Problem 1: Textbook RSA and hybrid encryption

A common variant of textbook RSA is the following: During key generation, the modulus $N$ is chosen as usual. We chose $e$ as $e := 3$ (instead of random). Then $d$ is chosen with $ed \equiv 1 \mod \varphi(N)$ (as usual). This is implemented by the Python functions rsa_keygen, rsa_enc, rsa_dec below.

We use this in a “hybrid encryption”, which first picks an AES key $k$, encrypts it with RSA, and then encrypts the actual message with AES using the key $k$. (Functions hyb_enc, hyb_dec.)

Your task is to write an adversary that, given the public key $pk$, and the hybrid encryption $c$ of some message $m$, finds $m$. That is, fill in the function body of the function adv below so that the function test_adv prints Success. The adversary broke the scheme.

Hint: We discuss/discussed in the practice the problem with RSA with $e = 3$ when RSA-encrypting short messages.

(You find the following file on the lecture webpage, too.)

# Use "pip install sympy" (possibly with sudo) to install sympy
# And "Crypto" might need "pip install pycrypto" if it's not installed

import sympy, math, Crypto, random

prime_len = 1024

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m
def rsa_keygen():
    while True:
        try:
            p = sympy.ntheory.generate.randprime(2**prime_len,2**(prime_len+1))
            q = sympy.ntheory.generate.randprime(2**prime_len,2**(prime_len+1))
            e = 3
            N = p*q
            phiN = (p-1)*(q-1)
            pk=(N,e)
            sk=(N,modinv(e,phiN))
            return (pk,sk)
        except Exception as e:
            pass

# Rough ad-hoc algorithm, not optimized
def exp_mod(a,e,N):
    res = 1
    b = a
    i = 0
    while e>=2**i: # Invariant: b=a**(2**i)
        if e & 2**i != 0:
            e -= 2**i
            res = (res*b) % N
        b=(b*b) % N
        i += 1
    assert e==0
    return res

# Just a test
assert exp_mod(23123,323,657238293) == ((23123**323) % 657238293)

def rsa_enc(pk,m):
    (N,e) = pk
    return exp_mod(m,e,N)

def rsa_dec(sk,c):
    (N,d) = sk
    return exp_mod(c,d,N)

def int_to_bytes(i,len): # Not optimized
    res = b"
    for j in range(len):
        res += str(i[j]).encode()
    return res
res += chr(i%256)
i = i>>8
return res

def aes_cbc_enc(k,m):
    from Crypto.Cipher import AES
    from Crypto import Random
    assert len(m)%AES.block_size == 0
    k = int_to_bytes(k,AES.block_size)
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(k, AES.MODE_CBC, iv)
    return iv + cipher.encrypt(m)

def aes_cbc_dec(k,m):
    from Crypto.Cipher import AES
    from Crypto import Random
    k = int_to_bytes(k,AES.block_size)
    iv = m[:AES.block_size]
    cipher = AES.new(k, AES.MODE_CBC, iv)
    return cipher.decrypt(m[AES.block_size:]).decode('utf-8')

# Just a test
assert aes_cbc_dec(2123414234,aes_cbc_enc(2123414234,'hello there test')) == 'hello there test'

def hyb_enc(pk,m):
    k = random.getrandbits(256)
    aes_k_m = aes_cbc_enc(k,m)
    assert aes_cbc_dec(2123414234,aes_cbc_enc(k,m))
    assert m == aes_cbc_dec(k,aes_k_m)
    rsa_pk_k = rsa_enc(pk,k)
    return (rsa_pk_k,aes_k_m)

def hyb_dec(sk,c):
    (c1,c2) = c
    k = rsa_dec(sk,c1)
    m = aes_cbc_dec(k,c2)
    return m

def adv(pk,c):
    m = "put the right message here"
    return m

def test_adv():
    pass
(pk,sk) = rsa_keygen()
# Generate a message m
m = "a few random words to be shuffle randomly to get some interesting ciphertext not random. shuffle(m)
m = " ".join(m)
# Get a key pair
(pk,sk) = rsa_keygen()
# Encrypt m
c = hyb_enc(pk,m)
# Just a test
assert m == hyb_dec(sk,c)
# Call the adversary, let him guess m
m2 = adv(pk,c)
# Check
if m==m2:
    print "Success. The adversary broke the scheme"
else:
    print "*** Failure ***"

test_adv()

