Problem 1: One-time-pad in CBC mode

Assume someone uses the one-time pad in CBC mode. That is, the block cipher is
\( E(k, m_{\text{block}}) := k \oplus m_{\text{block}} \), and that block cipher is used in CBC mode.

(a) Assume a message \( m = m_1 \parallel m_2 \parallel m_3 \parallel m_4 \) is encrypted where all \( m_1 = m_2 = m_3 = m_4 \) are blocks consisting only of only zeroes.

What is the resulting ciphertext? (That is, give an explicit simple formula for each of the ciphertext blocks.)

(b) Assume a message \( m = m_1 \parallel m_2 \parallel m_3 \parallel m_4 \) is encrypted. What is the resulting ciphertext? (Give a formula in terms of the \( m_1, m_2, m_3, m_4 \), simplified as much as possible.)

(c) Explain how to compute \( m_3 \oplus m_4 \) from the resulting ciphertext. (Without using the key.)

(d) Explain why the above implies that the one-time pad in CBC mode is not IND-CPA secure (not even IND-OT-CPA).

Problem 2: “Inverse” CBC

Consider the following mode of operation (which I call “inverse CBC”):

To encrypt a message \( m \) consisting of blocks \( m_1, \ldots, m_n \) with key \( k \), pick a random initialization vector \( iv \) and then compute \( c_1 := E_0(k, m_1) \oplus iv \) and \( c_i := E_0(k, m_i) \oplus m_{i-1} \) for \( i = 2, \ldots, n \). Here \( E_0 \) is the block cipher. And \( E(k, m) := iv || c_1 || \ldots || c_n \).

The adversary has intercepted a ciphertext \( c = E(k, m) \). He happens to know the last block \( m_n \) of \( m \) (e.g., because that one is prescribed by the protocol).

(a) Explain how the adversary can completely decrypt \( m \). He can make chosen plaintext queries (i.e., he can ask for encryptions of arbitrary message \( m' \)). He cannot make decryption queries.

Hint: First think how you can, e.g., find out \( E_0(k, m_n) \) by performing an encryption query \( E(k, m_n) \).

(b) Suggest how to fix the mode of operation so that it becomes secure at least again this attack (and simple modifications thereof). You do not need to prove security.
Problem 3: 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 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
      return (p, q, e)
    except:
      pass
\[ N = p\times q \]
\[ \phi(N) = (p-1)\times(q-1) \]
\[ pk = (N, e) \]
\[ sk = (N, \text{modinv}(e, \phi(N))) \]

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

# 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():
    (pk,sk) = rsa_keygen()
    # Generate a message m
    m = "a few random words to be shuffle randomly to get some interesting ciphertext not really much sense in it but seemed fun to do instead of random bits etc bla bla"
    random.shuffle(m)
    m = " ".join(m)
    # Get a key pair
    (pk,sk) = rsa_keygen()
Problem 4: Breaking ECB (Bonus points)

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 neat 0.0 or 1.0 means a successful attack
test_adv(adv)