**Student name: Yskak Toltay**

**Student ID: 37791214**

**Task 1 – Affine cipher encryptor**

First, I build a table of the letter mapping in my code. I define a function called buildTable, that takes two arguments a and b that are used in the affine cipher formula. For every letter in the ascii\_uppercase alphabet I apply the formula (a \* i + b) mod 26 (sizeOfAlphabet ), where i is a letter, so each letter gets mapped to another:

def buildTable(a, b):

    cipherTable = {}

    sizeOfAlphabet = len(string.ascii\_uppercase)

    for i in range(sizeOfAlphabet):

        plainChar = string.ascii\_uppercase[i]

        cipherChar= string.ascii\_uppercase[(a\*i+b)%sizeOfAlphabet] # apply the affine cipher formula to map each letter in the alphabet

        cipherTable[plainChar] = cipherChar

    return cipherTable

I then define a function called encryptAffine, that encrypts every letter in the string to it’s mapped value if it’s in the table from buildTable function. If it’s not, the letter remains the same:

def encryptAffine(plainText, a, b):

    cipherTable = buildTable(a, b)

    cipherText = ""

    for char in plainText:

        if char in cipherTable:

            cipherText += cipherTable[char] # switch characters that are in the table to its' mapped values

        else:

            cipherText += char  # leave characters that are not in the table unchanged

    return cipherText

**Task 2 – Avalanche calculator**

In my avalancheCalculator function, I first apply sha256 hashing function to both strings and get their hashed values in hex using hashlib library. Then I convert both hashed values to binary using bin(), because it is easier to compare single bits in binary. Because bin() removes zeros at the start since they don’t add any value, I also added a for loop for each binary value to append zeros back at the start, as the hashed value of sha256 function has to be 256 bits long:

def avalancheCalculator(string1, string2):

    string1Hex = hashlib.sha256(string1.encode()).hexdigest()

    string2Hex = hashlib.sha256(string2.encode()).hexdigest() # apply sha256 hashing to both strings

    string1Bin = bin(int(string1Hex, 16))[2:] # convert to binary to find the number of bits that are different between hashed values

    string2Bin = bin(int(string2Hex, 16))[2:] # remove prefix 0b from the strings with [2:0] (start strings from second char)

    # add zeros to the start of strings' binary values so they are 256 bits long. They should be 256 bits long because we are using sha256 hash function which outputs 256 bit values. bin() function removes zeros from the start as they don't add any value.

    if len(string1Bin) < 256:

        while len(string1Bin) < 256:

            string1Bin = "0" + string1Bin

    if len(string2Bin) < 256:

            while len(string2Bin) < 256:

                string2Bin = "0" + string2Bin

Finally, I add a for loop that compares each bit in the binary values and increments avalanche number if they are different.

    avalanceNumber = 0

    for i in range(256):

        if string1Bin[i] != string2Bin[i]: # if bits are different, increment avalanche number

            avalanceNumber += 1

    return avalanceNumber

**Task 3 – Diffie Helman**

In my Diffie\_Hellman function, I first load the dh parameters and peer public key provided to python types that cryptography module understands:

dhParameters = serialization.load\_pem\_parameters(\*provided\_dh\_parameters\*)

peerPublicKey = serialization.load\_pem\_public\_key(\*provided\_peer\_public\_key\*)

I then generate my private key X\_A and my public key Y\_A using dh parameters, so X\_A < q and Y\_A = α^X\_A mod q:

myPrivateKey = dhParameters.generate\_private\_key()

myPublicKey = myPrivateKey.public\_key()

Then, I generate a shared key (secret) using exchange() with my private key and peer public key and find derived key using the HMAC-based Extract-and-Expand key derivation function with the following setup: SHA256 for the algorithm, length = 32, salt=None and info=b'handshake data':

sharedKey = myPrivateKey.exchange(peerPublicKey)

    derivedKey = HKDF(

        algorithm=hashes.SHA256(),

        length=32,

        salt=None,

        info=b'handshake data',

    ).derive(sharedKey) # derive key using peer public key and my private key (Y\_A)^(X\_B) mod q

