**Assignment 1:**

from sys import exit

from time import time

KeyLength = 10

SubKeyLength = 8

DataLength = 8

FLength = 4

# Tables for initial and final permutations (b1, b2, b3, ... b8)

IPtable = (2, 6, 3, 1, 4, 8, 5, 7)

FPtable = (4, 1, 3, 5, 7, 2, 8, 6)

# Tables for subkey generation (k1, k2, k3, ... k10)

P10table = (3, 5, 2, 7, 4, 10, 1, 9, 8, 6)

P8table = (6, 3, 7, 4, 8, 5, 10, 9)

# Tables for the fk function

EPtable = (4, 1, 2, 3, 2, 3, 4, 1)

S0table = (1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)

S1table = (0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)

P4table = (2, 4, 3, 1)

def perm(inputByte, permTable):

"""Permute input byte according to permutation table"""

outputByte = 0

for index, elem in enumerate(permTable):

if index >= elem:

outputByte |= (inputByte & (128 >> (elem - 1))) >> (index - (elem - 1))

else:

outputByte |= (inputByte & (128 >> (elem - 1))) << ((elem - 1) - index)

return outputByte

def ip(inputByte):

"""Perform the initial permutation on data"""

return perm(inputByte, IPtable)

def fp(inputByte):

"""Perform the final permutation on data"""

return perm(inputByte, FPtable)

def swapNibbles(inputByte):

"""Swap the two nibbles of data"""

return (inputByte << 4 | inputByte >> 4) & 0xff

def keyGen(key):

"""Generate the two required subkeys"""

def leftShift(keyBitList):

"""Perform a circular left shift on the first and second five bits"""

shiftedKey = [None] \* KeyLength

shiftedKey[0:9] = keyBitList[1:10]

shiftedKey[4] = keyBitList[0]

shiftedKey[9] = keyBitList[5]

return shiftedKey

# Converts input key (integer) into a list of binary digits

keyList = [(key & 1 << i) >> i for i in reversed(range(KeyLength))]

permKeyList = [None] \* KeyLength

for index, elem in enumerate(P10table):

permKeyList[index] = keyList[elem - 1]

shiftedOnceKey = leftShift(permKeyList)

shiftedTwiceKey = leftShift(leftShift(shiftedOnceKey))

subKey1 = subKey2 = 0

for index, elem in enumerate(P8table):

subKey1 += (128 >> index) \* shiftedOnceKey[elem - 1]

subKey2 += (128 >> index) \* shiftedTwiceKey[elem - 1]

return (subKey1, subKey2)

def fk(subKey, inputData):

"""Apply Feistel function on data with given subkey"""

def F(sKey, rightNibble):

aux = sKey ^ perm(swapNibbles(rightNibble), EPtable)

index1 = ((aux & 0x80) >> 4) + ((aux & 0x40) >> 5) + \

((aux & 0x20) >> 5) + ((aux & 0x10) >> 2)

index2 = ((aux & 0x08) >> 0) + ((aux & 0x04) >> 1) + \

((aux & 0x02) >> 1) + ((aux & 0x01) << 2)

sboxOutputs = swapNibbles((S0table[index1] << 2) + S1table[index2])

return perm(sboxOutputs, P4table)

leftNibble, rightNibble = inputData & 0xf0, inputData & 0x0f

return (leftNibble ^ F(subKey, rightNibble)) | rightNibble

def encrypt(key, plaintext):

"""Encrypt plaintext with given key"""

data = fk(keyGen(key)[0], ip(plaintext))

return fp(fk(keyGen(key)[1], swapNibbles(data)))

def decrypt(key, ciphertext):

"""Decrypt ciphertext with given key"""

data = fk(keyGen(key)[1], ip(ciphertext))

return fp(fk(keyGen(key)[0], swapNibbles(data)))

if \_\_name\_\_ == '\_\_main\_\_':

try:

assert encrypt(0b0000000000, 0b10101010) == 0b00010001

except AssertionError:

print("Error on encrypt:")

print("Output: ", encrypt(0b0000000000, 0b10101010), "Expected: ", 0b00010001)

exit(1)

try:

assert encrypt(0b1110001110, 0b10101010) == 0b11001010

except AssertionError:

print("Error on encrypt:")

print("Output: ", encrypt(0b1110001110, 0b10101010), "Expected: ", 0b11001010)

exit(1)

try:

assert encrypt(0b1110001110, 0b01010101) == 0b01110000

except AssertionError:

print("Error on encrypt:")

print("Output: ", encrypt(0b1110001110, 0b01010101), "Expected: ", 0b01110000)

exit(1)

try:

assert encrypt(0b1111111111, 0b10101010) == 0b00000100

except AssertionError:

print("Error on encrypt:")

print("Output: ", encrypt(0b1111111111, 0b10101010), "Expected: ", 0b00000100)

exit(1)

t1 = time()

for i in range(1000):

encrypt(0b1110001110, 0b10101010)

t2 = time()

print("Elapsed time for 1,000 encryptions: {:0.3f}s".format(t2 - t1))

