

Public Key Cryptography - Assignment 3

Raphael Gonon - Student number: 23040098

May 17th 2023



1 Source Code - Function from previous assignment

1.1 GCD function

```
def gcd(a,b):
    a = -a if a<0 else a
    b = -b if b<0 else b
    x = b if a < b else a
    y = b if a > b else a
    r = 1
    while r !=0:
        r = x%y
        x = y
        y = r
    return x
```

1.2 Extended GCD

```
def extended_gcd(a,b):
    x0,y0= (1,0)
    x1,y1 = (0,1)
    u = a
    v = b
    while (v != 0):
        q = u // v
        x2,y2 = (x0 -q*x1),(y0 - q*y1)
        u = v
        v = a*x2 + b*y2
        x0,y0 = x1,y1
        x1,y1 = x2,y2
    return (u,x0,y0)
```

1.3 ModExp function

```
def to_bin(n):
    s=""
    while n!=0:
        r = n%2
        s = str(r) + s
        n//=2
    return s

def mod_exp(a,e,n):
    b = to_bin(e)
    s = 1
    l = len(b)
    r=0
    for i in range(l):
        if b[i] == "1":
            r = (s*a) % n
        else:
            r = s
        s= (r**2) % n
    return r
```

2 Source Code - New function

2.1 RSA Genkey function - Miller and Rabin Primality testing

```
def miller_rabin(n):
    t = 2
    k = 0
    while (n-1)%t==0:
        t *=2
        k +=1
    m = (n-1)/(2**k)
    a = random.randint(2,(n-2))
    b = mod_exp(a,m,n)
    if b == 1 or b == -1:
        return True
    b_k = 1
    while b_k < k:
        b = mod_exp(b,2,n)
        if b == 1:
            return False
        elif b == n-1:
            #n-1 instead of -1 bc python does not return negative mod
            return True
        b_k += 1
    return False if b != n-1 else True

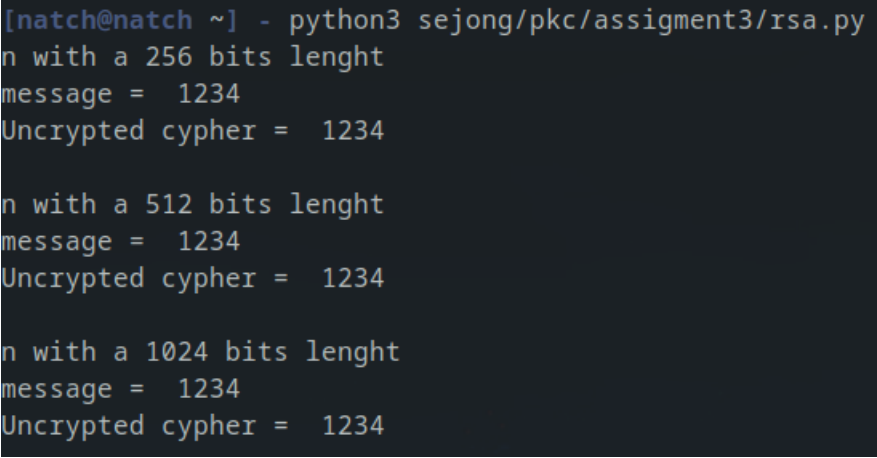
def rsa_genkey(key_length):
    a = False
    b = False
    k = key_length //2
    while not a:
        p = random.randint(2**(k-1),2**k-1)
        if p %2 ==0:
            p = p+1 if p<2**k-1 else p-1
        a = miller_rabin(p)
    while not b:
        q = random.randint(2**(k-1),2**k-1)
        if q %2==0:
            q = q+1 if q<2**k-1 else q-1
        b = miller_rabin(q)
    n = p*q
    e = 65537
    phi = (p-1)*(q-1)
    if gcd(e,phi) !=1:
        return rsa_genkey(key_length)
    d = extended_gcd(phi,e)[2]
    d = d%phi if d<0 else d
    return (n,e),(p,q,d)
```

