice's public key is $(n, e) = (33, 3)$, and her private key is $(n, d) = (33, 7)$. Bob transmits a message to Alice.

- Bob encrypts $M = 4$ with Alice's public key: $4^3 \equiv 31 \pmod{33}$, the cipher is $C = 31$ which Bob passes on to Alice.
- Decryption of $C = 31$ by Alice with her private key: $31^7 \equiv 4 \pmod{33}$, Alice finds the original message $M = 4$.

The mechanism of signing by Alice, using her private key, is analogous, by exchanging the keys.

2 Functional description of the application

2.1 Generate a prime number

This first functionality aims to generate prime numbers of the length desired by the user via the Rabin-Miller method.

The input of the length of this number (in bits) is done in the command terminal.

```
You want your number to be what length (bits) : 512
Prime number found in 0.5672278404235 84 seconds
24592317814922051333402362882461782008466589425020908400273881157094796181324140938165513207101142279117663650537041009102982136357929464241150325936467631
```

2.2 Encrypt plain text

This feature allows you to encrypt a message using RSA encryption.

By executing this function you will have the ability to choose the length of the primes numbers p and q in bits (I recommend that you use 512-bit prime numbers), as well as access the Euler totient of n, the exponent e, the private key d.

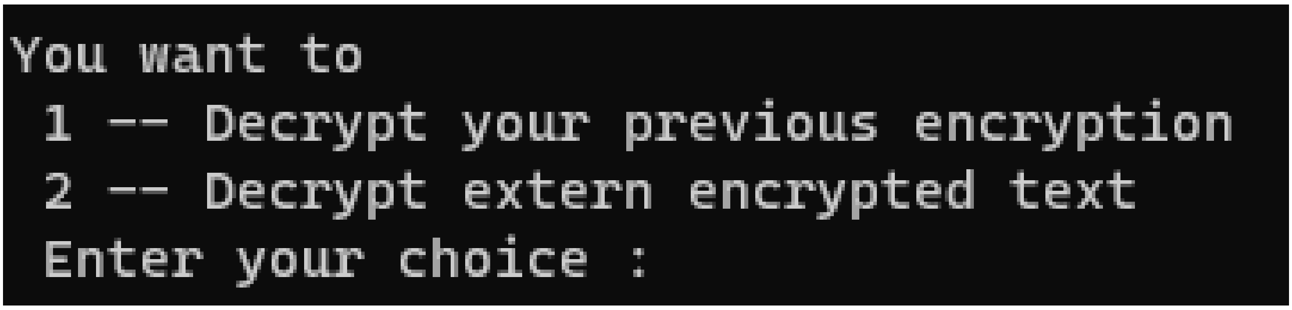
```
Enter your choice: 2
You want your factors n q to be what length (bits) : 512
Prime number found in 0.3194098472595215 seconds
Prime number found in 0.9263780117034912 seconds


We have n = 309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558354114671872409831731682201638849145662861185017476396619356672786165549725487427

We have phi(309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558354114671872409831731682201638849145662861185017476396619356672786165549725487427) = 309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558354114671872409831731682201638849145662861185017476396619356672786165549725487427

We have e = 112815563475445820758575606745504012989194460209451354867273684671496488894564547975368843527961009740607284375395020734606272784780347971253950413106990817241569382634152877552570769903927525794113843217127806723692982052967395170344063967949250637527170043011057683435305568067664182912441620446217240769133

We have d = 272762379630413960323911284421652836128046942743318346475839280552101328594990419509270880180770200635733173070259923939886030213081147669519851015133572159085594956421189529592279879891792191601908087381337310017153950131574278729824639985448860915178143423237829727807680792553635609961671421872357257067301

Public key : 309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558354114671872409831731682201638849145662861185017476396619356672786165549725487427 112815563475445820758575606745504012989194460209451354867273684671496488894564547975368843527961009740607284375395020734606272784780347971253950413106990817241569382634152877552570769903927525794113843217127806723692982052967395170344063967949250637527170043011057683435305568067664182912441620446217240769133

Private key : 272762379630413960323911284421652836128046942743318346475839280552101328594990419509270880180770200635733173070259923939886030213081147669519851015133572159085594956421189529592279879891792191601908087381337310017153950131574278729824639985448860915178143423237829727807680792553635609961671421872357257067301
```

