# Cryptography Homework 4

## Required Reading

Cryptology 4 slides

## Computing Keys

This lab allows us to play with the math behind RSA encryption. It is “schoolbook RSA”, and is \*not\* safe to use for encrypting data outside of teaching labs and CTFs. Remember that real RSA protects itself against attacks using PKCS#1 hashing and padding.

To gain more practice in Python, we will compute exponents (d, in the slides) for a private key modulo Λ(n), Φ(n), and n. The first two should work, and the last one using modulo n should not (if it did, RSA would be broken.) We will use the procedure below for all three attempts. Fill in your results in the Turn In section.

### Alice Generates a Key

1. Use prime numbers p = 131 and q = 157
2. Compute n = p \* q
3. Compute either
   1. Λ(n) = lcm((p – 1)(q – 1)) OR (your choice)
   2. Φ(n) = (p – 1)(q – 1))
4. Pick a small number for e, for the public key
5. Compute gcd(e, Λ(n)) or gcd(e, Φ(n)), depending on what you did in step 3. The GCD must equal 1, which means e and Λ(n) or Φ(n) are relatively prime.
6. Compute the number, d, for the private key from d = findModInverse(e, Λ(n)) or d = findModInverse(e, Φ(n) ). Note: if findModInverse runs, but does not give you a value for d, it probably means that the GCD is not 1.
7. Alice gives public key [n, e] to Bob, keeps private key [n, d] secret

### Bob Encrypts a Message

1. Pick an integer, plaintext, that is less than n, that will represent Bob’s message. We will pretend that it decodes into a few ASCII letters.
2. Bob encrypts m with Alice’s public key using ciphertext = pow(plaintext, e, n)
3. Bob gives ciphertext to Alice

### Alice Decrypts the Message

1. Alice computes plaintext = pow(ciphertext, d, n)
2. The answer should be the same as the plaintext Bob started with

## Repeat with n, doomed to failure

Let us assume that an attacker has n and e from the public key, but they do not have p or q so they cannot compute Φ(n) or Λ(n). Since they are wishful thinkers, they try to compute the private exponent d using findModInverse(e, n). This should fail miserably. If it succeeds, it will mean that RSA encryption doesn’t work.

What does failure look like? If you encrypt a message and then try to decrypt it using an incorrectly computed d, the decrypted message will be entirely different from the original message.

# Turn in

This data is the same for both attempts:

p = 131

q = 157

n = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ p\*q

### Attempt 1, using Λ(n) or Φ(n) (this should work—decrypted = original plaintext)

Λ(n) or Φ(n) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ lcm(p-1, q-1) or (p-1)\*(q-1)

e = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pick this yourself

gcd(e, Λ(n)) or gcd(e, Φ(n)) = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ must equal 1. If not, pick a new e

d = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ findModInverse(e, Λ(n)) or findModInverse(e, Φ(n))

public key [n, e] \_\_\_\_\_\_\_\_\_\_\_\_\_\_

private key [d, e] \_\_\_\_\_\_\_\_\_\_\_\_\_\_

plaintext integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ pick this yourself, < n

ciphertext integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ pow(plaintext, e, n)

decrypted integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ pow(ciphertext, d, n)

### Attempt 2, using n (this should fail—decrypted not equal to plaintext)

n = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ n, same as above

e = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Use the same e that you did in Attempt 1

d = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ findModInverse(e, n)

public key [n, e] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

private key [d, e] \_\_\_\_\_\_\_\_\_\_\_\_\_\_

plaintext integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Use the same plaintext that you used in Attempt 1

ciphertext integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ pow(plaintext, e, n)

decrypted integer \_\_\_\_\_\_\_\_\_\_\_\_\_\_ pow(ciphertext, d, n)

## Notes

The formula for Λ is Λ = lcm(p-1, q-1). The lcm function is not in the cryptomath.py file from <https://nostarch.com/download/CrackingCodesFiles.zip>. You can add it if you like, or just manually use this formula for lcm(a, b).

(a \* b)//gcd(a, b)

This works for lcm(a, b):

def lcm(a, b):

# Return the Least Common Multiple of a and b

if a and b:

return abs(a // gcd(a, b) \* b)

else:

return 0

## Python Hints

When you import a file into Python, you do not use the extension. You use this statement to import the file cryptomath.py:  
 import cryptomath  
If you import cryptomath that way, you have to preface every function call with cryptomath, for example:  
 cryptomath.gcd(a, b)

To lessen the typing load, you can import the file this way:  
 import cryptomath as cm  
Then call functions this way:  
 cm.gcd(a, b)

To lessen the typing even more you can import the functions you want by name:  
 from cryptomath import gcd, lcm  
Then you can call them without a prefix:  
 gcd(a, b)  
 lcm(a, b)