ECE 458

Assignment 2

JULY 5, 2021

Question 1

For question 1, our implementation goes through every secret key and returns the public key.

```
def question1(g, p, sks):
    for sk in sks:
        #goes through all secret keys and generates public key
        pki = generate_public_key(g, sk, p)
        pk.append(pki)
        #print(sk, pki)
    #print(pk)
```

In our generate_public_key method we use the pow function which allows us to replicate g^sk mod p in python. We used this function as it is computationally less intense than actually taking powers and mods in python.

```
def generate_public_key(g, sk, p):
    #print("GENRATING PUBLIC KEY")
    #pow does gives us g ^sk %p
    output = pow(g, sk, p)
    return output
```

We got the following results:

pairs of (ski, pki) for i = 1, 2, 3:

Sk1: 432398415306986194693973996870836079581453988813,

pk1:
49336018324808093534733548840411752485726058527829630668967480568854756416567496216
29491905191014868618662270686970232166446509470324736864650682101529030248099045013
02806169292269172462551470632923017242976806834012586361821855991241311700775484507
54294083728885075516985144944984920010138492897272069257160

Sk2: 165849943586922055423650237226339279137759546603,

pk2:
15805870909801658055470664266537756807095291079892225065940408602107787041560306349
91442689557657923705109503128714241617779465259782688524304338594046010559983239609

119885068493690814059903733444769313742704042835877179267984159785563583549068863675731166833742495836608835505849987814561035731216254434283852

Sk3: 627658512551971075308886219669315148725310346887,

Pk3:
10748734721713476921614873611718904745721634734864852672984965178578400403592975468637711919072521139171475696788980946180070433308221960020597389241184295873409184038758067499585819584316876921039282554039249439646055641729080454908899640888805475449690115972381324471288563867011425441594691721425168305284

Question 2

In question 2, we are converting the input into an int, and then to a hex value. Next, the leading zeros are stripped out and turned to bytes. The hashobj is created by using the *sha3_224()* method from the *hashlib* library. Finally, the hashed value is obtained by using the hashobj and returned in binary.

SHA3-224 for number 10:

10011110100001101111111110110100101010101011111001010100101011111010000000101111100001000000010010011010010110100001011011001110001110001110101000000110011011001100001001111011101001010000101111010010000010101101101010100010 11

SHA3-224 for number 20:

10111111100001100111011100100010011000111010101101010100001111010100100101111001001111011010101101001001100001111110111110011000010011001111101101111001101111001010001111111011010010110011000000000001100001101011010001011110 1

SHA3-224 for number 30:

11101011111101101000100000100001110101101010000100010110010101111010101001100000000101111010001000011100001111010010000010111011110000100110100001110100001011101101100101011000000001111001110000010100011001111011000001111101

Question 3

For question 3 we generate the m1 and m2 using the needed primary keys for each message and the amnt value. We pass these values in from our array into the question3 function.

```
elif selected =='3':
    #gets messages for both m1 and m2. Question 1 populates primary key array
    question1(g,p,sks)
    print("M1")
    question3(pk[0], pk[1], 2)
```

```
    print("M2")
    question3(pk[1], pk[2], 3)
```

The question3 function calls the generate message function.

```
def question3(pk1, pk2, amnt):
    #calls the generate message function to generate the concatenated message
    print(generate_message(pk1,pk2,amnt))
```

The generate _message function gets the binary representation of all inputs, concats together and then returns an integer.

```
def generate_message(pk1, pk2, amnt):
    #take the primary keys, convert to binary keeping significant digits of 399, and then add the amnt to it. We then 
convert back into an integer
    pk1= bin(pk1)[2:401]
    pk2 = bin(pk2)[2:401]
    amnt = bin(amnt)[2:] if amnt != 1 else "01" #special case for 1.

    m1 = int(pk1 + pk2 + amnt, 2)
    return m1
```

Thus we get our messages. Below is our outputs:

M1:

3659949252416110243842695635193186175332314062381277840275697280706181584387548098013985066122238476554562154647781507975854150384723431517763123752624375814447084742521086657783968078531722961438847114809276456296127939875396313500323209774

M2:

5862723359560769902306160798198708078190251863174731720976853028828587598690079473740178204613802974429824697161497463490301926377746107083836495644583074647011520782397841059567377684346054784824548991914382152504386907016111662295782911303

Question 4

For question 4 we must both generate a digital signature and then verify it.

