

RSA Encryption implementation from scratch in python



ECRYPT PROJECT

AIMIN Charles
GADRAT Alessandro

2022

Contents

1	The RSA Cryposystem	2
1.1	Introduction	2
1.2	How to encrypt in RSA	2
1.2.1	Creating the keys	2
1.2.2	Message encryption	3
1.2.3	Message decryption	3
1.2.4	Example	3
2	Functional description of the application	4
2.1	Generate a prime number	4
2.2	Encrypt plain text	4
2.3	Decrypt cipher text	5
2.4	OpenSSL keys generator	7
2.5	Encrypt with OpenSSL as a check	8
3	Description of designed code structure	8
3.1	Main	8
3.1.1	Function home()	8
3.1.2	Function menu()	8
3.2	RabinMiller	9
3.2.1	Function prime_test(n, a)	9
3.2.2	Function rabin_miller(n, k=80)	9
3.3	GeneratePrimes	10
3.3.1	Function Generate_primes(bits)	10
3.4	RSAS Encrypt	10
3.4.1	Function factors()	10
3.4.2	Function gcd(n, phi)	11
3.4.3	Function find_inv(n, phi)	11
3.4.4	Function input_text()	12
3.4.5	Function encrypt()	12
3.5	RSADecrypt	13
3.5.1	RSADecrypt()	13
3.6	OpenSSLcheck	13
3.6.1	Function gen_int()	13
3.6.2	Function findModInverse()	14
3.6.3	Function create_prikey_file()	15
3.6.4	Function create_pubkey_file()	16
3.6.5	Function encrypt_message()	16
3.6.6	Function decrypt_message()	16
3.7	OAEP_test	17
3.7.1	Function get_byte_lenght()	17
3.7.2	Function padding()	17
3.7.3	Function main()	18
4	Tests	19
4.1	Our tests	19

1 The RSA Cryptosystem

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

$$C \equiv M^e \pmod{n} \quad (1)$$

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

$$M \equiv C^d \pmod{n} \quad (2)$$

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 0.567227840423584 seconds
245923178149220513334023628824617820084665894250209084002738811570947961813241409381655132071011422791176636505370410091
02982136357929464241150325936467631
```

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 = 309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723
876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558
354114671872409831731682201638849145662861185017476396619356672786165549725487427

We have phi(309676006436329745520512297822007310466699729428218128312306431575953501819566727027434662312420495224274723
876998083713056233159626083063845987794063459458836683924879272846616658093907641248713635132077226911286149593982140558
354114671872409831731682201638849145662861185017476396619356672786165549725487427) = 30967600643632974552051229782200731
046669972942821812831230643157595350181956672702743466231242049522427472387699808371305623315962608306384598779406345942
316512257799444060934516585174940869489126512077454176733101557040431276030658254410243500141006479250062289762399020919
3082644250859893710029561760394816

We have e = 112815563475445820758575606745504012989194460209451354867273684671496488894564547975368843527961009740607284
375395020734606272784780347971253950413106990817241569382634152877552570769903927525794113843217127806723692982052967395
170344063967949250637527170043011057683435305568067664182912441620446217240769133

We have d = 272762379630413960323911284421652836128046942743318346475839280552101328594990419509270880180770200635733173
070259923939886030213081147669519851015133572159085594956421189529592279879891792191601908087381337310017153950131574278
729824639985448860915178143423237829727807680792553635609961671421872357257067301

Public key : 30967600643632974552051229782200731046669972942821812831230643157595350181956672702743466231242049522427472
387699808371305623315962608306384598779406345945883668392487927284661665809390764124871363513207722691128614959398214055
8354114671872409831731682201638849145662861185017476396619356672786165549725487427 1128155634754458207585756067455040129
891944602094513548672736846714964888945645479753688435279610097406072843753950207346062727847803479712539504131069908172
415693826341528775525707699039275257941138432171278067236929820529673951703440639679492506375271700430110576834353055680
67664182912441620446217240769133

Private key : 2727623796304139603239112844216528361280469427433183464758392805521013285949904195092708801807702006357331
730702599239398860302130811476695198510151335721590855949564211895295922798798917921916019080873813373100171539501315742
78729824639985448860915178143423237829727807680792553635609961671421872357257067301
```

