RSA algorithm

Encryption Techniques and Systems

Jere Pesonen M3227, TTV18S1

Course assignment
November-2020
Information and Communication Technology

Sisällys

1 Introduction

In this assignment, I am implementing a working RSA algorithm, and an RSA Broadcast attack for attacking and decrypting the encrypted message. Instruction for these challenges can be found from https://cryptopals.com/sets/5, challenges 39 and 40. Every code is written in Python language.

Through this document, I will shortly brief the operations of working RSA algorithm and mathematics behind it. Then ill go through the cryptopals tasks, their requirements, and my own implementations.

This assignment is a part of Encryption Techniques and Systems course.

2 RSA algorithm

The RSA algorithm is asymmetric cryptography algorithm, which means that encryption and decryption are performed with different keys. These keys are generated with math's by examining numbers congruences and remainders.

For the start you will need two prime numbers called p and q. Larger numbers provide safer encryption. Multiplying them by each other, we get "n".

$$\mathrm{n} = \mathrm{pq}$$

Next, you need to count of existing numbers that are coprime to "n" and are between 1, and "n". This can be achieved with simple function called Euler's totient. In Euler's totient, we need the factors of "n", decreasing them by one, and then multiply them by each other. In this case the two factors are "p" and "q" of course.

$$et = (p - 1)(q - 1)$$

Next you need an integer e, which includes in the public key. This integer must be coprime to et, and it must be between 1, and et. In the cryptopals challenge, "e" is predefined to 3.

Last integer we need, is “d”, which is part of private key. “d” Has to satisfy following function. So, “d” multiplied with e should not have any common factors with “et” (except 1 of course).

$$de \ \equiv 1 \, mod(et)$$

By following those functions, we can compute our public and private keys, so that asymmetric encryption works. Public key is (e,n), and private key is (d,n). The “d”, “p” and “q” must all remain secret, because “p” and “q” are the factors of “n”, and they can be used for computing “d”. In functions “m” stands for plain message, and “c” stands for ciphertext.

The encryption function:

$$m^e \, mod(n)$$

The decryption function:

$$c^d mod(n)$$

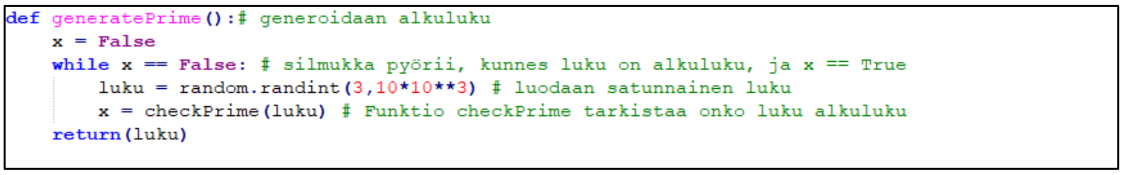
3 Implementation

3.1 RSA

There are couple main parts, which need little more thinking when implementing RSA: Generating prime numbers and creating inverse mod function. The encryption and decryptions are rather easy, but generating working numbers, and implementing a working code, that is the hard part.

3.1.1 Prime generating

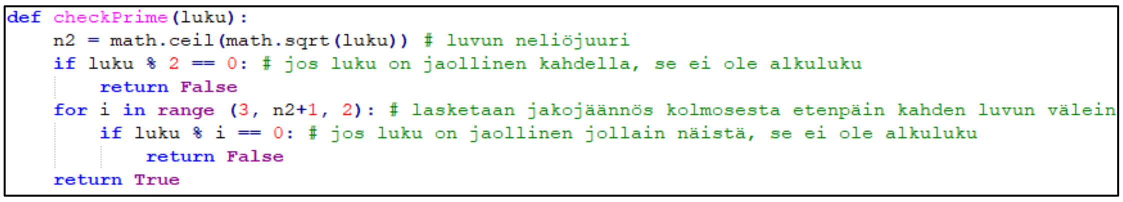
The first thing I started with, was creating a function, that checks if certain number is prime. I used two functions, (Figure 1 and Figure 2Figure 2. checkPrime) where other one just randomly generates random integer, and other one checks if it is prime number. Prime numbers generated are between 3, and 10000, because the checkPrime function is too slow for bigger numbers.



```
def generatePrime():# generoidaan alkuluku
    x = False
    while x == False: # silmukka pyörii, kunnes luku on alkuluku, ja x == True
        luku = random.randint(3,10*10**3) # luodaan satunnainen luku
        x = checkPrime(luku) # Funktio checkPrime tarkistaa onko luku alkuluku
    return(luku)
```

Figure 1. generatePrime function.