We get the signature pair using the method given for digital signatures. So we first find a random k in between 0 and p. We then take the message and hash it. From there we determine r and s using the respective inputs needed to determine their value.

```python
def generate_signature(message, p, q, g, sk):
    #get random k in between 0 and p
    k = randint(0, p-1)


    #convert message into hex and then bytes
    integer = hex(message)
    integer = integer[2:]
    integer = bytes(integer, encoding='utf-8')

    #hash the message
    hashobj.update(integer)
    hashed = str(hashobj.hexdigest())
    #convert from hex string to int
    hashed_num = int(hashed, 16) #this is our h(m)




    y = pow(g, sk, p) # public key
    #print('public key ', y)

    r = pow(g, k, p) #first integer of signature paur


    # hashed_num = (sk1 * r + k * s) % q
    s = ((hashed_num - sk * r)  * mod_inverse(k, q)) % q #second half of signature pair

    #print((hashed_num - sk * r)  * mod_inverse(k, q))
    print('r ', r)
    print('s ', s)
    #send signature pair and publick key abck
    return hashed_num, r, s, y
```

This is how we can get Sig1 and Sig2.

We verify by calling the verify_signature method. We first check to make sure that the parameters fall in our expected ranges. If it is we go onto verifying it by back calculating an r value(w) and comparing it to the r value we found in our initial signature generation. That way we know it is working. We do this by using the method discussed in the lecture notes.

```
def verify_signature(hashed_message, r, s, g, p, q, pk):
    #check if values fall in expected rangers
    if r < 0 or r > p or s < 0 or s > p:
        print ("verification rejected ")
        return False



    s1 = mod_inverse(s, q)
    u = hashed_message * mod_inverse(s, q) % q    # u = hashed_num * s inverse mod q,
    v = (-r * s1) % q    # and v = -r * s inverse mod q

    arg1 = pow(g, u, p) #g ^ u mod p
    arg2 = pow(pk, v, p) #pk ^ v mod p

    w = arg1 * arg2
    w = pow(w, 1, p) # g^u y^v mod p
    #this value is our generated r
    print("w", w)

    #if the r is the same we can verify that we generated a valid signature
    if r == w:
        print("verification succesful ")
        return True
```

This is our output for the question.

Output

```
IndexError: list index out of range
DaivikMac:Assignment 2 daivikgoel$ python3 assignment2.py
WHICH QUESTION DO YOU WANT: 4
r  810589125034580997099852498362216118417379205273175690438538889284414086982589954
986254662771404106957874315758119966017050449656555561538146520728206922553162171656
097046443968765626237342479373744818062848943408402330146411598857450760811471223162
96371162777820868479779412405529957693476205858345707057298
s  82312537251777474857406676638257961685160321020
w 8105891250345809970998524983622161184173792052731756904385388892844140869825899549
862546627714041069578743157581199660170504496565555615381465207282069225531621716560
970464439687656262373424793737448180628489434084023301464115988574507608114712231629
6371162777820868479779412405529957693476205858345707057298
verification succesful
r  115981462476884199700876396428684859792112075633525002670666855814338887454096389
906776852598820252800120147217242565372530136539194514517329317404599330240464474409
939216097532918199328733951954168650826976898839799716232874206186807558435369756590
845271784511671833379279608277153701556839005025112976455530
s  668174398700767433498639110966695275529316861058
```

Question 5

For this question we want to find a nonce value that gives us 24 leading zeroes. First we need to create a message and hashed message of the previous for our Ti. Hence first we go through every message and generate a hashed equivalent (Note: Since T0 would use the hash of amnt0 our hashed messages array has that value first. This creates an offset of 1.

```
def find_nounce():
    found = 0

    NONCE = 0

    question1(g,p,sks) #generate the public keys

    messages = [] #save m1, m2, m3 etc
    hashed_messages = [question2(amnt[0])] #Offset by 1 as h(m0) is h(amnt0)

    for i in range(len(pk)-1):
        #create a message using primary keys
        messages.append(generate_message(pk[i], pk[i+1], amnt[i+1]))
        #create hashed message using these messages
        hashed_messages.append(question2(generate_message(pk[i], pk[i+1], amnt[i+1])))
    nonces = []
```

From there we start a clock to determine the time. Our function will get both nonce 1 and 2 and keep going till we find a value with 24 leading zeroes. We add a nonce value, our mi and h(m(i-1)) and hash it.

If this value has the 24 leading zeroes we stop else we keep iterating the nonce value by 1. We figure out if it has 24 leading zeroes if the hash values's length is 200. This is because python will not show leading zeroes so we know should the length be 200, we had 24 leading zeroes.

```
start_time = datetime.datetime.now()

for i in range(2):
    #find nonce1 and nonce2
    found = 0
    NONCE = 0
    print(i)
    #for y in [2,4,6,8,10,12,14,16,18,20,22]: #this is for testing purposes on effort required
        #found = 0
    while found == 0:
        #because the array is offset by 1 in hashedmessages it represents the equation NONCE + mi + h(m i-1)
        z = NONCE + messages[i] + int(hashed_messages[i])
        #find hash of the concatenation.

        new_hash = question2(z) #new_hash is Ti
        # since the first 24 bits aren't shown in binary representation, when the length of the new hash is exactly 24
bits short then we know we had 24 leading zeroes
        if 224 - len(new_hash) == 24:
            #if we have 24 leading zeroes we have found a nonce
            found = 1
            end_time = datetime.datetime.now()

            print("elapsed time for finding nonce: ", (end_time - start_time))
            nonces.append(NONCE)
        NONCE +=1

#show both nonces
print(nonces)
```

