Assignment 6: Public Key Cryptography

By Derfel Terciano

Background

Public key cryptography is a security measure that can be used to help encrypt and decrypt files. This system uses pairs of keys that consists of a public and a private key. The most popular public key cryptography algorithm used today is the RSA algorithm which was created by Ronald (R)ivest, Adi (S)hamir, and Leonard (A)dleman.

In this assignment, we will be using the RSA algorithm in order to generate a public and a private key pair that will be used to help encrypt and decrypt our files. RSA uses the idea of factoring two very large numbers in order to encrypt a message. The algorithm makes use of creating a public and a private key. The public key is what is used to encrypt the file and this key is known to everyone which is why it is public. Once a message is encrypted with a public key, it can only be decrypted with a private key which is not available to everyone.

The public key uses the modulus of some n and the public exponent of e. The private key uses a private exponent of d which is to be kept very secret. In addition, the variables that are used to calculate d must be kept secret. That means p, q, and $\varphi(n)$ must be kept secret as well. Fortunately, those variables can be discarded after.

Now, let's dive deeper into the specifics of this assignment. That way we can get a good understanding of how we will be encrypting files using public and private keys.

Number Theoretic Functions

In order for us to create a public or private key, we would need to first create some function that can handle the big parts of the calculations of the keys. The following functions that we will implement are what drive the RSA algorithm:

- Modular Exponentiation
- Miller-Rabin primality testing

- Modular Inverses

Modular Exponentiation

- void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)
    - This function uses a fast modular exponentiation technique that computes a base raised to the exponent mod n.
    - The pseudocode for this is not needed since it is provided in the assignment doc.

Miller-Rabin Primality Testing

- The main purpose of this function is to check whether or not a number is prime or not.
- void is_prime(mpz_t n, uint64_t iters)
    - This function uses the Miller-Rabin primality test to determine where or not *n* is prime or not. Iters determines how many iterations of the test we should use.
    - The pseudocode for this function is not needed since it already given to us in the assignment doc
- void make_prime(mpz_t p, uint64_t bits, uint64_t iters)
    - This function should generate a new prime number that is nbits long, and is prime using the is_prime() function.
    - Below is some python pseudocode I made:
    - ```
      def make_prime(bits, iters):
          #create a variable that will store the random num
          while (is_prime(random_num, ) == False and bitsize(random_num) == nbits):
              #generate_random number
      ```
    - The pseudocode was made by me.

Modular Inverses

- We will be needing to find the greatest common divisor in order to determine our public exponent. In this assignment, we will be using the Euclidean algorithm in order to find the greatest common divisor.
- void gcd(mpz_t d, mpz_t a, mpz_t b)
    - This function uses the Euclidean algorithm in order to find the gcd.
      ```

    - No pseudocode is needed here since it is already given to us in the assignment document
- void mod_inverse(mpz_t i, mpz_t a, mpz_t n)
    - This function computes the inverse of i of mod *n*. If we cannot find the inverse then we would return 0.
    - No pseudocode is needed for this since it is already provided in the assignment doc.

The RSA Library

In order to make our public cryptography easier to implement, we will break each major part of the program into smaller parts. Each function in this library makes implementing the keygen, encryption, and the decryption binaries easy and fun.

- void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters)
    - This function creates 2 big primes, calculates their product *n* around the size of *nbits*. Also, we need to calculate the public exponent of the algorithm by finding a number that is coprime to the totient of p and q.
    - Remember: the totient is (p-1)(q-1)
    - Below is python pseudocode as to how I will kind of implement this function:
    - ```
      def rsa_make_pub(p, q, n, e, nbits, iters):
          #p_bits = random_number % (nbits/2) + (nbits/4)
          #q_bits = nbits - p_bits

          #make_prime(p,  p_bits, iters)
          #make_prime(q, q_bits, iters)

          #n = p * q

          #totient = (p-1) * (q-1)

          #e = coprime(random_number_of_nbits, totient)
          #(coprime will use the gcd() function)

          return p, q, n, e
      ```
    - NOTE: The pseudocode was made by me.
- void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
      ```

    - This function writes the public rsa key to a file. The public rsa key is usually generated from the rsa_make_pub() function.
    - We also need to output the file in the following manner:
        - Public modulus n
        - Public exponent e
        - Signature s
        - Username[]
    - I will not be providing any pseudo code for this due to the fact that we will just use 3 lines of gmp_fprintf() statements to print n, e, s and a single fprintf() to write the username[] to the file.
    - On another note: the outputs must be printed in a hex string and not in a decimal
- void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
    - This function reads the output of rsa_write_pub in the same order. However, instead of using gmp_fprintf() and fprintf(), we will be using gmpg_fscanf() and fscanf() in the same order as rsa_write_pub().
    - With that being said, pseudocode is not necessary for this function since it is a very simple function.
- void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)
    - This function generates the private key *d*. The private key is based on the primes *p*, *q* and the public exponent *e*.
    - *D* is computed as $d = e^{-1} mod\ \varphi(n)$ where the totient equals (p-1)(q-1).
    - No pseudocode is needed here since we will just be calling our inverse_mod() function in order to find *d*.
- void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)
    - This function gmp_frpintf() the public modulus *n*, then the private key *d* to the outfile or *pvfile* in this case.
    - No pseudocode is needed here since we will be utilizing only gmp_fprintf().

- NOTE: everything should be outputted as a hex string with a newline at the end.