You can then enter a plain text which will then be encrypted and printed on the terminal as a list of hexadecimal numbers

```
Message to encrypt : Hello World !



The plain text is : Hello World !


The encrypted message is :  ['0x14b6342354d67df33ab2950efae489545455a0e04bcff45ef6a067c517947b903438a462217170af7124b8ab
1a1153eca652be5b09d1587e8a368ce25b8d9904d6cd55c3d867b63af7752b5ed39abc58bb59e62bfcfe5eb511cfcb4758f7d7beef22b7271af882c5
796ed96cb1b455873fa6a7ccc97dcbef14b9c29a2dd50dc2d', '0x1a370c15f39085302c695af6e10600b7a6aa03e4d485cc148fae901a22c7c33a1
cc650778980488cda78adab00aed927344679c9c3b51bf7223c83565f595c2bdceca9b97318a3d9520a253e3746d357dd222bf5cd192e9706de77468
47326075f624c76f72e7ac07d89380013a93d0409a1cf8dcb9fefa8607948a094956c480', '0x35774bbffb98c200e845d1ea7fb32753445cf48285
a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb92d1d31c2b83c926f87f88c607eca88ca414
69e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a1d555457a', '0x35774bbffb98c200e845
d1ea7fb32753445cf48285a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb92d1d31c2b83c9
26f87f88c607eca88ca41469e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a1d555457a', '
0xf88a8345405fa1c4b45dd7abe7f1646495a78021536380291aea961f92e01999a4a89be74e78d732f6b41d6b84abd8e9077ce07de177210e791649
b6e7c9dc7fd4ecc155c2c1fd71d7c5a9ebd319d587d44470ec59b17b202669251d28aa0c7c7d864518f93ccdd18e213c30e2b7575764688584faf02e
d9bd39112afca464b6', '0x198295b0f692271a3bf384297332da0cb35783de6d83f054bc75871c011098a909ded08b65db259cc60d42cdb10b2ac3
6bdf2b46530b010235ff28b026cb32d494e026f52ce75879c36c35f80a6730641e71116fcfe8afdcb3a5fa3541130c51f243400f845f679f6645e2d7
649968e3fa5656b653a6cf3469c720a619b7cc67c', '0x6464ae34537412da03582e2f051d061d94854de98a5fcab25f3a188d34f8f2dfbfbd059ae
6d2e2138332ccd02e0e67d5841d22266689e45fc4fa51d3ab52fff391f0ba9663aadffbc8c0b11437c1f70a2cf82ca0e5f449b3dab5136450a49b874
ca4e41e081d17987258905171dd2e8e85428269f9e29222b77be29ed859a6d9', '0xf88a8345405fa1c4b45dd7abe7f1646495a78021536380291ae
a961f92e01999a4a89be74e78d732f6b41d6b84abd8e9077ce07de177210e791649b6e7c9dc7fd4ecc155c2c1fd71d7c5a9ebd319d587d44470ec59b
17b202669251d28aa0c7c7d864518f93ccdd18e213c30e2b7575764688584faf02ed9bd39112afca464b6', '0xa3029f113905c4799f3cdd0298292
0a72430a4d05aec0904717b387940ab3d233d0e7e429395480f87de7b6731effcaa4ab7d1f71faef402a5dd64d5afe955766e5e606a0f87921dc2610
1804f36cab5a9560cd8b838a35d87dbaca65a2ae37bf76902a9d91d77dc87eabb488bf7bf2fc4793ab30f1e9fdaea453c2281efbe4b', '0x35774bb
ffb98c200e845d1ea7fb32753445cf48285a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb9
2d1d31c2b83c926f87f88c607eca88ca41469e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a
1d555457a', '0x171115306128607f52fbda1d5261d05637fa8c124159dc105563a67e54864daea7c04077671d1205ad8abcec815d2e66099e27ca4
b995ecc762d34411d3ab79688217df6d559d97424b193c5678bf1b76a502bd416b003da36035fba3aa25bc3cd144a7f01ac45a4404cf4ab865a2f592
fc066949dbcea7ef683ab3bcd09b100f', '0x198295b0f692271a3bf384297332da0cb35783de6d83f054bc75871c011098a909ded08b65db259cc6
0d42cdb10b2ac36bdf2b46530b010235ff28b026cb32d494e026f52ce75879c36c35f80a6730641e71116fcfe8afdcb3a5fa3541130c51f243400f84
5f679f6645e2d7649968e3fa5656b653a6cf3469c720a619b7cc67c', '0x77a5fa22b3f0b1daf007aa3814fd527a688b2d96f7fb4f7bad28ed3bbb1
3c2be34dc6e161e0d5118b6a6dc80b1536dccd2d26db7c3c1d0e3710d9ceb197c707842428e7a3420f931013e0ca5e3a5c2ba8884be0c61d7d7a1f33
752a3dd482d5fe5fc592e791abbba76a96d7f678ffa2885c70763746bb2a604d10be7af81b8ed']
```

2.3 Decrypt cipher text

This feature contains several functions.



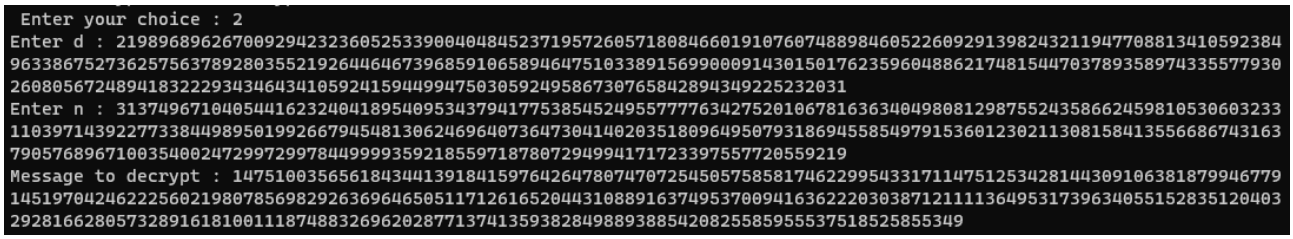
You can :

-Decrypt a text cypher previously encrypted by the program.

