Luke Canfield ECN Login: lcanfiel Due Date: 2/27/2024

HW06 ECE 40400

Part 1: RSA Implementation

I didn't mess with the PrimeGenerator at all for part 1. I read the p and q values directly from the p.txt and q.txt files and then calculate the modulus n from the values read. I opened the input file and turned it into a bitvector before opening the ciphertext output file. Then I have a while loop which reads 128 bits from the bitvector padding the bitvector from the right if the length isn't 128 bits and afterwards pads the bitvector from the left with 128 zeros. Then I use the pow function to calculate the cipher integer and then create a bit vector from the cipher integer and read it as hex into the ciphertext file.

For the decrypt function I open the ciphertext file and create a bitvector variable which reads it as a hex string. Then I open the p.txt and q.txt files as before and once again calculate the modulus n. Then I calculate the totient of n and create a bitvector variable for e and the totient of n. I calculated d by taking the multiplicative inverse of e mod the totient n. I then create bitvector variables for p and q so that I can calculate Xp and Xq which the equations can be found on Lecture 12 page 37. I then open the recovered plaintext file and have a "for" loop which iterates through the bitvector calculating the Vp, Vq, and plaintext variables which the equations for are also found on Lecture 12 page 37. Once I have the plaintext which is an integer, I create a bitvector variable setting the intVal to the plaintext and setting the size to 256. Then I create another variable which reads the right 128 bits from the plaintext bitvector variable and I write the rightmost bits as ascii to the recovered plaintext file.

Part 2: CRT Implementation

For the encrypt function I imported the PrimeGenerator class from the file we were given to generate the three p's and q's needed. The only difference in my encrypt function from the rsa.py file is I create a list of p's and q's using a function I made which calls the generator.findPrime from the PrimeGenerator class and from that I create a list of modulus n's which I write to the n_1_2_3.txt file. Then I have the same while loop as before but it calculates three ciphertexts from the three different modulus n's and writes them to the enc1.txt, enc2.txt, enc3.txt files.

For my break function I open the three ciphertext files and create a bitvector variable for each which reads the file as a hex string. Then I read the three modulus n's from the n_1_2_3.txt file and create a variable which is a list of the values. I then calculate the mod N which is the modulus n's multiplied together. Then I calculate the pairwise values from the equation on Lecture 11 page 56. I also create a bitvector variable for each of the modulus n's and pairwise values so that I can calculate c1, c2, and c3 for the CRT function. The equations for each c can be found on Lecture 11 page 58. I have a similar "for" loop implementation where I iterate through the bitvector of the first ciphertext file (all files are the same size) and calculate the plaintext integer value from the equation found on Lecture 11 page 58. I then calculate the cube root of the plaintext integer using the solve_pRoot function we were given. Then I create a bitvector variable setting the intVal as the cube root value and the size of the bitvector to 256 bits. Finally I create a variable rightmost_bits that reads the right 128 bits from the bitvector just created and then I write the rightmost_bits as ascii to the cracked.txt file.