- void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)
  - This function is the same as rsa_write_priv() except we will be using gmp_fscanf() instead.
  - With that being said, no pseudocode is necessary.
  - NOTE: We need to keep in mind that our inputs will be in hex strings.
- void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
  - This function encrypts the message m with the following formula:
    - $E(m) = c = m^e (mod\ n)$
  - No pseudocode is needed here since all we need to do is call our pow_mod() function.
- void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)
  - This function converts all the messages into a big integer, and then encrypts that integer. This continuously happens until we reach the end of file.
  - Realistically, we can't just encrypt the entire file at once so in order to do this, we must encrypt only blocks of bytes at a time.
  - We do this using the following pseudocode in python:

```
def rsa_encrypt_file(infile, outfile, n, e):
    k = math.floor((math.log(n)-1)/8)

    block[k] #array must have k elements

    block[0] = 0xFF #prepend first element

    while(readline()):
        #read(k-1 bytes)
        #mpz_import(read bytes)
        #rsa_encrypt(read_bytes_from_mpz_import)
        #write(rsa_encrypt output)
    return None
```

  - The pseudocode was made by me
- void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n)

- This function takes in an encrypted message and decrypts it using the following formula:
- $D(c) = m = c^d (mod\ n)$
- No pseudocode is needed here since all we need to do is implement the above equations.

- void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)
  - This function processes an encrypted file and decrypts it. The resulting output of the function would be a decrypted message that has the same message as the original file given to rsa_encrypt_file()
  - Below is some python pseudocode that shows how I would approach writing this function:

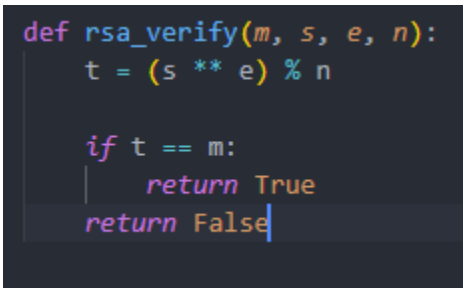
```
#!/usr/bin/env python3
import math

def rsa_decrypt_file(infile, outfile, n, d):
    k = math.floor((math.log2(n) - 1) / 8)

    block = [0 for i in range(k)]

    while (input() != EOF):
        #convert hex to decimal
        #rsa_decrypt()
        #mpz_export() decimal number to block
        #write out block
    return None
```

  - NOTE: The pseudocode was created by me.
- void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)
  - Signing is calculated by using the following:
    - $s = m^d (mod\ n)$
  - No pseudocode is needed here since we will be using a call to our pow_mod() function.

- bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)
    - This function verifies that our signature is authentic. This is done by comparing the expected message *m* to the inverse power mod of rsa_sign. The signature is only verified if *m* is equal to the inverse power mod of rsa_sign.
    - Below is some python pseudocode that I made in order to help me approach this function:
    - 
    - NOTE: This pseudocode was made by me.

Keygen, Encryptor, and Decryptor

- Overall, the assignment doc explains how to write the keygen, encryptor, and decryptor pretty well. With that being said, I do not plan on writing pseudocode for any of the three files however, I will be writing down what the command line options are for each three.

- **Keygen**
    - My keygen program will take in the following command line options:
        - -b:
            - This specifies the minimum bit length required for our public modulus *n*.
        - -i:
            - This tells the make_prime() function how many iterations of the Miller-Rabin algorithm we should use.
            - By default, 50 iterations will be used.
        - -n *pbfile*

        - This specifies the location of where to generate the file that contains the public files.
        - By default, a file named “rsa.pub” will be created locally
    - -d *pvfile*
        - This specifies the location of where the keygen should generate the
    - -s:
        - This specifies the random seed for the GMP random state initialization.
        - By default, the seed is set to the time since the UNIX epoch.
    - -v:
        - This enables the verbose output of the keygen. The verbose output here is the statistics of the keygen.
        - The verbose output displays the following statistics in order:
            - Username
            - Signature s
            - The first large prime p
            - The second large prime q
            - The public modulus n
            - The public exponent e
            - The private key d
    - -h:
        - This displays a help message and exits the program.

- **Encryptor**
    - The encryptor will take in the following options:
        - -i:
            - This specifies the file that you want encrypted.
        - -o:

            - This specified the location of where you want the encrypted file to be.
        - -n:
            - This specifies where the public rsa key is located.
        - -v:
            - This prints our verbose statistics about the public key in the following order:
                - Username
                - Signature *s*
                - Public modulus *n*
                - Public exponent *e*
        - -h:
            - Prints out a help message then exits the program.
- **Decryptor**
    - The following are the command-line options for my decryptor:
        - -i:
            - This specifies the location of the encrypted file you would like to decrypt.
        - -o:
            - This specifies the location of where you want the file to be decrypted.
        - -n:
            - This specifies the location of the private RSA key.
        - -v:
            - This displays verbose statistics about the private key in the following order:
                - The public modulus *n*
                - The private key *e*
        - -h:

- This outputs a help message and then it quits the program.

Credits

- 11/9: I attended Eugene's section and he gave me insight on how to approach all the number theory functions.
- 11/9: The Discord channel had a discussion about how there will be some bits being lost while doing our bit calculations in the rsa library. After some mathematical proofs that other students have contributed, it turns out that in order to fix our bit loss, we would need to just add an offset to our desired bit length.
- 11/12: Prof provided the python code for calculating log base 2.