Public Key Cryptography - Assignment 5

Raphaël Gonon - Student number: 23040098

June 13th 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 Generate prime subgroup

```
def generate_prime_subgroup(plen, qlen = 160):
    a1 = False
    b = False
    while not a1:
        q = random.randint(2**(qlen-1),2**qlen-1)
        if q %2==0:
            q = q+1 if q<2**qlen-1 else q-1
        a1 = miller_rabin(q)
    while not b:
        k = random.randint(2**(plen-qlen-1),(2**(plen-qlen)-1))
        p = k*q+1
        if p % 2 == 0:
            p = p + 1 if p < 2 ** (plen-qlen)- 1 else p-1
        b = miller_rabin(p)
    a = random.randint(2,p-1)
    g = mod_exp(a,k,p)
    return (p,q,g)
```

2.2 Schnorr setup

```
def schnorr_setup(plen):
    pp = generate_prime_subgroup(plen)
    return pp
```

2.3 Schnorr genkey

```
def schnorr_genkey(pp):
    a = random.randint(1,pp[1]-1)
    A = mod_exp(pp[2],a,pp[0])
    H = hashlib.sha1()
    sk = a
    pk = (pp[0],pp[1],pp[2],A,H)
    return sk,pk
```

2.4 Schnorr Hash

```
def schnorr_hash(R, msg, q):
    s = 0
    for c in msg:
        s+=ord(c)
    hash_fun = hashlib.sha1()
    res = (R | s)
    hash_fun.update(str(res).encode())
    h = hash_fun.digest()
    h = int.from_bytes(h,"big") %q
    return h
```

2.5 Schnorr Sign

```
def schnorr_sign(msg, sk, pp):
    r = random.randint(0,pk[1]-1)
    R = mod_exp(pk[2],r,pk[0])
    h = schnorr_hash(R,msg,pp[1])
    s = (r + sk * h) % pp[1]
    return (R,s)
```

2.6 Schnorr Verify

```
def schnorr_verify(sig, msg, pk, pp):
    h = schnorr_hash(sig[0],msg,pp[1])
    left = mod_exp(pk[2],sig[1],pk[0])
    right = (sig[0]*mod_exp(pk[3],h,pk[0]))%pk[0]
    return 1 if left == right else 0
```

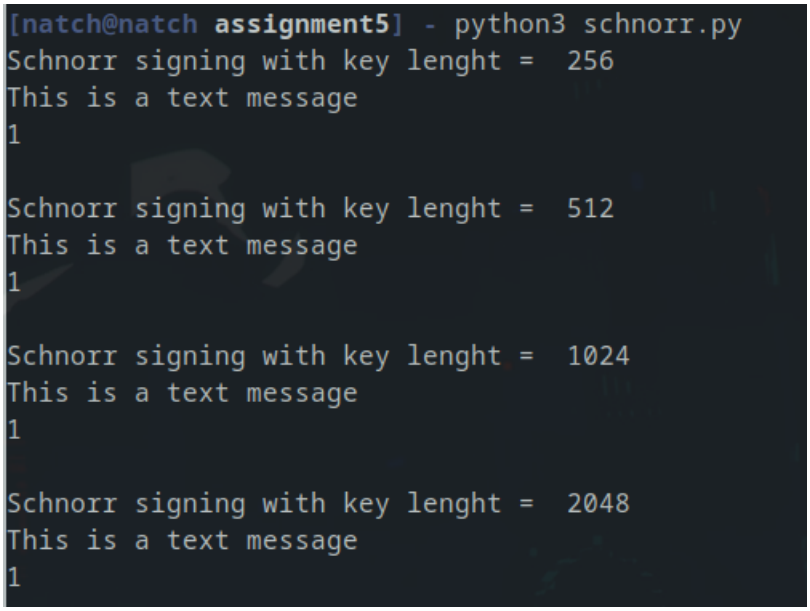

Figure 1: Output

3 Explaination

3.1 Generate Prime Subgroup

Following the given algorithm, we first need to find a prime q of the lenght given in parameter of the function. To do so we generate a random number using a power of two to have the bits length required. Once this number is generated we can check wether it is a prime number or not using the miller and rabin primality testing.

After finding q, we can now compute the value of p which is $p = k * q + 1$ with k a random integer. The following step is to generate k, knowing that we will multiply this value to q and that we want p to respect a specific bit lenght. This is why we generate k with the power of $plen - qlen$ because of the exponent addition while we multiply two number ($2^a * 2^b = 2^{a+b}$). Now that k is generate we compute p and test if it is a prime number or not using the miller and rabin test. If not we need to generate a new value k until we find a prime p.

Moreover, with the values of p and q, we now have to compute the value of the generator g modulo p, according to the equation $g = a^k \ mod \ p$ with a, a random integer value such as $1 < a < p$. Finally we can compute g and return a 3-uplet (p, q, g).

3.2 Schnorr setup

The schnorr setup fonction take a prime bit lenght for p as argument (it will be always 160 for q), and return the a 3-uplet pp, containing the value p,q and g. Actually as we define the previous function with qlen as a default parameter with a value of 160, we do not need to specify the len of q in that call of the function.

3.3 Schnorr genkey

The schnorr generation key needs few things to be performed. First of all we need to generate the private key exponent a, which the key of the DLP and we generate it as $1 < a < q - 1$. Then we are able to compute the value of A, $A = g^a$ which is a part of the public key needed to perform the verification process.

Now we have to create a object which contains the hash function using hashlib. Finally we return all the values as sk (a) and pk (p, q, g, A, H) using for p, q and g those in parameters.

3.4 Schnorr hash

To perform the schnorr hashing function we first need to convert the string message as an integer. To do so, I sum up the ascii value of each char of the string message. Then we have to hash the result of the or operation between R and the int value of the string (here s). I choose to compute this value in a variable before hashing to have a code more understandable.

To generate the hash we use the function of the hashlib that take parameter in bytes, so we first need to convert the result of the or operation in bytes. Once the hash done, we have to convert the bytes value to int and perform the required modulo q operation before returning the final hash value.

3.5 Schnor sign

In order to processed the schnorr signature, we first have to generate a random value r between 0 and q - 1. With this random value r we can compute R which is the first component of the signature, with $R = g^r$. Then we hash the message with the computed value of R and then q. The final step is to compute value of s, which is used as a exponent value for the verification part. We have the following equation for s, $s = r + a * h \ mod \ q$. Then we can return the forged signature, composed of (R, s).

3.6 Schnorr Verify

For the schnorr verify function we have to verify the following equation : $g^s \cong RA^h \ mod \ p$. To do so, we need to compute the hash value h, using the given message and R the first part of the signature. Once it is done we can compute in one variable the value of g^s and in a other variable the value of $RA^h \ mod \ p$. The last step is to return 1 if the two variable are equals, 0 otherwise. By performing the schnorr verification we can verify the authenticity of the given message.