Dependent on the "encrypt" functionality

-Decrypt a cypher text with custom parameters.

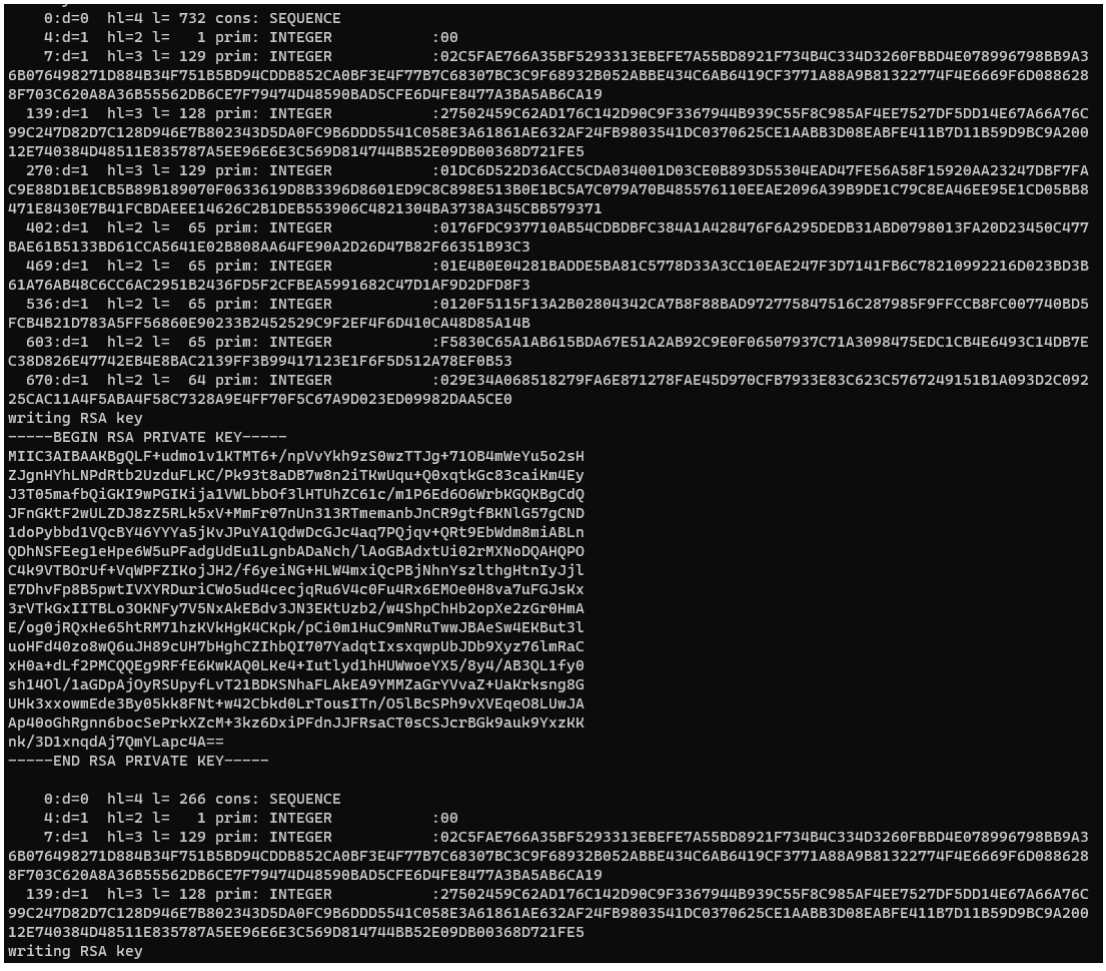
Warning ! To use this feature you must first know the integer d and n that were used to encrypt the message



2.4 OpenSSl keys generator

Allows you to generate public and private keys with OpenSSL using the settings formerly defined in "Encrypt".
Returns a pair of keys: pubkey.pem (public key) and prikey.pem (private key)

Dependencies : Execution of the Encrypt function, requires OpenSSL and Linux.



The generation of keys via OpenSSL confirms that the preliminary steps of RSA are valid because in the code we use the parameters defined in the Encrypt plain text function to generate the OpenSSL keys.

2.5 Encrypt with OpenSSL as a check

This feature allows you to encrypt the plaintext defined in "Encrypt" with RSAUTL in order to compare the cypher of this program with that of OpenSSL.

Dependencies: Execution of the Encrypt and Keygen functions, requires OpenSSL and Linux.

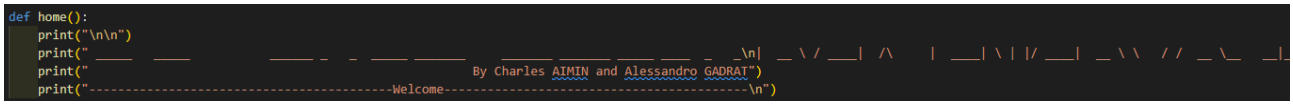
We can see the cipher text encrypted by OpenSSL and the plain text decrypted by OpenSSL.

