# ECE 404 Homework #6

Due: Tuesday 02/28/2023 at 5:59 PM

## Introduction

The goal of this homework is to give you a deeper understanding of RSA encryption and decryption, its underlying principles and standard representation.

Before starting this assignment, make sure that you understand the relationship between the modulus and the block size for RSA cipher and how RSA is made practically possible by the fact that modular exponentiation possesses a fast implementation. Also, before starting to write your own code for RSA, play with the script PrimeGenerator.py that is discussed in Lecture 12. You can download the script from the lecture notes web site.

### Part 1: RSA Encryption and Decryption

Write a Python script to implement a 256-bit RSA algorithm for encryption and decryption. The plaintext message has been provided in the zip file and is called message.txt. Your data block from the text will be of 128-bits. For the reasons explained in Section 4 of Lecture 12 (12.4), prepend it with 128 zeroes on the left to make it a 256-bit block. For this assignment, if the overall plaintext length is not a multiple of 128 bits, pad an appropriate number of zero bits from the right so that it becomes a multiple of 128 bits (Also do not forget to pad again to make it a 256-bit block as mentioned previously). This way of creating blocks is a little tricky so make sure you understand it or you will have issues when trying to get the correct encryption result.

Regarding key generation, remember the following points:

1. The priority in RSA is to select a particular value of $e$ and choose $p$ and $q$ accordingly. For this assignment, use $e = 65537$.

2. Use the PrimeGenerator.py script mentioned above (you can import it to your script) to generate values of $p$ and $q$. Both $p$ and $q$ must satisfy the following conditions:

   (a) The two leftmost bits of both p and q must be set.

   (b) $p$ and $q$ should not be equal.

   (c) $(p-1)$ and $(q-1)$ should be co-prime to $e$. Hence, $gcd(p-1, e)$ and $gcd(q-1, e)$ should be 1. Use Euclid's algorithm to compute the gcd.

   If any of the above condition is not satisfied, repeat Step 2.

3. Compute $d$. You may use the *multiplicative_inverse* function from Python's BitVector class.

4. To compute the modular exponentiation for decryption, use the Chinese Remainder Theorem (CRT). Implementation details are in Section 12.5 of the lecture notes.

5. After decryption, remove the padded 128 zeroes from each block to make the plaintext printable in ASCII form.

**Program Requirements**

Your script for should have the following command-line syntax:

```
python rsa.py -g p.txt q.txt
python rsa.py -e message.txt p.txt q.txt encrypted.txt
python rsa.py -d encrypted.txt p.txt q.txt decrypted.txt
```

An explanation of this syntax is as follows:

- **For key generation** (indicated with -g):
  - The generated values of $p$ and $q$ will be written to **p.txt** and **q.txt**, respectively. The .txt files should contain the number as an integer represented in ASCII. So, for example, if $p = 7$, the corresponding text file will display 7 when opened in a text editor.
- **For encryption** (indicate with -e)
  - Read the input text from a file called **message.txt** (or whatever the name of the command-line argument after -e is) and use the $p$ and $q$ values found in the command-line arguments **p.txt** and **q.txt** for encryption. For performing encryption and decryption, use the p and q values we have provided you with. The key generation step mentioned in the previous bullet point is there to simply make you aware of the necessity in real world applications.
  - The encrypted output should be saved in **hexstring** format to a file with the name of the final argument, in this case a file called **encrypted.txt**.
- **For decryption**(indicated with the -d argument):
  - The input ciphertext file (in hexstring format) is specified with argument after -d, in this case **encrypted.txt**. As with encryption, use the $p$ and $q$ values found in the command-line arguments **p.txt** and **q.txt** to decrypt the ciphertext.
  - The decrypted output should be saved to a file with the name specified by the last argument, in this case **decrypted.txt**.

Remember to parse the command-line arguments for your program using the calling conventions described above. Please do not hard-code the file names into your program.

**Part 2: Breaking RSA Encryption for small values of $e$**

Section 12.3.2 in Lecture 12 describes a method for breaking RSA encryption for small values of $e$, like 3. In this scenario, a sender $A$ sends the same message M to 3 different receivers using their respective public keys. All of the public keys have the same value of $e$, but different values of $n$. An attacker can intercept the three cipher texts and use the Chinese Remainder Theorem to calculate the value of $M^3$ mod $N$, where $N$ is the product of the values of $n$. The attacker can then solve the cube-root to get the plaintext message $M$. Write a script that does the following:

1. Generates three sets of public and private keys with $e = 3$
2. Encrypts the given plaintext with each of the three public keys
3. Takes the three encrypted files and the public keys, and outputs the decrypted file as cracked.txt. Because Python's *pow*() function will not provide enough precision to solve the cube-root, we have provided code that should provide the necessary precision. This code is listed in `solve_pRoot.py`.

### Program Requirements

Your script should have the following call syntax:

```
python breakRSA.py -e message.txt enc1.txt enc2.txt enc3.txt n_1_2_3.txt #Steps 1 and 2
python breakRSA.py -c enc1.txt enc2.txt enc3.txt n_1_2_3.txt cracked.txt #Step 3
```

An explanation of this syntax is as follows:

- **For Encryption** (indicated with the `-e` argument):
  - The program should read in the plaintext file (in this case **message.txt**).
  - Generate the three different public and private keys, encrypts the plaintext with each of the three public keys $(n_1, n_2, n_3)$, and write each ciphertext to **enc1.txt**, **enc2.txt**, and **enc3.txt**, respectively.
  - Then the program should write each of the public keys $(n_1, n_2, n_3)$ to **n_1_2_3.txt**, with each key separated with a newline character.
  - **For cracking the encryption** (indicated with the `-c` argument):
    - The program should read each of the different encrypted files (in this case, from **enc1.txt**, **enc2.txt**, and **enc3.txt**) and the public keys (in this case, from **n_1_2_3.txt**). For testing purposes, you can use the provided `n_1_2_3.txt` file.
    - Then, use this information to crack the encryption, and then write the recovered plaintext to a file (in this case, **cracked.txt**).

  Remember to parse the command-line arguments for your program using the calling conventions described above. **Please do not hard-code the file names into your program.**

## Submission Instructions

- Make sure to follow program requirements specified above. **Failure to follow these instructions may result in loss of points!**.
- For this homework you will be submitting a zip file titled `HW06_<last_name>_<first_name>.zip` to Brightspace containing:
  - The file `rsa.py` containing your code for Part 1.
  - The file `breakRSA.py` containing your code for Part 2.
  - You can import `PrimeGenerator.py` and `solve_pRoot.py` into your .py files with the assumption that it will be in the same directory as your files when being graded.
  - a pdf titled `HW06_<last_name>_<first_name>.pdf` containing a detailed explanation of how you implemented RSA in part 1 and the CRT to break RSA in part 2.
- In your program file, include a header as described on the ECE 404 Homework Page.