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 : ['0x14b6342354d67df33ab2950efae489545455a0e04bcff45ef6a067c517947b903438a462217170af7124b8ab1a1153eca652be5b09d1587e8a368ce25b8d9904d6cd55c3d867b63af7752b5ed39abc58bb59e62bfcfe5eb511cfcb4758f7d7beef22b7271af882c5796ed96cb1b455873fa6a7ccc97dcbef14b9c29a2dd50dc2d', '0x1a370c15f39085302c695af6e10600b7a6aa03e4d485cc148fae901a22c7c33a1cc650778980488cda78adab00aed927344679c9c3b51bf7223c83565f595c2bdceca9b97318a3d9520a253e3746d357dd222bf5cd192e9706de7746847326075f624c76f72e7ac07d89380013a93d0409a1cf8dc99fefa8607948a094956c480', '0x35774bbfffb98c200e845d1ea7fb32753445cf48285a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb92d1d31c2b83c926f87f88c607eca88ca41469e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a1d555457a', '0x35774bbfffb98c200e845d1ea7fb32753445cf48285a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb92d1d31c2b83c926f87f88c607eca88ca41469e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a1d555457a', '0xf88a8345405fa1c4b45dd7abe7f1646495a78021536380291aea961f92e01999a4a89be74e78d732f6b41d6b84abd8e9077ce07de177210e791649b6e7c9dc7fd4ecc155c2c1fd71d7c5a9ebd319d587d44470ec59b17b202669251d28aa0c7c7d864518f93ccdd18e213c30e2b7575764688584faf02ed9bd39112afca464b6', '0x198295b0f692271a3bf384297332da0cb35783de6d83f054bc75871c011098a909ded08b65db259cc60d42cddb10b2ac36bdf2b46530b010235ff28b026cb32d494e026f52ce75879c36c35f80a6730641e71116fcfe8afdc3a5fa3541130c51f243400f845f679f6645e2d7649968e3fa5656b653a6cf3469c720a619b7cc67c', '0x6464ae34537412da03582e2f051d061d94854de98a5fca25f3a188d34f8f2dfbfb059ae6d2e2138332ccd02e0e67d5841d22266689e45fc4fa51d3ab52fff391f0ba9663aadffbc8c0b11437c1f70a2cf82ca0e5f449b3dab5136450a49b874ca4e41e081d17987258905171dd2e8e85428269f9e29222b77be29ed859a6d9', '0xf88a8345405fa1c4b45dd7abe7f1646495a78021536380291aea961f92e01999a4a89be74e78d732f6b41d6b84abd8e9077ce07de177210e791649b6e7c9dc7fd4ecc155c2c1fd71d7c5a9ebd319d587d44470ec59b17b202669251d28aa0c7c7d864518f93ccdd18e213c30e2b7575764688584faf02ed9bd39112afca464b6', '0xa3029f113905c4799f3cdd02982920a72430a4d05aec0904717b387940ab3d233d0e7e429395480f87de7b6731efffcaa4ab7d1f71faef402a5dd64d5afe955766e5e606a0f87921dc26101804f36cab5a9560cd8b838a35d87dbaca65a2ae37bf76902a9d91d77dc87eabb488bf7bf2fc4793ab30f1e9fdaea453c2281efbe4b', '0x35774bbfffb98c200e845d1ea7fb32753445cf48285a292a3be5b2795b7960ee63e35efaf677de725e6360b99d1b8c1743de0e5c42707938d30869d37bf82cb92d1d31c2b83c926f87f88c607eca88ca41469e191686382feca5c77e90acc5865417827fa83d37d7bb36bb37fc931b4ab38365604bb0cd8cac86985a1d555457a', '0x171115306128607f52fbd1d5261d05637fa8c124159dc105563a67e54864daea7c04077671d1205ad8abcc815d2e66099e27ca4b995ecc762d34411d3ab79688217df6d559d97424b193c5678bf1b76a502bd416b003da36035fba3aa25bc3cd144a7f01ac45a4404cf4ab865a2f592fc066949dbcea7ef683ab3bcd09b100f', '0x198295b0f692271a3bf384297332da0cb35783de6d83f054bc75871c011098a909ded08b65db259cc60d42cddb10b2ac36bdf2b46530b010235ff28b026cb32d494e026f52ce75879c36c35f80a6730641e71116fcfe8afdc3a5fa3541130c51f243400f845f679f6645e2d7649968e3fa5656b653a6cf3469c720a619b7cc67c', '0x77a5fa22b3f0b1daf007aa3814fd527a688b2d96f7fb4f7bad28ed3bbb13c2be34dc6e161e0d5118b6a6dc80b1536dccc2d26db7c3c1d0e3710d9ceb197c707842428e7a3420f931013e0ca5e3a5c2ba8884be0c61d7d7a1f33752a3dd482d5fe5fc592e791abbbba76a96d7f678ffa2885c70767346bb2a604d10be7af81b8ed']

2.3 Decrypt cipher text

This feature contains several functions.

```

You want to
1 -- Decrypt your previous encryption
2 -- Decrypt extern encrypted text
Enter your choice :

```

You can :

-Decrypt a text cypher previously encrypted by the program.

Dependent on the "encrypt" functionality

```

Enter your choice : 1
The encrypted text : [1475100356561843441391841597642647807470725450575858174622995433171147512534281443091063818799467
79145197042462256021980785698292636964650511712616520443108891637495370094163622203038712111136495317396340551528351204
0329281662805732891618100111874883269620287713741359382849889388542082558595537518525855349, 26967722152022402454108467
170302022407334751473565395662496567032613273974006772344360365846228049826417977249787838815090858768008139785701574841
706757411020964559240691017299159181112543559227067349662290811772008725153297588224410718661952902455735964568559671414
5270260174172392551603025173767513400661754, 189102395733892568943428375856087918414672959405994073700966705048927872459
888431175085635646889799935100716155627845455042353053943472325233583039321337194627112855397715529964728448330709869518
819733118434940893381034745641251825390569891024060721581104513498155944491617541946929040045857020685874874316597, 1891
023957338925689434283758560879184146729594059940737009667050489278724598884311750856356468897999351007161556278454550423
530539434723252335830393213371946271128553977155299647284483307098695188197331184349408933810347456412518253905698910240
60721581104513498155944491617541946929040045857020685874874316597, 26485281628778165716162234213902372791087241799014784
428157712929628054651285420378054478608517771928642322938857091420600039719466694269439839494081456739531827392602851118
901730234631478773776291100954007391165105125723688641490318507636950567693937279856020375549648753684038784106203332250
048804022454308, 1570847391790425621503946170618946799764170035790106338805837701650437282315608076054451286379836745428
997126302611399621954126295899987715779719880738019791217399989462687226981379567794457766350518585355489678403849526568
504148697739978631643391607568144666473833163014307756092974087678877654165700922026, 166825708091058205634453777401333
013402817042299776084597920433837448308646827433417862961892544082118089982673755092614925080400585087430315281497527147
33456849678468582617687776356395378782341414178905028513025064522342377393660375228122950646058672905409746695293171607
599538026489861147051114680373884923, 2648528162877816571616223421390237279108724179901478442815771292962805465128542037
805447860851777192864232293885709142060003971946669426943983949408145673953182739260285111890173023463147877377629110095
4007391165105125723688641490318507636950567693937279856020375549648753684038784106203332250048804022454308, 560497502007
031854331055323656848341308257173975569606544178260644955065839682644127445210139965301439824116985382159079065381499248
983055001880321622393501609106023774516005317870743171971508988220631172563129938123805355307196355121977102549118361379
3993938754092868237350267601033390716011706542563001448, 18910239573389256894342837585608791841467295940599407370096670
504892787245988843117508563564688979993510071615562784545504235305394347232523358303932133719462711285539771552996472844
833070986951881973311843494089338103474564125182539056989102406072158110451349815594449161754194692904004585702068587487
4316597, 133226359154584474307632477101314447720413720106965987513776138754522901644224558458167948305326224199404016609
080428938454573161617241201329493278738196946383246227208421341348069643837826103403367541274552590772152242248367009217
025118957998859210890898493886853680892128664167523648575855940745481643898022, 1570847391790425621503946170618946799764
170035790106338805837701650437282315608076054451286379836745428997126302611399621954126295899987715779719880738019791217
3999894626872269813795677944577663505185853554896784038495265850414869773997863164339160756814466647383316301430775609
2974087678877654165700922026, 508910035421819516849844088600481772244219060421596173388212816118473734627273410286843796
441119895171781561699022054062098351766599778820471476439913192770761786190365513236209650489917655308092671038154811209
53834272453565781940060591112715065546132731477519979181091647045653976821260153256488519075566600]
The plain text is : Hello World !