3 Description of designed code structure

3.1 Main

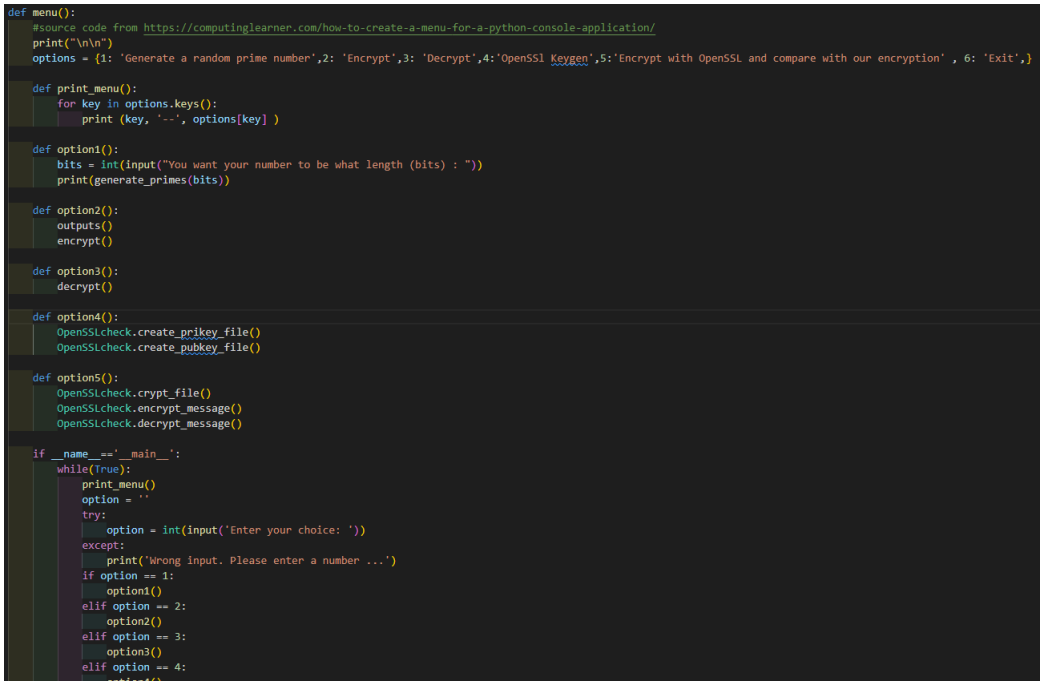
3.1.1 Function home()

This function displays an ASCII art with the program name and our names.



3.1.2 Function menu()

This function displays a menu that allows the user to choose the functionality they want to use.



3.2 RabinMiller

3.2.1 Function prime_test(n, a)

This function performs the Miller Rabin test on a randomly generated integer.

```
def prime_test(n, a):
    """
    It returns True if n is prime, and False if n is composite
    :param n: the number to be tested
    :param a: the number to be tested
    :return: True or False
    """

    x = n - 1
    while not x & 1:
        x >>=1
    if pow(a, x, n) == 1:
        return True
    while x < n - 1:
        if pow(a, x, n) == n - 1:
            return True
        x <<= 1

    return False
```

3.2.2 Function rabin_miller(n, k=80)

This function defines how many times we perform the Miller Rabin test here, we do the test 80 times.

```
def rabin_miller(n, k = 80):
    """
    If the number passes the Miller-Rabin test k times, then it's probably prime
    :param n: the number to be tested
    :param k: the number of times to run the test, defaults to 80 (optional)
    :return: True or False
    """
    for i in range(k):
        a = rand.randrange(2, n - 1)
        if not prime_test(n, a):
            return False
    return True
```

3.3 GeneratePrimes

3.3.1 Function Generate_primes(bits)

It generates a random number of the specified number of bits, and then tests it to see if it's prime. If it's not, it generates another random number and tests it. It keeps doing this until it finds a prime number

```
def generate_primes(bits):
    """
    It generates a random number of the specified number of bits, and then tests it to see if it's
    prime. If it's not, it generates another random number and tests it. It keeps doing this until it
    finds a prime number
    :param bits: The number of bits in the prime number
    :return: A prime number
    """


    timestart = time.time()
    while True:
        a = (rand.randrange(1 << bits - 1, 1 << bits) << 1) + 1
        if rabin_miller(a):
            print(f"Prime number found in {time.time() - timestart} seconds")
            return a
```

3.4 RSAEncrypt

3.4.1 Function factors()

It asks the user for a number of bits, then generates two random primes of that length, and returns the product of those primes and the totient of that product

```
def factors():
    """
    It asks the user for a number of bits, then generates two random primes of that length, and returns
    the product of those primes and the totient of that product
    :return: n, phi
    """
    global p, q
    b = int(input("You want your factors n q to be what length (bits) : "))
    p = generate_primes(b)
    q = generate_primes(b)
    n = p*q
    print("\n")
    print("We have n = %d \n" %n)
    phi = int((p - 1)*(q - 1))
    print("We have phi(%d) = %d \n" %(n, phi))
    return n, phi
```

3.4.2 Function gcd(n, phi)

It generates a random number between 1 and n, and then checks if it is coprime with phi. If it is, it returns the number. If it isn't, it generates another random number and checks again

```
def gcd_e(n, phi): #totient of n
    """
    It generates a random number between 1 and n, and then checks if it is coprime with phi. If it is,
    it returns the number. If it isn't, it generates another random number and checks again
    :param n: the modulus
    :param phi: the totient of n
    :param e: the public key
    :return: The value of e.
    """
    while True:
        e = rand.randrange(1, n)
        gcd = int(math.gcd(e, phi))
        if gcd == 1:
            break
    print("We have e = %d \n" %e)
    return e
```

