INTE2401/2402 Lab 4

Student ID: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In this week’s lab, we implement the RSA based digital signature with JavaScript. RSA is the first practical public-key encryption algorithm devised in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and named by their names. Aside of encryption, the RSA algorithm can be used to generate digital signature, for verifying the authenticity of digital messages. The RSA based digital signature protocol involves three algorithms: key generation, signing and verifying.

As usual, you may use Notepad(or other text editor) to write a JavaScript file with the extension name **.html** and then open it with Internet Explore(or other web browser).

Task. Implementation of the RSA based digital signature (unpadded)

**Key generation**

In RSA, the encryption key (aka public key) is public and differs from the decryption key (aka private key) which should be kept secret[1]. We denote the public key as (*e, n*) and private key as (*d, p, q*).

Technically, the way to calculate the public key *e* is based on a classic mathematical problem – finding the **greatest common divisor (gcd)** of two integers based on the **Euclidean algorithm**. In this lab, you must use the Euclidean algorithm to select the public key e.

The private key d is found based on e·d=1 (mod (n)). It means , where y is an integer. Given that gcd(e, (n))=1, the equation can be wrote as ; aka, the Extended Euclidean Algorithm shown in lecture slides. Thus, for finding the private key d, you must use the **Extended Euclidean algorithm**.

**Signing and verifying**

The signer uses the signing algorithm to generate a unique signature for message m under the private key (d, p, q). The signing function is formulated as**: .** Upon receiving the message, the verifier uses the verifying algorithm to check if the received message is valid. The verifying function is formulated as: **,** where (e, n) is the public key.

Normally, the resulting integer will be **too big to handle by JavaScript Math.pow(m, d)** method. For example, the 525=298,023,223,876,953,125 in reality differs from 525=298,023,223,876,953,150 by using JavaScript Math.pow(5, 25). Thereby, we use **fast exponentiation** introduced in lecture’s slides.

Consider computing 2623 (mod 51), we first convert the power 23 as binary 10111. Then we compute 2610111= 2616+4+2+1=26 **·**262 **·** (262)2  **·** (0\*((262)2)2)**·** (((262)2)2)2 (mod 51).

Each time, we will modulo 51 such that the exponentiation result won’t exceed the limitations of integer.

Sample form:

