

CS4150: Computer Networks Lab

Lab11

111901030

Mayank Singla

Q1. Write a code to generate the public and private keys, using RSA, for two users A and B. Store these keys as ASCII in the files A.pub, A.pri, B.pub, and B.pri.

I created the function **genRandom()** to generate a random number in the given range.

I created the function **nBitRandom()** to generate a random number having n-bits. For this, I picked a random number in the range $[2^{n-1} + 1, 2^n - 1]$

I created the function **isPrime()** which checks if the given number is prime or not in $O(\sqrt{n})$. It simply loops over all the odd numbers less than \sqrt{n} and checks for any factor of n other than 1.

I created the function **genPrime()** which generates a random number of the given number of bits. For this, I refer to the method given [here](#) according to which we first create a random n-bit number and set its 1st bit to make it odd and set its highest 2 bits to make n with double the bits which will be used later. Then, we incremented the random number generated by 2 until we get a prime number.

I created the function **genPrimePairs()** which creates two random prime numbers p and q for the RSA algorithm. I choose a fixed key length k for the number n. According to the link specified, p should have a length of k/2 and q should have a length of (k - k/2) for n to have a length of k.

I created a **Triplet** class to store x, y, and gcd for solving the equation $ax + by = \text{gcd}(a, b)$ for given a and b.

Then, I created the function **extendedEuclid()** which solves the above equation for the given values of a and b. This is a recursive function that uses the standard Euclid algorithm to solve the equation.

I created the function **modInv()** which computes the multiplicative modulo inverse for given numbers a and m i.e. solve for x in the equation $(a \cdot x) \bmod m = 1$. From the standard number theory, we know this problem is similar to solving for x in the equation $ax + my = 1$ given that $\text{gcd}(a, m) = 1$. For solving this, I used the extended Euclid algorithm using the function created above.

I created the function **genPublicPrivateKeys()** which creates the public-private key tuple (N, e, d) using the RSA algorithm. The idea is from what was given in the link mentioned above. It selects the value of e first from a given set of primes. Then, it generates the primes p and q till $(p \% e == 1)$ and $(q \% e == 1)$. This is to make sure that the totient function $\phi = (p-1) * (q-1)$ is co-prime with e. Then, we calculate the multiplicative modulo inverse of (e, Phi) to get d.

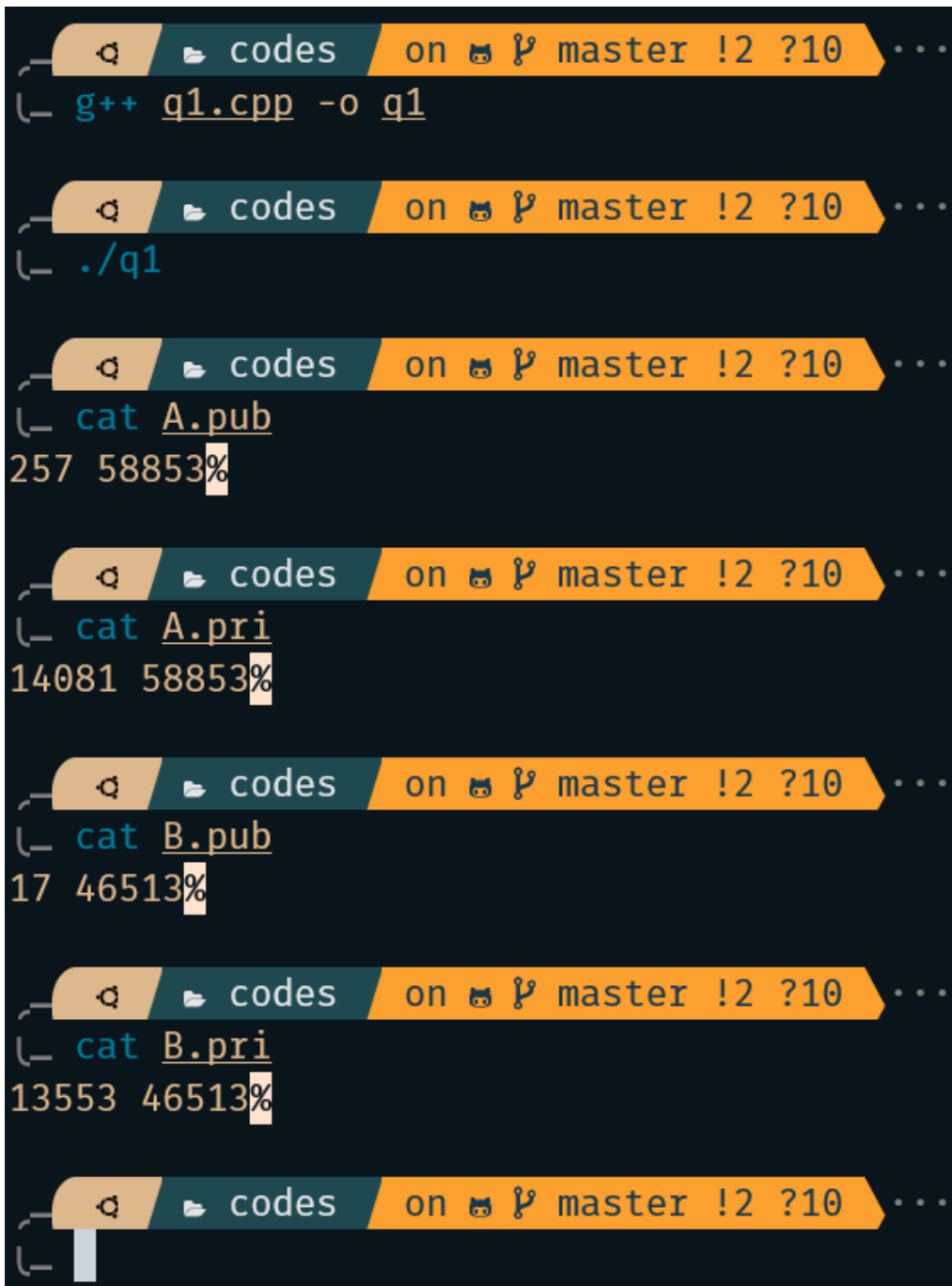
Finally, I created the function **createRSAKeys()** which creates the public-private RSA keys and stores them in the given public and private files. I then call this function for user **A** and user **B**.

Here is the sample output.

(Here I have taken the key length as **16** due to the maximum value of **int** in C++)

The **.pub** file contains the pair (**e**, **n**)

The **.pri** file contains the pair (**d**, **n**)



```
codes on master !2 ?10 ...
└─ g++ q1.cpp -o q1

codes on master !2 ?10 ...
└─ ./q1

codes on master !2 ?10 ...
└─ cat A.pub
257 58853%

codes on master !2 ?10 ...
└─ cat A.pri
14081 58853%

codes on master !2 ?10 ...
└─ cat B.pub
17 46513%

codes on master !2 ?10 ...
└─ cat B.pri
13553 46513%

codes on master !2 ?10 ...
└─
```

Q2. A wants to digitally sign a message and securely send it to **B**. Write a code that will sign and encrypt the ASCII text in file `message.txt`. Store the signed and encrypted text as ASCII in the file `secret.txt`. *Note: You are required to use the keys generated in the previous question.*

I created the function `pw()` which calculates $(x^y) \% m$ using the standard binary exponentiation method.

I created the function `msgDigest()` which calculates the message digest for the given character. It simply multiplies the ASCII value of the character with a chosen prime number. It acts like a hash function for the message.

I created the function `sign()` which reads a text message from a file, computes its message digest, and signs it based on the given private key. It computes this for each character of the text message separately. The signed message is created as $s = (MD(p) ^ d) \% n$ where p is the character and d and n are taken from the private key.

I created the function `encrypt()` which encrypts the given signed messages with the given public key and stores it in the given secret file as space-separated. The encrypted message is created as $c = (s ^ e) \% n$ where s is the signed message and e and n are taken from the public key.