3.4.3 Function find_inv(n, phi)

It finds the inverse of e mod phi.

```
def find_inv(e, phi): #private key d
    """
    It finds the inverse of e mod phi.

    :param e: public key
    :param phi: the totient of n
    :return: The private key d
    """
    #source : https://inventwithpython.com/cryptomath.py
    if math.gcd(e, phi) != 1:
            return None # no mod inverse if a & m aren't relatively prime

        # Calculate using the Extended Euclidean Algorithm:
    d, u2, u3 = 1, 0, e
    v1, v2, v3 = 0, 1, phi
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, d, u2, u3 = (d - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return d % phi
```

The source code is taken from an internet source : https://inventwithpython.com/cryptomath.py

3.4.4 Function input_text()

It takes a string, converts it to a list, and returns the list.

```
def input_text():
    """
    It takes a string, converts it to a list, and returns the list.
    :return: A list of the characters in the input string.
    """
    global plain_list, plain
    plain = input("Message to encrypt : ")
    print("\n")
    plain_list  = []
    for k in range (len(plain)):
        plain_list.append(plain[k])
    print("The plain text is :", "".join(plain_list))
    print("\n")
    file = open("message.txt", "w")
    file.write(plain)
    return plain_list, plain
```

3.4.5 Function encrypt()

It takes the input text, converts it to ASCII, encrypts it, and then converts it to hexadecimal

```
def encrypt():
    """
    It takes the input text, converts it to ASCII, encrypts it, and then converts it to hexadecimal
    :return: the encrypted message, the public key, the modulus, the private key, and the encrypted
    message in hexadecimal.
    """

    global encrypt_list, e, n, d, encrypt_list_hex
    plain_list, plain = input_text()
    encrypt_list = []
    encrypt_list_hex = []
    for i in range (len(plain_list)):
        plain_asc = ord(plain_list[i])
        encrypt_asc = pow(plain_asc,e,n)
        encrypt_list.append(encrypt_asc)
        encrypt_list_hex.append(hex(encrypt_list[i]))
    print("The encrypted message is : ", encrypt_list_hex)
    print("\n")

    return encrypt_list, e, n, d, encrypt_list_hex
```

3.5 RSADecrypt

3.5.1 RSADecrypt()

It takes the user's choice of whether to decrypt a previously encrypted message or to decrypt an external message, and then it decrypts the message using the user's input of d and n

```
def decrypt():
    """
    It takes the user's choice of whether to decrypt a previously encrypted message or to decrypt an
    external message, and then it decrypts the message using the user's input of d and n
    """
    choice = int(input("\n\nYou want to\n 1 -- Decrypt your previous encryption\n 2 -- Decrypt extern encrypted text\n Enter your choice : "))
    if choice == 1:
        encrypt_list, n, d, p, q, e, encrypt_list_hex = recup_ssl()

        decrypt_list = []
        for i in range (len(encrypt_list)):
            decrypt_asc = pow(encrypt_list[i],d,n)
            decrypt_list.append(chr(decrypt_asc))
        print("The encrypted text : ", encrypt_list)
        print("The plain text is : %s" %("".join(decrypt_list)))
    elif choice == 2:
        d = int(input("Enter d : "))
        n = int(input("Enter n : "))
        crypt = input("Message to decrypt : ")
        crypt_list  = crypt.split(" ")
        for k in range (len(crypt_list)):
            crypt_list[k] = int(crypt_list[k])
        decrypt_list = []
        for i in range (len(crypt_list)):
            decrypt_asc = pow(encrypt_list[i],d,n)
            decrypt_list.append(chr(decrypt_asc))
        print("The plain message is : %s" %("".join(decrypt_list)))
    else:
        print("Please select a existing menu")
```

3.6 OpenSSLcheck

3.6.1 Function gen_int()

This function takes the integers generated by the encrypt function and calculates e1 and e2 to generate OpenSSL keys with our parameters.

```
def gen_int():
    """
    :return: n, d, p, q, e, e1, e2
    """
    encrypt_list, n, d, p, q, e,encrypt_list_hex = recup_ssl()
    global e1, e2
    e1 = d %(p-1)
    e2 = d %(q-1)
    return encrypt_list, n, d, p, q, e, e1, e2, encrypt_list_hex
```

3.6.2 Function findModInverse()

It finds the inverse of e mod phi.

```
def find_inv(e, phi): #private key d
    """
    It finds the inverse of e mod phi.

    :param e: public key
    :param phi: the totient of n
    :return: The private key d
    """
    #source : https://inventwithpython.com/cryptomath.py
    if math.gcd(e, phi) != 1:
            return None # no mod inverse if a & m aren't relatively prime

        # Calculate using the Extended Euclidean Algorithm:
    d, u2, u3 = 1, 0, e
    v1, v2, v3 = 0, 1, phi
    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, d, u2, u3 = (d - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return d % phi
```

