HW06 - ECE404

Part 1 – RSA Encryption and Decryption

Explanation of Code: The implementation of RSA can be broken down into two functions: the key generation and the encryption. The Python script implements a 256-bit RSA algorithm with a provided message text. For my key generation function I use an 'e' value of 65537 that is given to us, where 'e' is an integer that is coprime to the totient of n, also known as the public exponent. First, I initialize the value of e, and then using the Prime Generator class, I create an instance of the it called generator with the input number of bits as 128. Then, my calling the findPrime() function twice (defined in the Prime Generator class), I assign two different values to p and q, where p and q are defined as two prime numbers. If we consider arithmetic modulo 'n' and assume that 'e' is an integer that is coprime to the totient of n where the totient of n is equal to the product of (p – 1) and (q – 1), then we can find its gcd. Therefore, next I calculate the gcd(p - 1, e) and gcd(q - 1, e). To generate values of p and q there are 3 conditions that must be met:

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

Until all these three conditions are met, I run a while loop that randomly generates values for p and q. Once the conditions are met, p and q are written to two different files.

The encrypt function takes in the message file, value of p, value of q, and output file as inputs. It converts the message text into a bit vector and reads 128 bits at a time. If the input is not 128 bits, it is padded from the right. Then we calculate 'n' as the product of p and q, obtained from the key generation function. Next, using the formula provided below we generate the ciphertext block as an integer, and then convert it to a bit vector.

$$C = M^e \bmod n$$

In the above formula, 'C' denotes the ciphertext block, 'M' denotes the message text block (256 bit integer), and {e, n} is the public key.

Once the ciphertext block is converted to a bit vector, it is padded to the left with zeroes, and written as a hex string to an output file. This is done until the entire message is read and encrypted.

The decrypt function for RSA, takes in the ciphertext, p value, q value and output file as an input. Here we first read the values of p and q and calculate 'n' as their product. Then, the totient of n 'phi' is calculated as the product of (p – 1) and (q – 1). The integer 'e' is then initialized and converted to a bit vector. Once we the all these values, we now use the private key {d, n} to decrypt the message. The value for 'd' is calculated as follows:

$$d = e^{-1} \ mod \ \phi(n)$$

We take the multiplicative inverse of e modulo (totient n), to obtain 'd'. Then, we open the ciphertext file and read 256 bits at a time and decrypt them using the formula below.

$$C^d \bmod n.$$

In the above formula, 'C' denotes the ciphertext block, and {d, n} is the private key, where 'd' is the private exponent. This is equated to the plaintext block or the message text. This is a 256 bit integer, that is then converted to a bit vector and divided into two halves. We are interested in the right half of the two of them, since the first 128 bits are simply zeroes that we padded them with during encryption. Once the 128-bit message block is obtained it is written to an output file in ascii.

Part 2 – Breaking RSA Encryption for small values of e

Explanation of Code: For the encrypt function in Part 2, we were to implement the same logic but with a small value of 'e'. where 'e' is the public exponent in public key cryptography. First, I ran a for loop that went through 3 iterations and generated 3 sets of values for p and q using the key generation function. This generated 3 different values for 'n' where 'n' denotes the product of p and q. Then, using the logic from part 1, I implemented the RSA encryption algorithm for values of 'n'. Each of the encrypted texts were written to 3 different output files. The three different values of 'n' were also written to an output file. To summarize, I generated three sets of public and private keys with e = 3, and encrypted the given plaintext with each of the three public keys. To crack the RSA algorithm when 'e' is a small number I used the Chinese Remainder Theorem method. The cracked() function first reads the three public keys from the file, and stores their product in an integer 'N'. Then, using CRT we know that we can write A mod B as $(\sum_{i=1}^{k} c_i * a_i) \bmod B$, where a_i = A mod b_i and every b_i is a pairwise coprime factor of B. Therefore, using this fact we know that,

- $C_1 = M^3 \bmod n_1$
- $C_2 = M^3 \bmod n_2$
- $C_3 = M^3 \bmod n_3$

where C1, C2 and C3 are ciphertext blocks, and M is the message block. Using this method, I first calculated N1, N2 and N3 which were done by dividing N by each of the public keys. For instance,

N1 = N / n1. Then, I converted each of them to a bit vector. Furthermore, using the formula below I obtained a constant c1, c2, and c3 for each of the public keys.

$$c_1 = \left(N_1^{-1} \mod n_1\right) \cdot N_1$$

Then, implemented a for loop until the length of one of the ciphertext files. Each 256 bit block was taken from the three encrypted files and multiplied by c1, c2 and c3 respectively as shown below. Finally, we take the sum of the three products, for each 256-bit block.

$$M^3 = c_1 * M_1 + c_2 * M_2 + c_3 * M_3$$

In the equation above, M1 denotes each 256-bit ciphertext bit vector converted to an integer from the first encrypted file, M2 denotes each 256-bit ciphertext bit vector converted to an integer from the second encrypted file and M3 denotes each 256-bit ciphertext bit vector converted to an integer from the third encrypted file. After that I performed mod N arithmetic on M_cube, and used the solve_pRoot() function provided to take the cube root of M_cube, since 'M' denoted the message block, not M_cube. Then, each value of M was converted to a bit vector and indices [128:256] were written in ASCII to an output file. It was indexed in the format above to ignore the zero padding done for encryption. In this manner, we could crack the RSA algorithm using the Chinese Remainder Theorem.