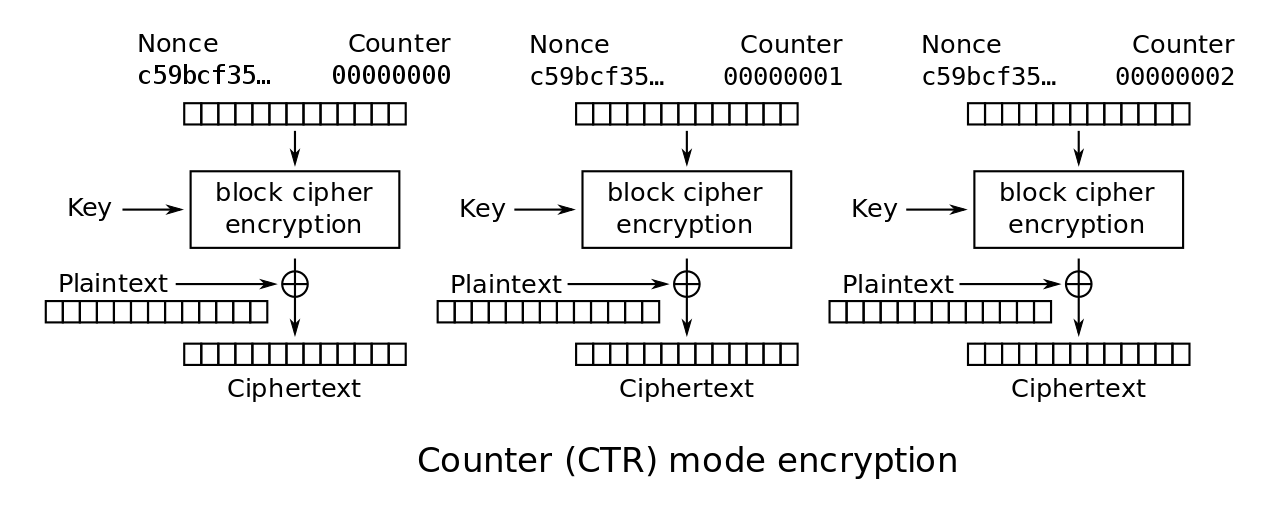
Finally, I encode my public key to PEM format and return it with the derived key.

myPublicKeyPem = myPublicKey.public\_bytes(serialization.Encoding.PEM, format=serialization.PublicFormat.SubjectPublicKeyInfo) # encode my public key to PEM format

return (myPublicKeyPem, derivedKey)

**Task 4 – Finding the ciphertext**

In the AES-CTR the IV(nonce) is combined or concatenated with the counter. As can be seen on the picture below, this value is then combined with the key and that whole value is then XOR’d with the plaintext to get the cipher text:



Therefore, since it is known that the same key and iv values are used to encrypt all subsequent messages, by simply XORing the plaintext with the ciphertext, the combined value of key, IV and counter can be retrieved. After that, this value can be XOR’d with the next message to get its’ ciphertext, or even XOR’d with the ciphertext to get the plaintext of the message.

In my code, I define a function xorMessages that uses zip() to get tuples of each byte in the byte strings and then XORs those bytes to get the combined value of key, IV and counter.

def xorMessages(message1, message2):

    # xor every byte of message 1 with message 2

    # the byte strings don't have to be the same size because zip function zips to whatever byte string is the shortest

    return bytes([a ^ b for a, b in zip(message1, message2)])

def findCiphertext():

    keyAndIvValue = xorMessages(messageA, cipherTextA) # xor message a plaintext and ciphertext to get the added value of key, iv and counter

Then I use this function to also XOR this combined key value with the plaintext of message B to get the ciphertext of message B.

cipherTextB = xorMessages(messageB, keyAndIvValue) # xor the found value with message b plaintext to find it's ciphertext

    return cipherTextB #ciphertext of messageB in bytes