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:** Note that $k < 3\sqrt{N}$ here. This makes it much simpler to find $k$ given $k^3 \mod N$!

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

```
#!/usr/bin/python3

# 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
```
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 = []
for j in range(len):
    res.append(i%256)
    i = i>>8
return bytes(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:])

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

def hyb_enc(pk,m):
    assert isinstance(m,bytes)
    k = random.getrandbits(256)
    aes_k_m = 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 = b"put the right message here"
    return m
def test_adv():
    (pk, sk) = rsa_keygen()
    # Generate a message m
    m = b"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".split()
    random.shuffle(m)
    m = b" ".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)
    assert isinstance(m2, bytes)
    # Check
    if m == m2:
        print("Success. The adversary broke the scheme")
    else:
        print("*** Failure ***")

test_adv()

Problem 2: Malleability of textbook RSA

The adversary get a textbook RSA encryption \( c = E(pk, m) \) for some unknown message \( m \). The adversary also knows \( pk = (N, e) \). The adversary wants to compute \( c' = E(pk, 2m) \). (This is a specific example of malleability.) How can the adversary efficiently compute \( c' \) from \( c \) and \( pk \)?

You may assume that \( 0 \leq m < N/2 \).