

# Public Key Cryptography - Assignment 4

Raphaël Gonon - Student number: 23040098

May 30th 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
```

## 1.4 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
```

## 2 Source Code - New function

### 2.1 RSA-PKS Genkey function

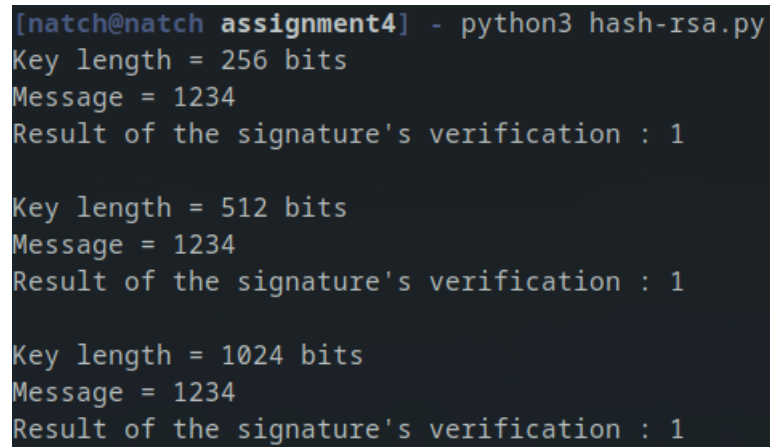
```
def rsa_pks_genkey(key_length):
    a = False
    b = False
    k = key_length //2
    while not a:
        p = random.randint(2**(k-1),2**k-1)
        if p %2 ==0:
            p = p+1 if p<2**k-1 else p-1
        a = miller_rabin(p)
    while not b:
        q = random.randint(2**(k-1),2**k-1)
        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
    H = hashlib.sha1()
    return (p,q,d),(n,e,H)
```

## 2.2 RSA-PKS Sign

```
def rsa_pks_sign(m,sk,pk):  
    hash_fun = pk[2].copy()  
    hash_fun.update((str(m)).encode())  
    h = hash_fun.digest()  
    h= int.from_bytes(h,"big")  
    return mod_exp(h,sk[2],pk[0])
```

## 2.3 RSA-PKS Verify

```
def rsa_pks_verify(sig,m,pk):  
    s = mod_exp(sig,pk[1],pk[0])  
    hash_fun = pk[2].copy()  
    hash_fun.update((str(m)).encode())  
    h = hash_fun.digest()  
    h = int.from_bytes(h,"big")  
    return 1 if s ==h else 0
```



```
[natch@natch assignment4] - python3 hash-rsa.py  
Key length = 256 bits  
Message = 1234  
Result of the signature's verification : 1  
  
Key length = 512 bits  
Message = 1234  
Result of the signature's verification : 1  
  
Key length = 1024 bits  
Message = 1234  
Result of the signature's verification : 1
```

Figure 1: Output

## 3 Explanation

### 3.1 RSA-PKS Genkey Function

The key generation function follow the same process as the textbook RSA and add a hash function to the public key. Here are the step for the key generation process for textbook RSA:

1. First of all generate two random primes numbers (or strong pseudo primes that can pass the Miller and Rabin test), and compute the integer  $n$  that will be a part of the public key.
2. Once we find the value of  $n$  we use the commonly used value of the exponent  $e = 65537$ . Before going to the next step we need to ensure that  $\gcd(\phi(n), e) = 1$ , otherwise we have to restart the process from the begging. Actually we could try to find a another value for  $e$ , but using another value could reveal some unknown threat, so it is safer to generate new primes.
3. The following step is to compute  $d$  the modulo inverse of  $e$ , such as  $ed \cong 1 \mod \phi(n)$  by running the extended GCD algorithm with  $e$  and  $\phi(n)$  ( $d$  is the exponent used for decryption stored in the private key). To be sure of having a positive value for  $d$ , we need to compute a modulo  $\phi(n)$  operation on  $d$  to avoid negative value since the result of the extended GCD can be negative.

After doing those step, we now add the hash function to the public key. To do so we create a variable that will stores the hash function object, provides by the hashlib library. Finally we return the private key with the value of  $p, q$  and  $d$ , and the public key with the value of  $n, e$  and  $H$  the object containing the hash function.

### 3.2 RSA-PKS Sign function

For the RSA-PKS Sign function we first need to generate a hash of the message and then we have to compute the modulo exponentiation such as  $\sigma \cong h^d \mod n$ .

In the source code we firstly create a copy of the object that contains the hash function, to being able to obtain the same hash with the same message in input of the hash function. Actually the object provided by the hashlib library is updated every time we add some data in the object, and by doing a copy of the initial object we can assure that the same message will always return the same output.

After creating the copy of the hash function's object, we update the hash function with the bytes representation of our messages by converting it as a string and then in bytes. Once it is done we can generate the hash of our messages using the digest method and store it in the variable  $h$ .

Finally we compute and return the signature following the previous equation using the modulo exponentiation thanks the private key to sign our message.

### 3.3 RSA-PKS Verify function

To proceed the RSA-PKS verify function, we first use the public key to recover the value of the hash message received, with  $h \cong \sigma^e \mod n$ . This equation is verify as  $d$  is the modulo inverse of  $e \mod n$ .

Then we create a variable to copy the hash function object, and we computes the hash of the message given in parameter using it bytes representation.

Once it is done we compare the two obtained hash message and if they match we have the correct message so we return 1 (for true) otherwise we return 0 (for false).