2.2 RSA Encrypt

```
def rsa_encrypt(m,pk):  
    return mod_exp(m,pk[1],pk[0])
```

2.3 RSA Decrypt

```
def rsa_decrypt(c,sk):  
    n = sk[0]*sk[1]  
    return mod_exp(c,sk[2],n)
```



```
[natch@natch ~] - python3 sejong/pkc/assigment3/rsa.py  
n with a 256 bits lenght  
message = 1234  
Uncrypted cypher = 1234  
  
n with a 512 bits lenght  
message = 1234  
Uncrypted cypher = 1234  
  
n with a 1024 bits lenght  
message = 1234  
Uncrypted cypher = 1234
```

Figure 1: Output

3 Explanation

3.1 RSA Genkey Function

To process the key generation of RSA algorithm we first need to find two random primes of a specific bit length. We need to generate two random odd integer with a length respectively equal to the half of the length given in parameter (key_length), and then test if those number are primes number or not using the Miller and Rabin primality testing.

3.1.1 Miller and Rabin function

The Miller and Rabin primality testing function follow the given algorithm:

1. First we need to compute the factorization of $n - 1$, with $n - 1 = m * 2^k$. So we first compute the value of k, by finding the highest power of 2 which divide $n - 1$, and then we get m by dividing $n - 1$ by 2^k .
2. Then we randomly choose a, with $1 < a < n - 1$. As the randint function include bounds we run the function between 2 and $n - 2$.
3. After doing so, we compute b which is the modular exponentiation of $a^m \bmod n$ where n is the tested integer. If the result is ± 1 then n is probably a prime, otherwise we continue the algorithm by squaring b.
4. Following the Miller and Rabin test, we create a loop to process the same operation before getting a result (± 1) or doing k round of the loop. First we square the value of b, then if b is equal to 1 then n is not a prime, else if b is equal to n-1 (we used $n - 1$ instead of -1 because python cannot return a negative value for a modulo operation), then n is probably a prime, otherwise we loop.
5. If after k operation b is different from $n - 1$, then n is not a prime, otherwise it is probably a prime.

Actually to be sure to find two number that can pass the Miller and Rabin primality test, we create two loop to generate new random integer if the previous one does not pass the test. The two primes number that pass the test will be respectively p and q, which are part of the RSA private key.

Then we can compute the value of n, where $n = p * q$ which is the first element of our public key. The second one is the public exponent value e, which is conventionally fixed to 65537. We can now form the RSA public key with the value of n and e. The last step is to compute d, the decryption exponent which is a part of the private key.

To do so, we need to compute the value of $\phi(n)$, where $\phi(n) = (p - 1)(q - 1)$ and verify that e and $\phi(n)$ are relatively prime. In my code I made the choice of running a another time the complete algorithm with the same key length with a recursive call, to handle the case where they are not relatively primes.

Once we have the value of $\phi(n)$ we can compute the value of d (modulo inverse of e), by running the extended GCD algorithm. As we need to have a positive value for d we compute a last modulo $\phi(n)$ operation to get a positive value.

Now that we have all the value of n, e, p, q, d we return a tuple for the public key (n, e) and a 3-uplet for the private key (p, q, d).

3.2 RSA Encryption function

For the RSA encryption function we just need to apply the following equation by using the public key : $c \cong m^e \text{ mod } n$

3.3 RSA Decryption function

For the RSA decryption function we just need to apply the following equation by using the private key : $m \cong c^d \text{ mod } n$

3.4 Correctness of RSA Scheme

To better understand the encryption and the decryption scheme we can show the correctness of RSA scheme, based on Euler theorem :

$$\begin{aligned} c^d &\cong (m^e)^d \text{ mod } n \\ &\cong m^{k*\phi(n)+1} \text{ mod } n \\ &\cong m^{k*\phi(n)} * m^1 \text{ mod } n \\ &\cong m^1 \text{ mod } n \\ &\cong m \text{ mod } n \end{aligned}$$

The correctness of that scheme is based on the fact that d (private key exponent), is the modulo inverse of e (public key exponent), so we get the following equation :

$$e * d \cong 1 \text{ mod } \phi(n)$$

Once we get the equation above, we can apply Euler theorem to recover the original unencrypted message. Thus we have $m^{k*\phi(n)} = 1 \text{ mod } n$ in the equation $m^{k*\phi(n)} * m^1 = 1 * m \text{ mod } n$, so we finally recover m.