```

-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

```

Enter your choice : 2
Enter d : 21989689626700929423236052533900404845237195726057180846601910760748898460522609291398243211947708813410592384
963386752736257563789280355219264464673968591065894647510338915699000914301501762359604886217481544703789358974335577930
2608056724894183222934346434105924159449947503059249586730765842894349225232031
Enter n : 3137496710405441623240418954095343794177538545249557776342752010678163634049808129875524358662459810530603233
110397143922773384498950199266794548130624696407364730414020351809649507931869455854979153601230211308158413556686743163
7905768967100354002472997299784499993592185597187807294994171723397557720559219
Message to decrypt : 147510035656184344139184159764264780747072545057585817462299543317114751253428144309106381879946779
145197042462225602198078569829263696465051171261652044310889163749537009416362220303871211113649531739634055152835120403
29281662805732891618100111874883269620287713741359382849889388542082558595537518525855349

```

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.

```

0:d=0  hl=4 l= 732 cons: SEQUENCE
4:d=1  hl=2 l=  1 prim: INTEGER           :00
7:d=1  hl=3 l= 129 prim: INTEGER           :02C5FAE766A35BF5293313EBEFE7A55BD8921F734B4C334D3260FBBDD4E078996798BB9A3
6B076498271D884B34F751B5BD94CDD8B52CA0BF3E4F77B7C68307BC3C9F68932B052ABBE434C6AB6419CF3771A88A9B81322774F4E6669F6D088628
8F703C620A8A36B5562DB6CE7F79474D48590BAD5CFE6D4FE8477A3BA5A86CA19
139:d=1  hl=3 l= 128 prim: INTEGER           :27502459C62AD176C142D90C9F3367944B939C55F8C985AF4EE7527DF5DD14E67A66A76C
99C247D82D7C128D946E7B802343D5A0FC9B6DD5541C058E3A61861AE632AF24FB9803541DC0370625CE1AAB83D08EABFE411B7D11B59D9BC9A200
12E740384D48511E835787A5EE96E6E3C569D0814744B852E09D800368D721FE5
270:d=1  hl=3 l= 129 prim: INTEGER           :01DC6D522D36ACC5DA034001D03CE0B893D55304EAD47FE56A58F15920AA23247DBF7FA
C9E88D18E1CB58B9B189070F0633619D8B3396D8601ED9C8C98E513B0E1BC5A7C079A70B485576110EEAE2096A39B9DE1C79C8EA46EE95E1CD05B8B
471E8430E7841FCBDAEE14626C2B1DEB553906C48213048A3738A345CB8579371
402:d=1  hl=2 l=  65 prim: INTEGER           :0176FDC937710A854CDBDFC384A1A428476F6A295DEDB31ABD0798013FA20D23450C477
BAE61B5133BD61CCA5641E02B808AA64FE9BA2D26D47B82F66351B93C3
469:d=1  hl=2 l=  65 prim: INTEGER           :01E4B0E04281BADDE5BA81C5778D3343CC10EAE247F3D7141FB6C78210992216D023BD3B
61A76AB48C6CC6AC2951B2436FD5F2CFBEA5991682C47D1AF9D2DFD8F3
536:d=1  hl=2 l=  65 prim: INTEGER           :0120F5115F13A2082804342CA7B8F88BAD972775847516C287985F9FFCCB8FC007740BD5
FCB4B21D783A5FF56860E90233B2452529C9F2EF4F6D410CA408D85A14B
603:d=1  hl=2 l=  65 prim: INTEGER           :F5830C65A1AB615BDA67E51A2AB92CE0F06507937C71A3098475EDC1CB4E6493C14DB7E
C38D26E47742EB4E8BAC2139FF3B99417123E1F6F5D512A78EF0B53
670:d=1  hl=2 l=  64 prim: INTEGER           :029E34A068518279FA6E871278FAE45D970CFB7933E83C623C5767249151B1A093D2C092
25CAC11A4F5AB4F58C7328A9E4FF70F5C67A9D023ED09982DAA5CE0
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIC3AIBAAKBgQLF+udmo1v1KTMtG+/npVvYkh9zS0wzTTJg+710B4mWeYu5o2sH
ZJgnHYhLNDRtb2UzduFLKC/Pk93t8aD87w8n2iTKwUqu+Q0xqtKgc83caiKm4Ey
J3T05mafBqIGKI9wPGIKiJa1VwLbbOf3LHTUHZC61c/m1P6Ed606WrbKGQKBgcDQ
JFngKtF2wULZDJ8zZ5RLk5xV+WmFr07nUn313RTmemaNbnJnCR9gtfBKNLG5g7CND
1doPybbd1VqcBY46YYa5jKvJPuYA1qndDcGJc4aq7PQjqv+Qrt9EBwmd8miABLn
QDhNSFEeg1eHpe6W5uPFadgUdUe1LgnbAdANch/LAoGBAdxtUi02rMXNoDQAHQPO
C4k9VTB0rUf+VqWPFZIKoJH2/f6yieNG+HLW4mxiQcPBjNhnYszlthGhtnIyJjL
E7DhVfP8BspwtIVXYRDurICWo5ud4cecjqRu64c0Fu4Rx6EM0e0H8va7uFGJsKx
3rVTkGxIIT8Lo30KNFY7V5NxAkE8dv3JN3EkTuzb2/w4ShpChHb2opXe2zGr0HmA
E/og0jRQxHe65htRM71hzKVkHgK4CKpk/pCi0m1Huc9NMruTwwJBAAeSw4EKuBut3l
uoHFD40zo8wQ6uJH89cUH7bHghCZiHbQI707YadqtIxsxqwpUbJDb9Xyz76lmRaC
xH0a+dL f2PMCQEQeg9RFfE6KwKAQ0LKe4+IutLyd1hHUWwoeYX5/8y4/AB3QL1fy0
sh140L/1aGDpAjoYRSUpyfLvT21BDKSNhaFLAeA9YMMZaGrYVvaZ+UaKrknsng8G
UHK3xxowmEde3By05kK8Fnt+w42Cbkd0LrTousITn/O5l8cSPH9vXVEqe08LUwJA
Ap40GhRgnnb6oc5ePrkXZcM+3kz6DxiPfdnJJFRsaCT0sCJcrB8G9auk9YxZKK
nk/3D1xnqdAj7QmYLapc4A==
-----END RSA PRIVATE KEY-----

0:d=0  hl=4 l= 266 cons: SEQUENCE
4:d=1  hl=2 l=  1 prim: INTEGER           :00
7:d=1  hl=3 l= 129 prim: INTEGER           :02C5FAE766A35BF5293313EBEFE7A55BD8921F734B4C334D3260FBBDD4E078996798BB9A3
6B076498271D884B34F751B5BD94CDD8B52CA0BF3E4F77B7C68307BC3C9F68932B052ABBE434C6AB6419CF3771A88A9B81322774F4E6669F6D088628
8F703C620A8A36B5562DB6CE7F79474D48590BAD5CFE6D4FE8477A3BA5A86CA19
139:d=1  hl=3 l= 128 prim: INTEGER           :27502459C62AD176C142D90C9F3367944B939C55F8C985AF4EE7527DF5DD14E67A66A76C
99C247D82D7C128D946E7B802343D5A0FC9B6DD5541C058E3A61861AE632AF24FB9803541DC0370625CE1AAB83D08EABFE411B7D11B59D9BC9A200
12E740384D48511E835787A5EE96E6E3C569D0814744B852E09D800368D721FE5
writing RSA key