The checkPrime (Figure 2) function works so, that firstly it checks if it given number is even, checking if it's divisible by 2. Next, it calculates the square root of given number. For loop goes through every odd number between 3 and the square root. If the number is divisible with any of these numbers, it means that it's not prime, and function returns: "False". The other case is when number is not divisible by any of tested numbers. Then the number is prime, and function returns "True"



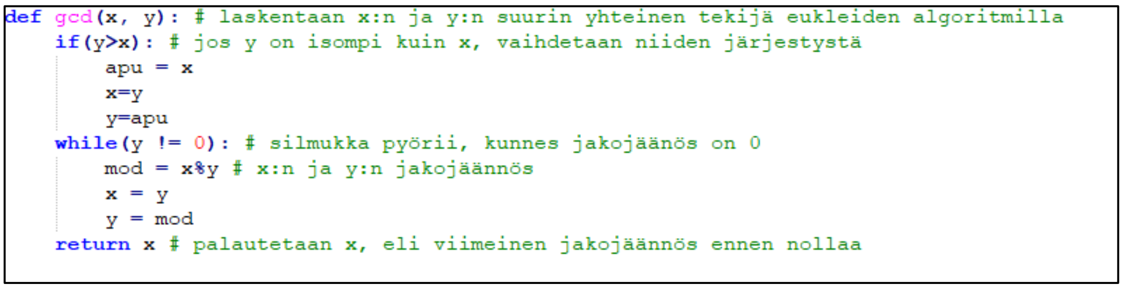
```
def checkPrime(luku):
    n2 = math.ceil(math.sqrt(luku)) # luvun neliöjuuri
    if luku % 2 == 0: # jos luku on jaollinen kahdella, se ei ole alkuluku
        return False
    for i in range (3, n2+1, 2): # lasketaan jakojäännös kolmosesta etenpäin kahden luvun välein
        if luku % i == 0: # jos luku on jaollinen jollain näistä, se ei ole alkuluku
            return False
    return True
```

Figure 2. checkPrime function.

3.1.2 Greatest common divider

The gdc function (Figure 3) is used to calculate greatest common divider between to integers ("x" and "y"). First operation is to make sure that greater number of given two, is the assigned to "x". If not, they are switched between each other.

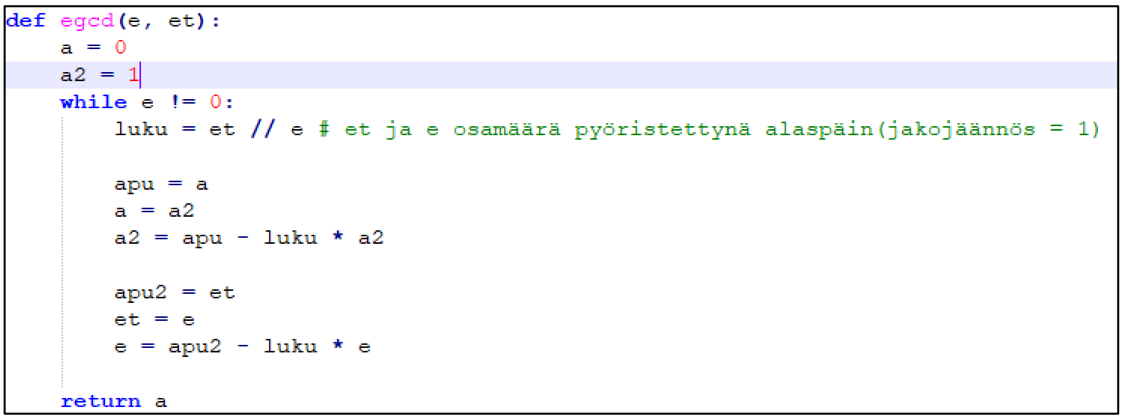
After that the greatest common divider is calculated with Euclidean algorithm. Algorithm works in a while loop, where Variable "y":s value is set to x, and The Remainder from "x" divided with "y" is set to variable "y". This continues until remainder is 0, then the greatest common divider is saved to variable "x".



```python
def gcd(x, y): # laskentaan x:n ja y:n suurin yhteinen tekijä eukleiden algoritmilla
    if(y>x): # jos y on isompi kuin x, vaihdetaan niiden järjestystä
        apu = x
        x=y
        y=apu
    while(y != 0): # silmukka pyörii, kunnes jakojäänös on 0
        mod = x%y # x:n ja y:n jakojäännös
        x = y
        y = mod
    return x # palautetaan x, eli viimeinen jakojäännös ennen nollaa
```

Figure 3. Calculate greatest common divider with Euclidean algorithm.

The extended Euclidean algorithm is used for generating the private key. It calculates a multiple of a certain number when divided by Euler’s totient number and the remainder must be 1. This is a bit more complicated piece of code, but for making it simple, it divides et with “e”, and rounds it down (et and e should be coprime as mentioned before.). Then the numbers are rotated between variables. While loop runs until “e” is 0, then. I made a demo of this algorithm in excel that is found in attachments.

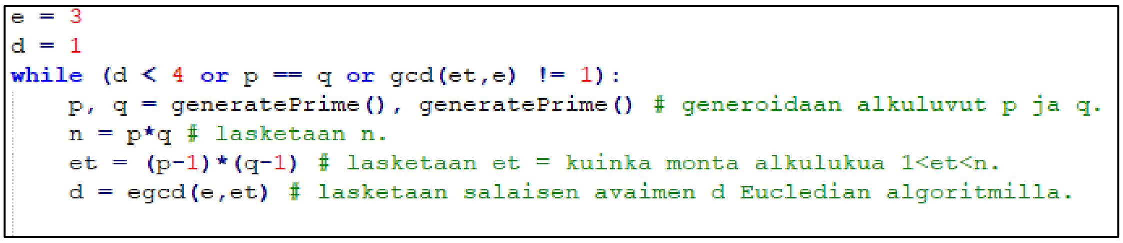


```python
def egcd(e, et):
    a = 0
    a2 = 1
    while e != 0:
        luku = et // e # et ja e osamäärä pyöristettynä alaspäin(jakojäännös = 1)

        apu = a
        a = a2
        a2 = apu - luku * a2

        apu2 = et
        et = e
        e = apu2 - luku * e

    return a
```

Figure 4. Extended Eucledian algorithm.

As staged before, n and et are simply calculated. In this challenge, the variable “e” is pre-defined as 3. At this point, there is a catch. Since the variable “e” is 3, and e and et cannot share any common factors. Not all prime numbers “p” and “q” are going to work with this condition.

So, I made while loop, which checks the mandatory relations between numbers (Figure 5). Every time, the while loops conditions are not met, it generates new prime numbers, and goes through the whole process again. Variable d cannot be smaller than 4(it cannot be 1, or same as “e”, and 2 would not work either.), p and q cannot be equal and the greatest common factor between e and et must be 1.