**Assignment 2**

# Description: Simplified AES implementation in Python 3

import sys

# S-Box

sBox = [0x9, 0x4, 0xa, 0xb, 0xd, 0x1, 0x8, 0x5,

0x6, 0x2, 0x0, 0x3, 0xc, 0xe, 0xf, 0x7]

# Inverse S-Box

sBoxI = [0xa, 0x5, 0x9, 0xb, 0x1, 0x7, 0x8, 0xf,

0x6, 0x0, 0x2, 0x3, 0xc, 0x4, 0xd, 0xe]

# Round keys: K0 = w0 + w1; K1 = w2 + w3; K2 = w4 + w5

w = [None] \* 6

def mult(p1, p2):

"""Multiply two polynomials in GF(2^4)/x^4 + x + 1"""

p = 0

while p2:

if p2 & 0b1:

p ^= p1

p1 <<= 1

if p1 & 0b10000:

p1 ^= 0b11

p2 >>= 1

return p & 0b1111

def intToVec(n):

"""Convert a 2-byte integer into a 4-element vector"""

return [n >> 12, (n >> 4) & 0xf, (n >> 8) & 0xf, n & 0xf]

def vecToInt(m):

"""Convert a 4-element vector into 2-byte integer"""

return (m[0] << 12) + (m[2] << 8) + (m[1] << 4) + m[3]

def addKey(s1, s2):

"""Add two keys in GF(2^4)"""

return [i ^ j for i, j in zip(s1, s2)]

def sub4NibList(sbox, s):

"""Nibble substitution function"""

return [sbox[e] for e in s]

def shiftRow(s):

"""ShiftRow function"""

return [s[0], s[1], s[3], s[2]]

def keyExp(key):

"""Generate the three round keys"""

def sub2Nib(b):

"""Swap each nibble and substitute it using sBox"""

return sBox[b >> 4] + (sBox[b & 0x0f] << 4)

Rcon1, Rcon2 = 0b10000000, 0b00110000

w[0] = (key & 0xff00) >> 8

w[1] = key & 0x00ff

w[2] = w[0] ^ Rcon1 ^ sub2Nib(w[1])

w[3] = w[2] ^ w[1]

w[4] = w[2] ^ Rcon2 ^ sub2Nib(w[3])

w[5] = w[4] ^ w[3]

def encrypt(ptext):

"""Encrypt plaintext block"""

def mixCol(s):

return [s[0] ^ mult(4, s[2]), s[1] ^ mult(4, s[3]),

s[2] ^ mult(4, s[0]), s[3] ^ mult(4, s[1])]

state = intToVec(((w[0] << 8) + w[1]) ^ ptext)

state = mixCol(shiftRow(sub4NibList(sBox, state)))

state = addKey(intToVec((w[2] << 8) + w[3]), state)

state = shiftRow(sub4NibList(sBox, state))

return vecToInt(addKey(intToVec((w[4] << 8) + w[5]), state))

def decrypt(ctext):

"""Decrypt ciphertext block"""

def iMixCol(s):

return [mult(9, s[0]) ^ mult(2, s[2]), mult(9, s[1]) ^ mult(2, s[3]),

mult(9, s[2]) ^ mult(2, s[0]), mult(9, s[3]) ^ mult(2, s[1])]

state = intToVec(((w[4] << 8) + w[5]) ^ ctext)

state = sub4NibList(sBoxI, shiftRow(state))

state = iMixCol(addKey(intToVec((w[2] << 8) + w[3]), state))

state = sub4NibList(sBoxI, shiftRow(state))

return vecToInt(addKey(intToVec((w[0] << 8) + w[1]), state))

if \_\_name\_\_ == '\_\_main\_\_':

plaintext = 0b1101011100101000

key = 0b0100101011110101

ciphertext = 0b0010010011101100

keyExp(key)

try:

assert encrypt(plaintext) == ciphertext

except AssertionError:

print("Encryption error")

print(encrypt(plaintext), ciphertext)

sys.exit(1)

try:

assert decrypt(ciphertext) == plaintext

except AssertionError:

print("Decryption error")

print(decrypt(ciphertext), plaintext)

sys.exit(1)

print("Test ok!")

