Viveka Agrawal
Professor Veenstra
CSE13S, Winter 2023

CSE13S ASGN5 DESIGN DOC

Description of the program:

In this assignment, we are creating public and private keys to be either encrypted or decrypted using the Schmidt-Samoa algorithm.

In this assignment, these source and header files must be created and submitted on git. They include:

decrypt.c: This contains the implementation and main() function for the decrypt program.

It accepts the follwing command-line options:
• -i: specifies the input file to decrypt (default: stdin).
• -o: specifies the output file to decrypt (default: stdout).
• -n: specifies the file containing the private key (default: ss.priv).
• -v: enables verbose output.
• -h: displays program synopsis and usage.

encrypt.c: This contains the implementation and main() function for the encrypt program.

It accepts the follwing command-line options:
• -i: specifies the input file to encrypt (default: stdin).
• -o: specifies the output file to encrypt (default: stdout).
• -n: specifies the file containing the public key (default: ss.pub).
• -v: enables verbose output.
• -h: displays program synopsis and usage.

keygen.c: This contains the implementation and main() function for the keygen program.

It accepts the follwing command-line options:
• -b: specifies the minimum bits needed for the public modulus n.
• -i: specifies the number of Miller-Rabin iterations for testing primes (default: 50).
• -n pbfile: specifies the public key file (default: ss.pub).
• -d pvfile: specifies the private key file (default: ss.priv).
• -s: specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL)).
• -v : enables verbose output.
• -h : displays program synopsis and usage.

numtheory.c: This contains the implementations of the number theory functions.

numtheory.h: This specifies the interface for the number theory functions.

randstate.c: This contains the implementation of the random state interface for the SS library and number theory functions.

randstate.h: This specifies the interface for initializing and clearing the random state.

ss.c: This contains the implementation of the SS library

ss.h: This specifies the interface for the SS library.

A makefile, readme document, and writeup must also be completed for this assignment.

Pseudocode:

randstate.c
Include all appropriate header files required

Create a function randstate_init which has an integer parameter named seed that initializes the global random state named state with a Mersenne Twister algorithm, using seed as the random seed.

- Call gmp_randinit_mt()
- Call gmp_randseed_ui()

Create a function randstate_clear which clears and frees all memory used by the initialized global random state named state.

- Call gmp_randclear()

<u>numtheory.c</u>

Include all appropriate header files required

Create a power modulus function called pow_mod that performs fast modular exponentiation. It computes base (a) raised to the exponent (d) and power modulo (n).

- Set variable v to 1
- Set variable p to a
- While d is greater than 0
  - If d is odd
    - Set v equal to v times p modulus n
  - Set p equal to p times p modulus n
  - Set d equal to d divided by 2
- Return v

Create a function called is_prime that conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations. The parameters for the function are a number (n) and the number of iterations (k).

- Write $n - 1 = 2^s r$ such that r is odd
- For variable i = 1 to k
  - Choose random variable a in the range 2 to n - 2

        - Call the power modulus function passing in a, variable r, and n and set that value as the value for variable y
        - If y doesnt equal 1 and y doesnt equal n - 1
            - Set variable j equal to 1
            - While j is less than or equal to s - 1 and y does not equal n - 1
                - Call the power modulus function passing in y, 2, and n and set that value as the value for variable y
                - If y equals 1
                    - Return false
                - Increment j by 1
            - If y does not equal n - 1
                - Return false
- Return true

Create a function called make_prime that generates a new prime number stored in p. The generated prime should be at least bits number of bits long. The primality of the generated number should be tested using is_prime() using iters number of iterations.

- Generate a random number
- While checking if the generated random number is prime by calling is_prime() is prime and the generated number is at least bits number of bits long
    - Store the random number in variable p

Create a function called gcd that computes the greatest common divisor of a and b.

- While b does not equal 0
    - Set variable t equal to b
    - Set b equal to a mod b
    - Set a equal to t
- Return a

Create a function called mod_inverse that computes the inverse a of a modulo n.

- Set variable r to n
- Set variable r' to a
- Set variable t to 0
- Set variable t' to 1
- While r' does not equal 0
  - Set variable q to r divided by r'
  - Set r equal to r'
  - Set r' equal to r - q times r'
  - Set t equal to t'
  - Set t' equal to t - q times t'
- If r is greater than 1
  - Return no inverse
- If t is less than 0
  - Set t equal to t plus n
- Return t

<u>ss.c</u>
Include all appropriate header files required

Create a function ss_make_pub that creates parts of a new SS public key: two large primes p and q, and n computed as p*p*q.
- Create prime p using make_prime()
- Create prime q using make_prime()
- Set number of bits for p (pbits) in range [nbits/4,(3×nbits)/4)]
- Set number of bits for q (qbits) equal to nbits - pbits
- Check that $p \nmid q-1$ and $q \nmid p-1$
- Compute gcd(p−1, q−1)