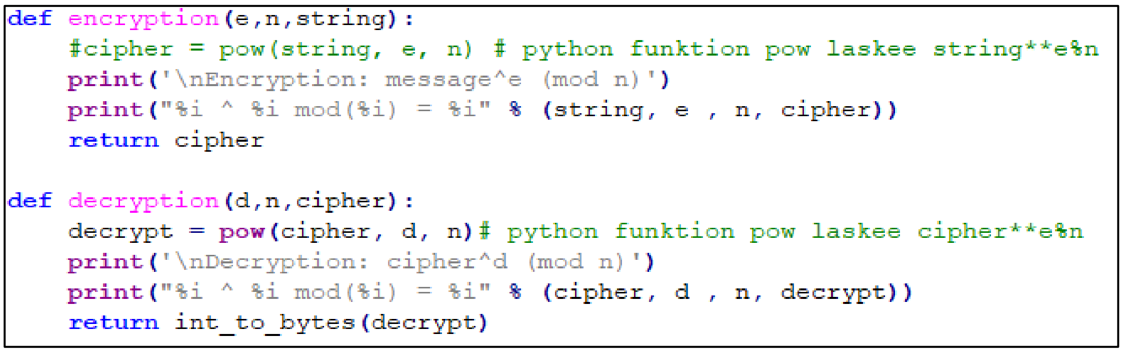


```
e = 3
d = 1
while (d < 4 or p == q or gcd(et,e) != 1):
    p, q = generatePrime(), generatePrime() # generoidaan alkuluvut p ja q.
    n = p*q # lasketaan n.
    et = (p-1)*(q-1) # lasketaan et = kuinka monta alkulukua 1<et<n.
    d = egcd(e,et) # lasketaan salaisen avaimen d Eucledian algoritmilla.
```

Figure 5. Loop, that checks that number relations are what they should.

3.1.3 Encryption/Decryption

Finally, we just need functions for encrypting and decrypting our message (Figure 6). As seen, these are just simple calculations, and should work if the number relations are working. The math is done with python build in function “pow”, which works same as just basic operation of “plaintext^e % n” only faster.

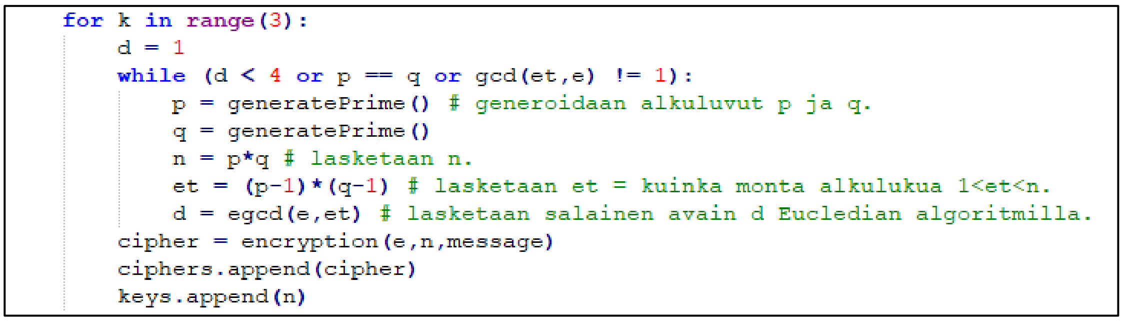


```
def encryption(e,n,string):
    #cipher = pow(string, e, n) # python funktion pow laskee string**e%n
    print('\nEncryption: message^e (mod n)')
    print("%i ^ %i mod(%i) = %i" % (string, e , n, cipher))
    return cipher

def decryption(d,n,cipher):
    decrypt = pow(cipher, d, n)# python funktion pow laskee cipher**e%n
    print('\nDecryption: cipher^d (mod n)')
    print("%i ^ %i mod(%i) = %i" % (cipher, d , n, decrypt))
    return int_to_bytes(decrypt)
```

Figure 6. Encryption and decryption algorithms.

3.2 Broadcast Attack

For the broadcast attack (Figure 7) we need to alter the code so that it encrypts the same plaintext three times with generating own p and q prime numbers for every single encryption. I added for loop for the code, so that it runs key generation, and encryption three times, and saves ciphertexts, and their corresponding public keys "n" values in to lists.



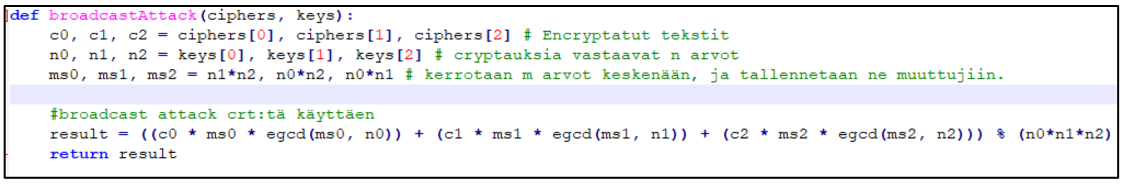
```
for k in range(3):
    d = 1
    while (d < 4 or p == q or gcd(et,e) != 1):
        p = generatePrime() # generoidaan alkuluvut p ja q.
        q = generatePrime()
        n = p*q # lasketaan n.
        et = (p-1)*(q-1) # lasketaan et = kuinka monta alkulukua 1<et<n.
        d = egcd(e,et) # lasketaan salainen avain d Eucledian algoritmilla.
    cipher = encryption(e,n,message)
    ciphers.append(cipher)
    keys.append(n)
```

Figure 7.

