hw06_Panchagnula_Raghava

Homework Number: HW06

Name: Raghava Vivekananda Panchagnula

ECN Login: rpanchag

Due Date: 2/27/2024

Code uses functions given in lecture notes, abiding instructions given in HW document.

```
(.venv) vivek@vivek-Inspiron-13-5378:~/Files/coursework/ECE-40400/Homework/HW06$ make test
python3 rsa.py -g p.txt q.txt
python3 rsa.py -e message.txt p.txt q.txt encrypted.txt
python3 breakRSA.py -e message.txt enc1.txt enc2.txt enc3.txt n_1_2_3.txt
python3 rsa.py -d encrypted.txt p.txt q.txt decrypted.txt
python3 breakRSA.py -c enc1.txt enc2.txt enc3.txt n_1_2_3.txt cracked.txt
```

Commands to run all required code.

```
(.venv) vivek@vivek-Inspiron-13-5378:~/Files/coursework/ECE-40400/Homework/HW06$ make submit
zip -r hw06_Panchagnula_Raghava.zip hw06_Panchagnula_Raghava.pdf rsa.py breakRSA.py
        zip warning: name not matched: hw06_Panchagnula_Raghava.pdf
updating: rsa.py (deflated 68%)
updating: breakRSA.py (deflated 67%)
```

Commands to create the output (ignore zip warning as PDF wasn't created for this example.

RSA.py

- Class RSA
    - Init
        - The __init__ function initializes an object with an attribute e, which is defined above in the file. It attempts to read values for p and q from files, defaulting to 0 if the files are not found. Then, it calculates n as the product of p and q. The variable d is then determined as the modular multiplicative inverse of e modulo the totient of n, where the totient is computed as (p-1)(q-1). If p and q are not set (both are 0), d is set to 0.
    - Encrypt
        - The plaintext file is read, and an output file is opened. The function then iterates through the input file, encrypting each 128-bit block. Padding is applied, first from the right to ensure a block size of 128 bits, then from the left to reach a total size of 256 bits. The block is encrypted by taking the bitvector to the power

of e in n modulus, and the result is written to the output file in hexadecimal format. Finally, the output file is closed.

- Decrypt
  - The decrypt function reads a ciphertext file, treating it as a hex string. It then utilizes a counter to iterate through the file, reading 256 bits at a time. CRT is used to decrypt each block, and the resulting plaintext block is obtained by removing the padding. The decrypted text is written to the output file in ASCII format. Notably, the function uses a counter "i" instead of more_to_read due to an issue with more_to_read when reading from a hex file. The loop increments the counter by 256 in each iteration. Finally, the output file is closed.
- Generate_keys
  - The generate_keys function creates and saves prime numbers p and q into specified files (p_file and q_file). It opens these files for writing and uses a PrimeGenerator to generate 128-bit prime numbers. The function ensures that the leftmost 2 bits of both p and q are set to 1 and regenerates them if not. Additionally, it ensures that p and q are not equal and that the greatest common divisor (gcd) of p-1 and e, as well as q-1 and e, is 1. If any of these conditions are not met, the function regenerates p and q. Finally, the generated p and q are written to their respective files, and the files are closed.
- Exec_crt
  - Following the steps outlined in Avi Kak's lecture and class notes, the function calculates vP and vQ using modular exponentiation. It then computes the modular multiplicative inverses of q modulo p and p modulo q. The Chinese Remainder Theorem is executed to obtain the plaintext (pt) using the formula (vP * xP + vQ * xQ) mod n. The resulting int val of the plaintext is returned by the function.
- Find_gcd
  - The find_gcd function efficiently computes the greatest common divisor (GCD) of two integers, a and b, using the Euclidean algorithm. The function iteratively replaces a with b and b with the remainder of the division a by b until b becomes zero. The final absolute value of a is returned as the GCD.

- Main function
  - The main block checks the command-line arguments and executes the corresponding actions based on the specified flag. If the flag is '-g', it expects two additional arguments for the filenames of 'p' and 'q' and generates RSA keys using the provided exponent e. If the flag is '-e' or '-d', it expects four additional arguments for input and output filenames, 'p' and 'q' filenames. In these cases, it initializes an RSA object with an exponent of 65537 and either encrypts or decrypts the input file accordingly. If an invalid flag is provided, it prints an error message and exits with a status code of 1.

breakRSA.py

- Find_gcd
  - Same as above
- Generate_keys

    - Mostly same as above, returns keys as ints instead of writing to files
- Encrypt
    - For each output file, the function generates new random primes 'p' and 'q' and calculates 'n'. The public key 'e' is fixed at an unknown value in the code, and the private key 'd' is calculated but not utilized. The function writes 'n' to the specified file. It then reads the input message file in 128-bit blocks, encrypts each block using the RSA encryption formula, and writes the resulting ciphertext to the respective output file in hexadecimal format. The function repeats this process for each of the three output files. Finally, it closes all files to complete the encryption process.
- Crt
    - It calculates three auxiliary values (n1, n2, n3) as the product of all moduli except the current one. The modular multiplicative inverses of these auxiliary values are then computed. The final decrypted block (x) is determined as the sum of the product of each encrypted block, its respective auxiliary value, and the modular multiplicative inverse. The result is taken modulo N, and the decrypted block is returned.
- Crack
    - It reads the modulus values (n_list) from a specified file (nFile) and calculates the overall modulus N. The function then reads the encrypted files into hex strings, iterating through the files and decrypting each 256-bit block using the crt function. A counter (a) is used instead of more_to_read due to an issue with the latter when reading from hex files. I discussed this issue earlier for RSA.py. For each block, the cube root of the decrypted value is calculated, and the result is written to the output file after removing padding. The process continues until all blocks have been decrypted and written to the output file.