3.6.3 Function create_prikey_file()

It generates a private key file using the OpenSSL command line tool with custom parameters from our program. This function return a prikey.pem file

```
def create_prikey_file():
    """
    It generates a private key file using the openssl command line tool with custom parameters from our program.
    """
    encrypt_list, n, d, p, q, e, e1, e2, encrypt_list_hex = gen_int()
    u1 = findModInverse()
    file = open("prikey_conf.txt", "w")
    file.write("asn1=SEQUENCE:rsa_key\n\n")
    file.write("[rsa_key]\n")
    file.write("version=INTEGER:0\n")
    file.write("modulus=INTEGER:%d\n" %n)
    file.write("pubExp=INTEGER:%d\n" %e)
    file.write("privExp=INTEGER:%d\n" %d)
    file.write("p=INTEGER:%d\n" %p)
    file.write("q=INTEGER:%d\n" %q)
    file.write("e1=INTEGER:%d\n" %e1)
    file.write("e2=INTEGER:%d\n" %e2)
    file.write("coeff=INTEGER:%d\n" %u1)
    file.close()
    os.system("openssl asn1parse -genconf prikey_conf.txt -out prikey.der")
    os.system('openssl rsa -in prikey.der -inform der -text -check > store_pri_key.txt')
    file=open("store_pri_key.txt", "r")
    keylist=[]
    for line in file:
        l=file.readline()
        if l==("RSA key ok\n"):
            while l != "-----END RSA PRIVATE KEY-----\n":
                l=file.readline()
                keylist.append(l)
    privatekey="".join(keylist)
    file.close()
    print(privatekey)
    file=open("prikey.txt", "w")
    file.write(privatekey)
    file.close()
    os.system('mv prikey.txt prikey.pem')
```

3.6.4 Function create_pubkey_file()

It creates a file called pubkey_conf.txt, which contains the public key information in ASN.1 format. Then, it uses the openssl command line tool to convert the ASN.1 format to a DER format, and then to a PEM format. The PEM format is the one that we will use to encrypt the message.

```
def create_pubkey_file():
    """
    It creates a file called pubkey_conf.txt, which contains the public key information in ASN.1 format.


    Then, it uses the openssl command line tool to convert the ASN.1 format to a DER format, and then to
    a PEM format.

    The PEM format is the one that we will use to encrypt the message.
    """

    encrypt_list, n, d, p, q, e, e1, e2, encrypt_list_hex = gen_int()
    file = open("pubkey_conf.txt", "w")
    file.write("asn1=SEQUENCE:rsa_key\n\n")
    file.write("[rsa_key]\n")
    file.write("version=INTEGER:0\n")
    file.write("modulus=INTEGER:%d\n" %n)
    file.write("pubExp=INTEGER:%d\n" %e)
    file.close()
    os.system("openssl asn1parse -genconf pubkey_conf.txt -out pubkey.der")
    os.system('openssl rsa -in prikey.pem -outform PEM -pubout -out pubkey.pem')
```

3.6.5 Function encrypt_message()

It encrypts the message.txt file using the public key and stores it in top_secret.enc with OpenSSL.

```
def encrypt_message():
    """
    It encrypts the message.txt file using the public key and stores it in top_secret.enc.
    """
    os.system("openssl rsautl -encrypt -inkey pubkey.pem -pubin -in message.txt -out top_secret.enc")
    try:
        with open('top_secret.enc'): pass
    except IOError:
        print("Error : The file cannot be created")
        exit(1)
    print("The encrypted text is : \n")
    os.system("cat top_secret.enc")
    print("\n")
```

3.6.6 Function decrypt_message()

This function decrypts the message using the private key and outputs the decrypted message

```
def decrypt_message():
    """
    This function decrypts the message using the private key and outputs the decrypted message
    """
    os.system("openssl rsautl  -decrypt -in top_secret.enc -out plain_out.txt -inkey prikey.pem")
    print("The encrypted text is : ")
    os.system("cat plain_out.txt")
    print("\n")
```

3.7 OAEP_test

In this file we have tried to do my own OAEP padding to follow the RSA RFC8017 encryption conventions (https://datatracker.ietf.org/doc/html/rfc8017).

We did not implement this part because I could not encrypt my plain text without padding in OpenSSl.

3.7.1 Function get_byte_lenght()

It returns the number of bytes needed to represent the given message.

```
def get_byte_length(message):
    res = 0
    if (len(bin(message)) - 2) % 8 != 0:
        res += 1
    res += (len(bin(message)) - 2) // 8
    return res
```

3.7.2 Function padding()

It takes a message and a target length, and returns a number that is the message padded with random bytes, a 0x02 byte, and a 0x00 byte.

```
def padding(message, target_length):
    # 02
    res = 0x02 << 8 * (target_length - 2)
    # random
    random_pad = os.urandom(target_length - 3 - get_byte_length(message))
    for idx, val in enumerate(random_pad):
        res += val << (len(random_pad) - idx + get_byte_length(message)) * 8
    # 00
    # message
    res += message
    print('random pad : ',random_pad)
    return res
```