Create a function ss_write_pub which writes a public SS key to pbfile. The parameters include n, the username, and the pbfile

- Print out the variable n
- Print out the usernames to pbfile

Create a function ss_read_pub which reads a public SS key from pbfile. The parameters include n, the username, and the pbfile

- Scan variable n from pbfile
- Scan the usernames from pbfile

Create a function ss_make_priv which creates a new SS private key. The parameters include d, pq, p, and q

- Compute inverse of n modulo $\lambda(pq) = \mathrm{lcm}(p-1, q-1)$

Create a function ss_write_priv which writes a private SS key to pvfile. The parameters include pq, d, and pvfile

- Print out the variable pq to pvfile
- Print out the variable d to pvfile

Create a function ss_read_priv which reads a private SS key from pvfile. The parameters include pq, d, and pvfile

- Scan variable pq from pbfile
- Scan variable d from pbfile

Create a function ss_encrypt which performs SS encryption, computing the ciphertext c by encrypting message m using the public key n. The parameters include c, m, n

- Call the pow_mod function with c, m, n, n passed in to solve for $c = m^n \pmod n$

Create a function ss_encrypt_file which encrypts the contents of infile, writing the encrypted contents to outfile. The parameters include infile, outfile, and n

- Calculate block size k where k = [(log2 (p n)−1)/8]
- Dynamically allocate an array that holds k bytes
- Set the zeroth byte of the block to 0xFF
- While there are still unprocessed bytes in infile:

  - Read at most k−1 bytes in from infile
  - let j be the number of bytes actually read
  - Place the read bytes into the allocated block starting from index 1 so as to not overwrite the 0xFF
  - Use mpz_import() to convert the read bytes
  - Encrypt m with ss_encrypt()
  - write the encrypted number to outfile as a hexstring followed by a trailing newline.

Create a function ss_decrypt which performs SS decryption, computing message m by decrypting ciphertext c using private key d and public modulus n. The parameters include m, c, d, and pq

- Call the pow_mod function with m, c, d, pq passed in to solve for m = c^d (mod pq)

Create a function ss_decrypt_file which decrypts the contents of infile, writing the decrypted contents to outfile. The parameters include infile, outfile, n pq, and d

- Dynamically allocate an array that can hold k = [(log2 (p n) −1)/8] bytes
- Iterating over the lines in infile:
  - Scan in a hexstring, saving the hexstring as a mpz_t c.
  - Using mpz_export(), convert c back into bytes, storing them in the allocated block.
  - Let j be the number of bytes actually converted.
  - Set the order parameter of mpz_export() to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter
  - Write out j−1 bytes starting from index 1 of the block to outfile.

<u>keygen.c</u>

Include all appropriate header files required

1. Parse command-line options using getopt().

2. Open the public and private key files using fopen(). Print a helpful error and exit the program in the event of failure.
3. Using fchmod() and fileno(), make sure that the private key file permissions are set to 0600, indicating read and write permissions for the user, and no permissions for anyone else.
4. Initialize the random state using randstate_init(), using the set seed.
5. Make the public and private keys using ss_make_pub() and ss_make_priv(), respectively.
6. Get the current user's name as a string. You will want to use getenv().
7. Write the computed public and private key to their respective files.
8. If verbose output is enabled print the following, each with a trailing newline, in order:
    (a) username
    (b) the signature s
    (c) the first large prime p
    (d) the second large prime q
    (e) the public key n
    (f) the private exponent d
    (g) the private modulus pq
    All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference key generator program for an example.
9. Close the public and private key files, clear the random state with randstate_clear(), and clear any mpz_t variables you may have used.

<u>encrypt.c</u>

Include all appropriate header files required

1. Parse command-line options using getopt().
2. Open the public key file using fopen(). Print a helpful error and exit the program in the event of failure.
3. Read the public key from the opened public key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:

    (a) username
    (b) the public key n
    All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference encryptor program for an example.
5. Encrypt the file using ss_encrypt_file().
6. Close the public key file and clear any mpz_t variables you have used.

<u>decrypt.c</u>

Include all appropriate header files required

1. Parse command-line options using getopt().
2. Open the private key file using fopen(). Print a helpful error and exit the program in the event of failure.
3. Read the private key from the opened private key file.
4. If verbose output is enabled print the following, each with a trailing newline, in order:
    (a) the private modulus pq
    (b) the private key d
    Both these values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference decryptor program for an example.
5. Decrypt the file using ss_decrypt_file().
6. Close the private key file and clear any mpz_t variables you have used.