Project # 1

This project is due Thursday, January 29.

1. Warm up (not graded): *Computing RSA by hand*

   Let $p = 13, q = 23, e = 17$ be your initial parameters.

   (a) *Key generation:* Compute N and $\phi(N)$. Compute the private key $kp = d = e^{-1} \bmod \phi(N)$ (e.g. by using the extended Euclidean algorithm). Show all intermediate results.

   (b) *Encryption:* Encrypt the message $m = 31$ by applying the square and multiply algorithm (first, transform the exponent to binary representation).

   (c) *Decryption:* Decrypt the ciphertext c computed above by applying the square and multiply algorithm.

2. Warm up (not graded): *RSA Cryptanalysis*

   Let $N = pq$ be the product of two distinct primes. Show that if $\phi(N)$ and N are known, then it is possible to compute p and q.
   (*Hint:* Derive a quadratic equation (over the integers) in the unknown p)

3. *Algorithms for RSA*

   (a) Implement the extended Euclidean algorithm as a function `xGCD(a,b)` in Python. Use the template given in myWPI as a reference implementation and paste your own code into the given function definition. (for more information on xGCD see algorithm 2.107 in HAC)

   (b) Implement the square and multiply algorithm in Python using the provided template.

4. *Textbook RSA*

   (a) Following the example for the RSA encryption function, write a function `RSAdec(c,d,N)` to decrypt ciphertexts. Instead of using the built-in `pow(a,b,N)` function, use your own square and multiply routine.

   (b) Implement a simplified RSA key generation algorithm `RSAKeyGen(p,q)` that outputs public and private key parameters for the given primes p and q. The public exponent e is always chosen as $e = 2^{16} + 1$.

(c) Compute the plaintext $m = c^d \mod N$ for the following parameters below.

(d) Given an RSA signature oracle returning signatures $\sigma' = \mathsf{Sign}_{sk}(m')$ for all $m' \neq m$. Show that the signature of m can be found efficiently, i.e. that *universal* forgeries are possible for RSA signatures. *Hint:* It might be helpful to consider that RSA is *multiplicative*: $m_1^e \cdot m_2^e = (m_1 m_2)^e \mod N$)

Example for the RSA encryption in Python (yes, indentation does matter in Python!):

```
def RSAenc(m,e,N): # return m^e mod N
    """Returns the RSA ciphertext of m encrypted with e and modulus N"""
    return pow(m,e,N)
```

$p =$
11774567196795186264890848937988269619683091795498048470488793460033796137085963964957988661149476070297371372186826196425817253890682262453606914957198881,
$q =$
8435082675921437589517253761085461515430725753300480036958921837140523617047618083576237472912904464351693736482550242276422621440841845620675882916844183,
and $c = 31$

5. *Straightforward AES*

(a) Implement the AES in Python. A template for this exercise is provided. Your code should support encryption and decryption for keys of 128, 192 and 256 bits. The code should take bytearrays as input (plaintext/ciphertext and key), and also return a bytearray as output.

(b) Using your implementation of AES, decrypt the following ciphertext (given in hexadecimal representation):

```
E5 5C D4 A8 EE E5 7D 26 1C 16 CA FE C9 40 A9 44
```

and the following 128-bit key:

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
```