3.7.3 Function main()

The main function performs a padding on a plain text according to the integer n and prints it in the terminal.

```
def main():
    n =  437272606206591106861256033136078687619825406017895551939289308715919384679348570
    N_size = get_byte_length(n)
    print(N_size)
    hexplain = []
    list_hex = []
    padded = []
    plain = "test hexa python"
    for i in range (len(plain)):
        plain_asc = ord(plain[i])
        hexplain.append(plain_asc)
        list_hex.append(hex(hexplain[i]))
    print(list_hex)
    for i in range (len(list_hex)):
        padded.append(padding(hexplain[i], N_size))
    print(padded)
```

4 Tests

We could not perform the test phase with OpenSSL because :

OpenSSL performs a padding, in order to bypass the automatic padding of OpenSSL, it is necessary to add the "-raw" parameter to the encrypt command of RSAUTL. The message and the integer n must then be exactly the same length, however RSAUTL always returned a size error whereas we made sure to have n and M of the same size. We were therefore unable to check the encryption and decryption of our program.

On the other hand, RSAUTL accepted our parameters to generate encryption keys, which means that the calculations of our implementation are correct, the only step that is missing being the padding.

4.1 Our tests

Encryption parameters are generated with the Ecrypt plain text function.

```
We have n = 426716490659420984965838450131286804719984037042457313534098188322448117882905129057127359365310477191470298
926896916810113018762262771104950802561784447065500950836939501266763409724565396221930555401474420704602378944278698740
690822282233008910631261152597643367479356106476579312138019752228254241784921813

We have phi(426716490659420984965838450131286804719984037042457313534098188322448117882905129057127359365310477191470298
926896916810113018762262771104950802561784447065500950836939501266763409724565396221930555401474420704602378944278698740
690822282233008910631261152597643367479356106476579312138019752228254241784921813) = 426716490659420984965838450131286804
471998403704245731353409818832244811788290512905712735936531047719147029892689691681011301876226277110495080256178444702
405700004286081447383185505246375223317791298516858584368154450196225371819560986408893760932682328046374045461122411314
043507441833342096169217557170816O

We have e = 335751174767406032515911945451600644338809095228508520269064632861820517840052780042109591748053318178384564
644939679291530670893250404542055496334841813239667686795407862595390257490354181589140156790163399200372699874404236105
763141148003014066144823574146478099601294767661494448233177333219499379185121629

We have d = 205501972305576293802518703017590764356274730385356494882275549150626264830384248287017541961799700484113176
255066992951006815419960524127171238188481680944779765989232893891635542274046074404325845457564866011232532167975693386
732561230194326712516055711781416193423187380339444166368713587873567140112446709

Public key : 426716490659420984965838450131286804719984037042457313534098188322448117882905129057127359365310477191470298
926896916810113018762262771104950802561784447065500950836939501266763409724565396221930555401474420704602378944278698740
690822282233008910631261152597643367479356106476579312138019752228254241784921813 335751174767406032515911945451600644
388090952285085202690646328618205178400527800421095917480533181783845646449396792915306708932504045420554963348418132396
676867954078625953902574903541815891401567901633992003726998744042361057631411480030140661448235741464780996012947676614
9444823317733321949937918512162 9

Private key : 205501972305576293802518703017590764356274730385356494882275549150626264830384248287017541961799700484113176
255066992951006815419960524127171238188481680944779765989232893891635542274046074404325845457564866011232532167975693386
732561230194326712516055711781416193423187380339444166368713587873567140112446709
```

We then generate keys with OpenSSL.