Problem 2: Breaking ECB

In the lecture we have seen that encrypting a file with ECB mode is not very secure. For example, if an uncompressed image file is encrypted, the result may still reveal much of the picture to the naked eye.

In this exercise, we consider the task of distinguishing the encryption of two given messages $m_0, m_1$ automatically. That is, assume that two messages $m_0, m_1$ (English texts) of the same length are given and known to the adversary. Furthermore, the adversary learns $c$, which is the ECB encryption of $m_0$ or $m_1$ (using a random and unknown key $k$). The adversary is now supposed to guess which message was encrypted. (I.e., we have a known plaintext attack, not a chosen plaintext attack.)

(a) Describe an algorithm that finds out (given $m_0, m_1, c$) whether $m_0$ or $m_1$ was encrypted. It should work on “typical” text files. (That is, it should not require, e.g., one of the text files to contain only spaces or similar.)

Example of “typical” text files are ecb-distinguish-1.txt and ecb-distinguish-2.txt from the lecture webpage.

(b) Implement the algorithm. That is, fill in the missing code for the function adv in the code below (also available on the lecture webpage):

```python
#!/usr/bin/python
```
# And "Crypto" might need "pip install pycrypto" if it's not installed

import Crypto, random
from Crypto.Cipher import AES

def int_to_bytes(i,len): # Not optimized
    res = b"
    for j in range(len):
        res += chr(i%256)
        i = i>>8
    return res

def aes_ecb_enc(k,m):
    from Crypto import Random
    assert len(m)%AES.block_size == 0, len(m)%AES.block_size
    k = int_to_bytes(k,AES.block_size)
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(k, AES.MODE_ECB, iv)
    return iv + cipher.encrypt(m)

def aes_ecb_dec(k,m):
    from Crypto import Random
    k = int_to_bytes(k,AES.block_size)
    iv = m[:AES.block_size]
    cipher = AES.new(k, AES.MODE_ECB, iv)
    return cipher.decrypt(m[AES.block_size:]),

# Just a test
assert aes_ecb_dec(2123414234,aes_ecb_enc(2123414234,'hello there test')) == 'hello there test'

# The game: it gets a prg and an adversary as arguments,
# as well as the messages to be distinguished

def guessing_game(adv,m0,m1):
    b = random.randint(0,1) # Random bit
    k = random.getrandbits(256) # Random AES key
    seed = random.randint(0,2**32-1) # Random seed
    rand = [random.randint(0,2**32-1) for i in range(10)] # Truly random output
    msg = (m0,m1)[b]
    ciph = aes_ecb_enc(k,msg)
    badv = adv(ciph)
    return b==badv
def adv(ciph):
    # blocks contains the ciphertext as a list of blocks
    blocks = [ciph[i*AES.block_size:(i+1)*AES.block_size] for i in range(len(ciph)/AES.block_size)]

    ???
    return ??? # return 0 or 1

def test_adv(adv):
    num_true = 0
    num_tries = 3000
    m0 = open("ecb-distinguish-1.txt","rb").read()
    m1 = open("ecb-distinguish-2.txt","rb").read()
    for i in range(num_tries):
        #if i%100==0: print(str(i)+"...")
        if guessing_game(adv,m0,m1): num_true += 1
    ratio = float(num_true)/num_tries
    print ratio

    # An output near 0.5 means no attack
    # An output near 0.0 or 1.0 means a successful attack
    test_adv(adv)