sys.exit()

**Assignment 3**

import random

'''

Euclid's algorithm for determining the greatest common divisor

Use iteration to make it faster for larger integers

'''

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

'''

Euclid's extended algorithm for finding the multiplicative inverse of two numbers

'''

def multiplicative\_inverse(e, phi):

d = 0

x1 = 0

x2 = 1

y1 = 1

temp\_phi = phi

while e > 0:

temp1 = temp\_phi//e

temp2 = temp\_phi - temp1 \* e

temp\_phi = e

e = temp2

x = x2- temp1\* x1

y = d - temp1 \* y1

x2 = x1

x1 = x

d = y1

y1 = y

if temp\_phi == 1:

return d + phi

'''

Tests to see if a number is prime.

'''

def is\_prime(num):

if num == 2:

return True

if num < 2 or num % 2 == 0:

return False

for n in range(3, int(num\*\*0.5)+2, 2):

if num % n == 0:

return False

return True

def generate\_keypair(p, q):

if not (is\_prime(p) and is\_prime(q)):

raise ValueError('Both numbers must be prime.')

elif p == q:

raise ValueError('p and q cannot be equal')

#n = pq

n = p \* q

#Phi is the totient of n

phi = (p-1) \* (q-1)

#Choose an integer e such that e and phi(n) are coprime

e = random.randrange(1, phi)

#Use Euclid's Algorithm to verify that e and phi(n) are comprime

g = gcd(e, phi)

while g != 1:

e = random.randrange(1, phi)

g = gcd(e, phi)

#Use Extended Euclid's Algorithm to generate the private key

d = multiplicative\_inverse(e, phi)

#Return public and private keypair

#Public key is (e, n) and private key is (d, n)

return ((e, n), (d, n))

def encrypt(pk, plaintext):

#Unpack the key into it's components

key, n = pk

#Convert each letter in the plaintext to numbers based on the character using a^b mod m

cipher = [(ord(char) \*\* key) % n for char in plaintext]

#Return the array of bytes

return cipher

def decrypt(pk, ciphertext):

#Unpack the key into its components

key, n = pk

#Generate the plaintext based on the ciphertext and key using a^b mod m

plain = [chr((char \*\* key) % n) for char in ciphertext]

#Return the array of bytes as a string

return ''.join(plain)

if \_\_name\_\_ == '\_\_main\_\_':

'''

Detect if the script is being run directly by the user

'''

print ("RSA Encrypter/ Decrypter")

p = int(input("Enter a prime number (17, 19, 23, etc): "))

q = int(input("Enter another prime number (Not one you entered above): "))

print ("Generating your public/private keypairs now . . .")

public, private = generate\_keypair(p, q)

print ("Your public key is ", public ," and your private key is ", private)

message = input("Enter a message to encrypt with your private key: ")

encrypted\_msg = encrypt(private, message)

print ("Your encrypted message is: ")

print (" ".join(map(lambda x: str(x), encrypted\_msg)))

print ("Decrypting message with public key ", public ," . . .")

print ("Your message is:")

print (decrypt(public, encrypted\_msg))

**Assignment 4**

# -\*- coding: encoding -\*-

from \_\_future\_\_ import print\_function

# Variables Used

sharedPrime = 23

sharedBase = 5

aliceSecret = 6

bobSecret = 15

# Begin

print("Publicly Shared Variables:")

print("Publicly Shared Prime: ", sharedPrime)

print("Publicly Shared Base: ", sharedBase)

# Alice Sends Bob A = g^a mod p

A = (sharedBase\*\*aliceSecret) % sharedPrime

print("\n Alice Sends Over Public Chanel: ", A)

# Bob Sends Alice B = g^b mod p

B = (sharedBase \*\* bobSecret) % sharedPrime

print(" \n Bob Sends Over Public Chanel: ", B)

print("\n------------\n")

print("Privately Calculated Shared Secret:")

# Alice Computes Shared Secret: s = B^a mod p

aliceSharedSecret = (B \*\* aliceSecret) % sharedPrime

print("Alice Shared Secret: ", aliceSharedSecret)

bobSharedSecret = (A\*\*bobSecret) % sharedPrime

print(" Bob Shared Secret: ", bobSharedSecret)