```

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.

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 $e \bmod \phi$.

```
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 = 43727260620659110686125603313607868761982540601789555193928930871591938467934857037
    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) = 42671649065942098496583845013128680
471998403704245731353409818832244811788290512905712735936531047719147029892689691681011301876226277110495080256178444702
405700004286081447383185505246375223317791298516858584368154450196225371819560986408893760932682328046374045461122411314
0435074418333420961692175571708160

We have e = 335751174767406032515911945451600644338809095228508520269064632861820517840052780042109591748053318178384564
644939679291530670893250404542055496334841813239667686795407862595390257490354181589140156790163399200372699874404236105
763141148003014066144823574146478099601294767661494448233177333219499379185121629

We have d = 205501972305576293802518703017590764356274730385356494882275549150626264830384248287017541961799700484113176
255066992951006815419960524127171238188481680944779765989232893891635542274046074404325845457564866011232532167975693386
732561230194326712516055711781416193423187380339444166368713587873567140112446709

Public key : 42671649065942098496583845013128680471998403704245731353409818832244811788290512905712735936531047719147029
892689691681011301876226277110495080256178444706550095083693950126676340972456539622193055540147442070460237894427869874
0690822282233008910631261152597643367479356106476579312138019752228254241784921813 3357511747674060325159119454516006443
388090952285085202690646328618205178400527800421095917480533181783845646449396792915306708932504045420554963348418132396
676867954078625953902574903541815891401567901633992003726998744042361057631411480030140661448235741464780996012947676614
94448233177333219499379185121629

Private key : 2055019723055762938025187030175907643562747303853564948822755491506262648303842482870175419617997004841131
762550669929510068154199605241271712381884816809447797659892328938916355422740460744043258454575648660112325321679756933
86732561230194326712516055711781416193423187380339444166368713587873567140112446709
```