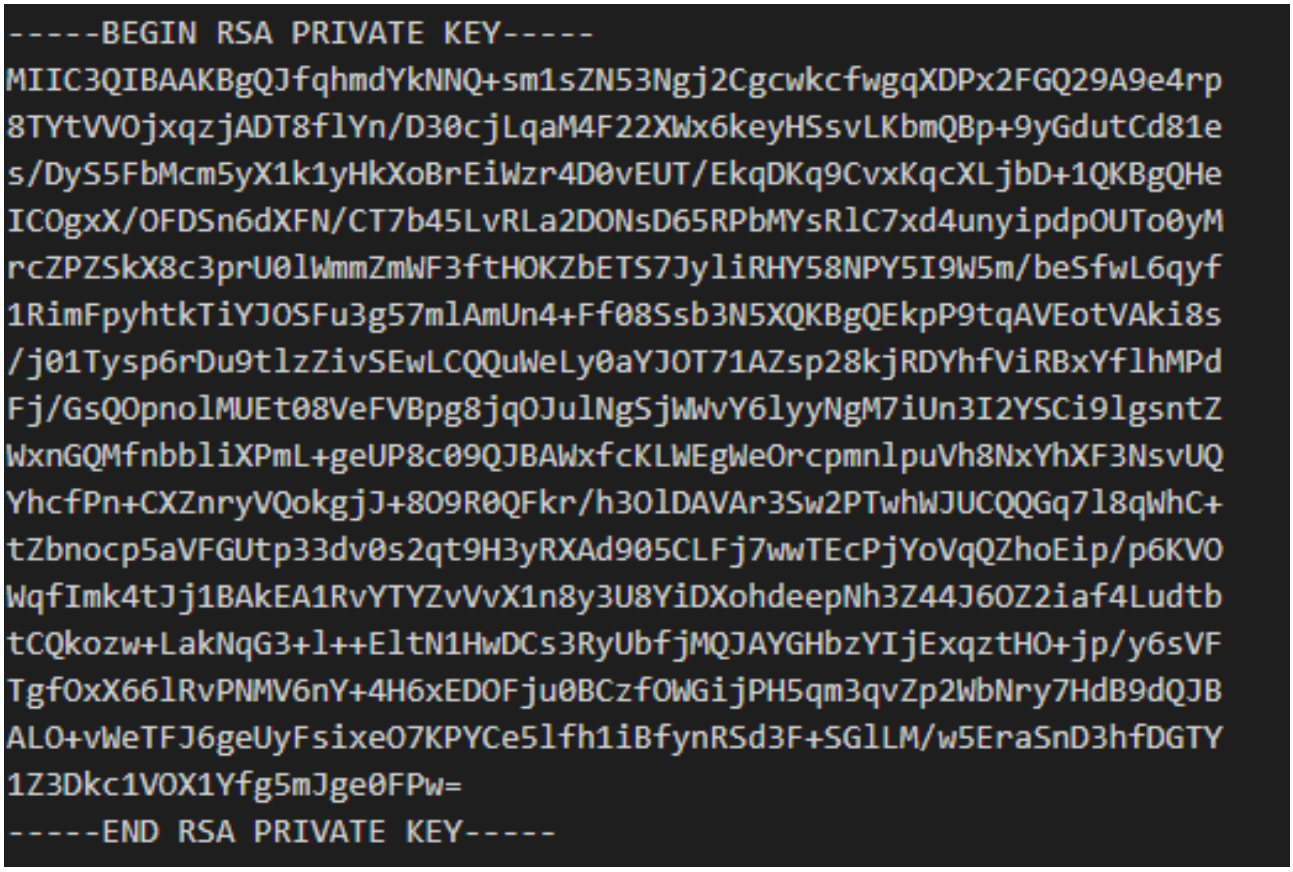


Figure 1: Private key stored in prikey.pem

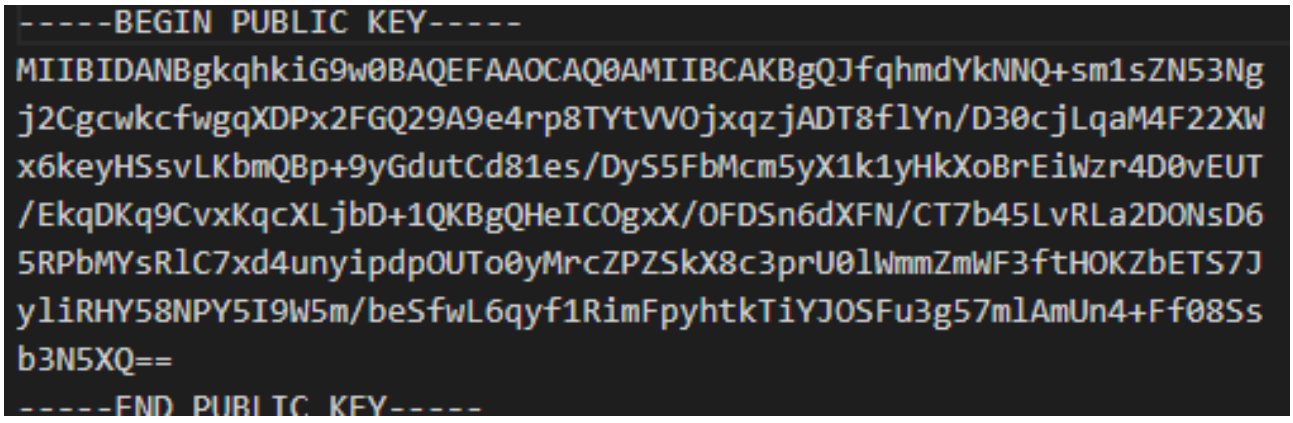


Figure 2: Public key stored in pubkey.pem

References

[1] ttps://security.stackexchange.com/questions/176394/how-does-openssl-generate-a-big-prime-number-so-fast

[2] ttps://gist.github.com/Ayrx/5884790

[3] ttps://www.alpertron.com.ar/ECM.HTM

[4] ttps://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html

[5] ttps://mathworld.wolfram.com/FermatsLittleTheorem.html

[6] ttps://artofproblemsolving.com/wiki/index.php/Euclid

[7] ttps://en.wikipedia.org/wiki/RSA_(cryptosystem)

[8] ttps://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

[9] ttp://www.man-linux-magique.net/man1/rsautl.html

[10] ttps://stackoverflow.com/questions/19850283/how-to-generate-rsa-keys-using-specific-input-numbers-in-openssl

[11] ttps://datatracker.ietf.org/doc/html/rfc8017

[12] ttps://datatracker.ietf.org/doc/html/rfc2313

[13] ttps://www.rfc-editor.org/rfc/rfc3447

[14] ttps://inventwithpython.com/cryptomath.py

[15] ttps://www.tutorialspoint.com/what-are-risks-in-implementing-the-rsa-algorithm-without-padding

[16] ttps://github.com/mimoo/RSA_PKCS1v1_5_attacks/blob/master/bb98_graphic.sage

[17] ttps://datatracker.ietf.org/doc/html/rfc5208