The broadcast attack (Figure 8) is based on a Chinese remainder theorem. It is something I didn't have time to get familiar with. but it provides the math behind the broadcast attack.

"The Chinese Remainder Theorem (CRT) is a technique to reduce modular calculations with large moduli to similar calculations for each of the factors of the modulus." (https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_2)

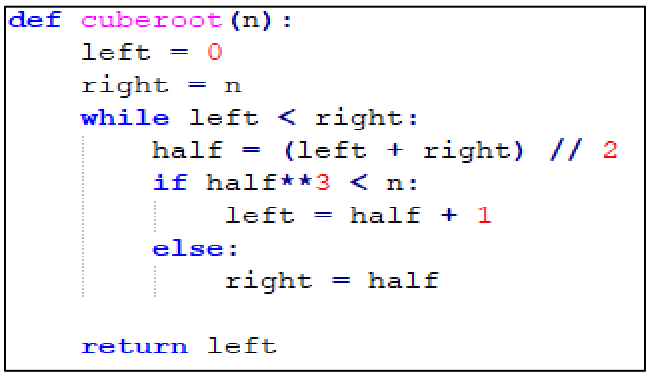
In the operation, broadcast attack multiple of n values across each other. Those numbers are saved in the "ms0"-"ms2" variables. The result is calculated by firstly multiplying each cipher value with corresponding ms0 value, and then multiplying that with Euclidean algorithm of corresponding "ms" and "n" values. Next thing is to multiply those three values together and calculate remainder when divided by public keys multiplied.



```
def broadcastAttack(ciphers, keys):
    c0, c1, c2 = ciphers[0], ciphers[1], ciphers[2] # Encryptatut tekstit
    n0, n1, n2 = keys[0], keys[1], keys[2] # cryptauksia vastaavat n arvot
    ms0, ms1, ms2 = n1*n2, n0*n2, n0*n1 # kerrotaan m arvot keskenään, ja tallennetaan ne muuttujiin.

    #broadcast attack crt:tä käyttäen
    result = ((c0 * ms0 * egcd(ms0, n0)) + (c1 * ms1 * egcd(ms1, n1)) + (c2 * ms2 * egcd(ms2, n2))) % (n0*n1*n2)
    return result
```

Figure 8. Broadcast attack

Last thing is to sort out the cubic root of the result (Figure 9). For this I created a binary search algorithm. Possibilities are from 0 to size of integer(n). The algorithm halves the list of possibilities and calculates the middle number with the power of 3. List of possible numbers to try here can be limited to half with every operation by adjusting the left and right variables. The termination condition is met when the variable left is bigger than variable left. Answer is saved to variable left.



```
def cuberoot(n):
    left = 0
    right = n
    while left < right:
        half = (left + right) // 2
        if half**3 < n:
            left = half + 1
        else:
            right = half

    return left
```

Figure 9. Find cube root function

4 Conclusion

This challenge was so much harder than I assumed when looking at the assignment info. The RSA implementation wasn't so easy after all, and it took me so long to get it work. The broadcast attack was not so hard to get, since there was pretty good info on how it's done in the assignment. Understanding the Chinese remainder theorem is something I didn't bother using my time, cause the main thing in this challenge was to get the implementations to work.

The implementations are not perfect. My RSA is not great when working with large numbers, because algorithm is quite slow when figuring if the numbers are prime or not. Also, since the prime numbers are small, the plaintext to encrypt cannot be very big. If it is, the encryption algorithm breaks.

Finally, I'm pretty satisfied of the outcome, and how the codes came together. All around the challenge was useful for me and great learn.