We then generate keys with OpenSSL.

```
-----BEGIN RSA PRIVATE KEY-----
MIIC3QIBAAKBgQJfqhmdYkNNQ+sm1sZN53Ngj2CgcwkcfwgqXDPx2FGQ29A9e4rp
8TYtVVOjxqzjADT8f1Yn/D30cjLqaM4F22XWx6keyHSsvLKbmQBp+9yGdutCd81e
s/DyS5FbMcm5yX1k1yHkXoBrEiWzr4D0vEUT/EkqDKq9CvxKqcXLjbd+1QKBgQHe
IC0gxX/OFDSn6dXFN/CT7b45LvRLa2DONsD65RPbMYsRlC7xd4unyipdpOUTo0yM
rcZPZSkX8c3prU0lWmmZmWF3ftHOKZbETS7JyliRHY58NPY5I9W5m/beSfwL6qyf
1RimFpyhtkTiYJOSFu3g57mlAmUn4+Ff08Ssb3N5XQKBgQEkpP9tqAVEotVAki8s
/j01Tysp6rDu9tlzZivSEwLCQQuWeLy0aYJOT71AZsp28kjRDYhfViRBxYflhMPd
Fj/GsQOpnoLMUEt08VeFVBpg8jq0Ju1NgSjWwvY6lyyNgM7iUn3I2YSCi9lgsntZ
WxnGQMfnbbliXPmL+geUP8c09QJBAWxfckLWEgWeOrcpmnlpuVh8NxYhXF3NsvUQ
YhcfPn+CXZnryVQokgjJ+809R0QFkr/h30lDAVAr3Sw2PTwhWJUCQQGq7l8qWhC+
tZbnocp5aVFGUtp33dv0s2qt9H3yRXAd905CLFj7wwTEcPjYoVqQZhoEip/p6KVO
WqfImk4tJj1BAkEA1RvYTYZvVvX1n8y3U8YiDXohdeepNh3Z44J6OZ2iaf4Ludtb
tCQkozw+LakNqG3+l++EltN1HwDCs3RyUbfjMQJAYGHbzYIjExqztH0+jp/y6sVF
Tgf0xX66lRvPNMV6nY+4H6xEDOFju0BCzfOWGijPH5qm3qvZp2WbNry7HdB9dQJB
AL0+vWeTFJ6geUyFsixe07KPYCe5lfh1iBfynRSd3F+SglLM/w5EraSnD3hfDGTy
1Z3Dkc1V0X1Yfg5mJge0FPw=
-----END RSA PRIVATE KEY-----
```

Figure 1: Private key stored in prikey.pem

```
-----BEGIN PUBLIC KEY-----
MIIBIDANBgkqhkiG9w0BAQEFAAOCAQ0AMIIBCACBgQJfqhmdYkNNQ+sm1sZN53Ng
j2CgcwkcfwgqXDPx2FGQ29A9e4rp8TYtVVOjxqzjADT8f1Yn/D30cjLqaM4F22XW
x6keyHSsvLKbmQBp+9yGdutCd81es/DyS5FbMcm5yX1k1yHkXoBrEiWzr4D0vEUT
/EkqDKq9CvxKqcXLjbd+1QKBgQHeIC0gxX/OFDSn6dXFN/CT7b45LvRLa2DONsD6
5RPbMYsRlC7xd4unyipdpOUTo0yMrcZPZSkX8c3prU0lWmmZmWF3ftHOKZbETS7J
yliRHY58NPY5I9W5m/beSfwL6qyf1RimFpyhtkTiYJOSFu3g57mlAmUn4+Ff08Ss
b3N5XQ==
-----END PUBLIC KEY-----
```

Figure 2: Public key stored in pubkey.pem

References

- [1] <https://security.stackexchange.com/questions/176394/how-does-openssl-generate-a-big-prime-number-so-fast>
- [2] <https://gist.github.com/Ayrx/5884790>
- [3] <https://www.alpertron.com.ar/ECM.HTM>
- [4] <https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html>
- [5] <https://mathworld.wolfram.com/FermatsLittleTheorem.html>
- [6] <https://artofproblemsolving.com/wiki/index.php/Euclid>
- [7] [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- [8] https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding
- [9] <http://www.man-linux-magique.net/man1/rsautl.html>
- [10] <https://stackoverflow.com/questions/19850283/how-to-generate-rsa-keys-using-specific-input-numbers-in-openssl>
- [11] <https://datatracker.ietf.org/doc/html/rfc8017>
- [12] <https://datatracker.ietf.org/doc/html/rfc2313>
- [13] <https://www.rfc-editor.org/rfc/rfc3447>
- [14] <https://inventwithpython.com/cryptomath.py>
- [15] <https://www.tutorialspoint.com/what-are-risks-in-implementing-the-rsa-algorithm-without-padding>
- [16] https://github.com/mimoo/RSA_PKCS1v1_5_attacks/blob/master/bb98_graphic.sage
- [17] <https://datatracker.ietf.org/doc/html/rfc5208>