I created the function `signAndEncrypt()` which signs and encrypts the message given the public and private keys to use and stores the encrypted message in the given secret file. It basically makes use of the functions defined above. The important thing here is that the value of n could be different for user **A** and user **B**. We need to sign with the key having less value of n and then encrypt with the key having a greater value of n . This is because while doing $\%$ operation on the other side, we can go from bigger modulo to smaller but not the other way around.

Overall, user **A** then sends the secret message to user **B** as the pair $M, E_b(D_a(MD(M)))$ (assuming $n_b \geq n_a$) where the first part is the original message stored in `message.txt` and the second part is the digitally signed and encrypted message stored in `secret.txt`.

Here E_a and D_a are the public and private keys of user **A**, and

E_b and D_b are the public and private keys of user **B**

Here is the `secret.txt` created using the public-private key pairs generated in the previous question for the shown message.

```
codes on P master !2 ?10 at 08:20:45 PM
└─ g++ q2.cpp -o q2

codes on P master !2 ?10 at 08:20:55 PM
└─ cat message.txt
Good Morning!
Have a nice day...

codes on P master !2 ?10 at 08:21:19 PM
└─ ./q2

codes on P master !2 ?10 at 08:21:24 PM
└─ cat secret.txt
40636 31382 31382 40723 43590 13708 31382 27753 17540 45761 17540 26440 38753 509 57912 54406 20246 44536 43590 54406 43590 17540 45761 52686 44
536 43590 40723 54406 24140 56856 56856 56856 509%
```

Q3. Write a code that will decrypt the message in `secret.txt` and verify if the message was indeed sent by **A. If the verification is successful, your code should print the original message from **A**. Else, your code should print the text `Message not verified`. Note: You are required to use the keys generated in the first question.**

I am having the same **pw()** function and the same **msgDigest()** function to check for message digest validation which is retrieved. The **msgDigest()** function must be the same as that on the sender side as it is acting like a hash function.

I created the **reverseDigest()** function to print the received ASCII text from the received message digest.

I created the function **decrypt()** which reads the secret message from the secret file and decrypts it using the given private key. It decrypts each space-separated message in the secret file. The decrypted message is retrieved as $s = (c^d) \bmod n$ where **c** is the encrypted character and **d** and **n** are taken from the private key.

I created the function **unsign()** which un-signs the given decrypted messages using the given public key. It un-signs each decrypted message as $md = (s^e) \bmod n$ where **s** is the decrypted character and **e** and **n** are taken from the public key.

I created the function **validateMsgDigest()** which validates the retrieved message digest above with the computed message digest of the received message file. If for all the characters, the message digest is the same, then the message is digitally verified and valid else it is not.

I created the function **decryptAndUnsign()** which decrypts and un-signs the received secret message using the given public and private keys and validates it with the received message file. If the message is verified, then it performs reverse digest and prints the original message. (While I could also simply print the contents of the received message file which should be ideal as both are the same). This function also reverses the order of keys used for decryption and un-sign based on the value of **n** of the keys as done on the sender side.

Overall, user **B** gets the pair $M', H = E_b(D_a(MD(M)))$ (assuming $n_b \geq n_a$)
 It decodes it as $MD(M) = E_a(D_b(H))$
 and checks if $MD(M') = MD(M)$

Here is the retrieved message using the **secret.txt** created in the previous question. As the **message.txt** file was not modified, we were able to successfully retrieve the original message.

```
codes on master !2 ?10
└─ g++ q3.cpp -o q3

codes on master !2 ?10
└─ ./q3
Received Msg: Good Morning!
Have a nice day...
```

Below, we can see on changing the original message in **message.txt**, the validation of the digital signature fails.

```
codes on master !2 ?10
└─ vim message.txt

codes on master !2 ?10
└─ cat message.txt
Good Evening...
Have a nice day!

codes on master !2 ?10
└─ ./q3
Message not verified
```