With this method we got the following nonce values:

```
DaivikMac:Assignment 2 daivikgoel$ python3 assignment2.py
WHICH QUESTION DO YOU WANT: 5
0
elapsed time for finding nonce:  0:04:03.778100
1
elapsed time for finding nonce:  0:09:34.984521
[46488426, 63390458]
```

Question 7

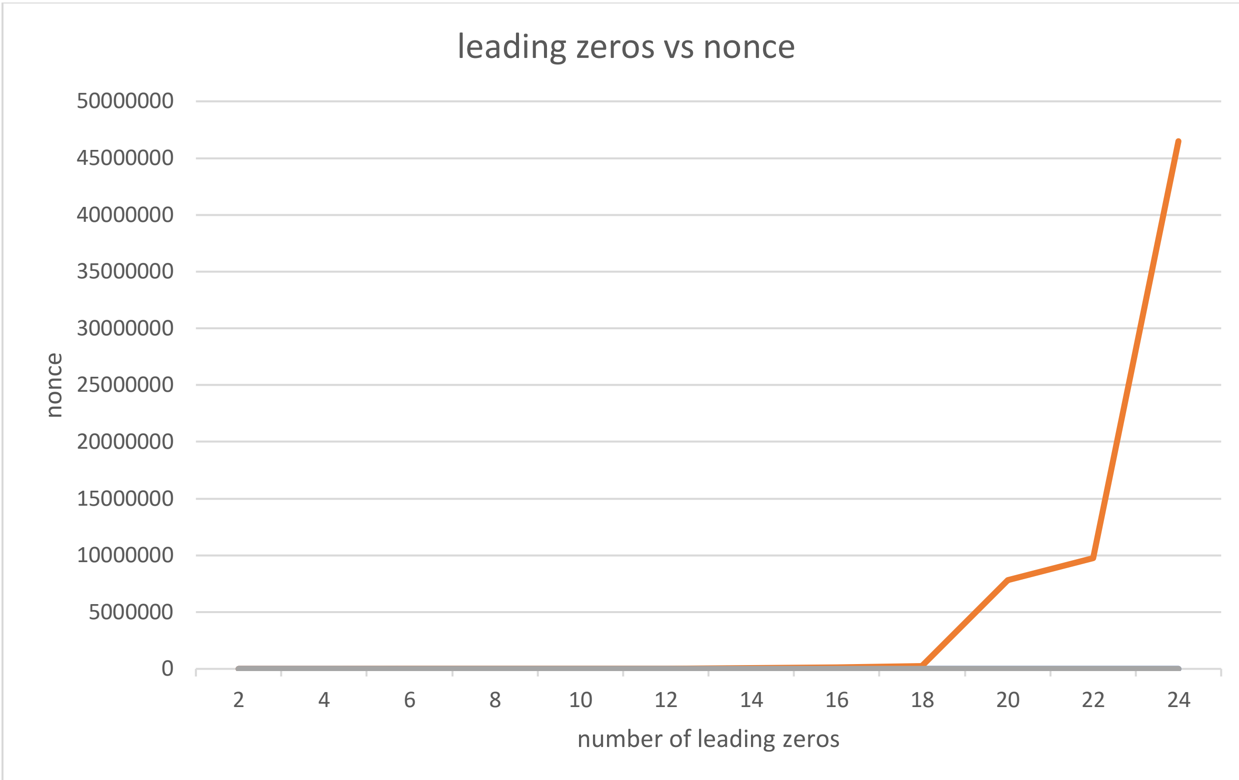
Number of exhaustive searches required for varying nonce in order to get hash value that satisfies k=24 leading zeros.

Since the hash function is collision resistant, we have no way of knowing in advance what the hashed output would be prior to trying out all the input values. Thus, we would need to brute force at most 2 ^ k times to ensure that we land on a hashed value with k leading zeros.

For k = 24, the number of iterations taken to find the nonce 1 was 46488426 while the number of iterations taken to find nonce 2 was 63390458. Their respective times taken being 0:04:03 and 0:09:34.

effort needed if we wanted to find each nonce with 32 leading zeros:

The effort to find each nonce with 32 leading zeros would be 2 ^32. (2^32 / 2^24 = 2^8= 256). ThusThus, it takes roughly 256 times more effort to generate nonce with 32 leading zeros compared to generating the nonce with 24 leading zeros. To further prove this, the figure below was made by plotting the values we received for increasing values of k, starting from k = 2 we can see that for increasing values of k, nonce increases exponentially.


leading zeros vs nonce
50000000
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
nonce
2
4
6
8
10
12
14
16
18
20
22
24
number of leading zeros