Krisha Sharma

CSE13s

Winter 2022 Long

02/04/2022

Assignment 5 DESIGN.pdf

Description of Program:

The main task in Assignment 5: Public Key Cryptography is to create three programs:

1. A key generator: keygen
2. A encryptor: encrypt
3. A decryptor: decrypt

The keygen program will be in charge of key generation, producing the RSA public and private key pairs. The encrypt program will encrypt files using a public key, and the decrypt program will decrypt the encrypted files using the corresponding private key.

It is necessary to implement two libraries and a random state module that will be used in each of the programs. One of the libraries will hold functions that are related to the mathematics behind RSA, and the other will contain the implementations of routines for RSA. It is also necessary to learn to use the GNU multiple precision arithmetic library.

Files to be included in directory "asgn5":

- `decrypt.c`
    - This contains the implementation and `main()` function for the decrypt program
- `encrypt.c`
    - This contains the implementation and `main()` function for the encrypt program

- `keygen.c`
  - This contains the implementation and `main()` function for the keygen program
- `numtheory.c`
  - This contains the implementation of the number theory functions
- `randstate.c`
  - This contains the implementation of the random state interface for the RSA library and number theory functions
- `randstate.h`
  - This specifies the interface for initializing and clearing the random state
- `rsa.c`
  - This contains the implementation of the RSA library
- `rsa.h`
  - This specifies the interface for the RSA library
- Makefile
  - `CC = clang` must be specified
  - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified
  - `pkg-config` to locate compilation and include flags for the GMP library must be used.
  - `make` must build the `encrypt, decrypt,` and `keygen` executable, as it should `make all.`
  - `make decrypt` should build only the `decrypt` program.
  - `make encrypt` should build only the `encrypt` program.

        - `make keygen` should build only the `keygen` program.
        - `make clean` must remove all files that are compiler generated.
        - `make format` should format all the source code, including the header files
- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - Describes design for the program thoroughly with pseudocode and visualizations.

General Notes:

- Assignment 5: Public Key Cryptography is focused around RSA algorithms.
- We are learning this in order to create an RSA key pair, similar to what ssh does when prompted by ssh keygen.
    - To create an RSA key pair (mathematics of RSA):
        - 1. Choose two large prime numbers p and q
        - 2. Compute the product of the two, aka public modulus, $n = p \cdot q$
        - 3. Compute d and e, where $d \cdot e \equiv 1 \; mod(\varphi(n))$
            - $\varphi(n) = \varphi(p) \cdot \varphi(q)$

              $= (p - 1) \cdot (q - 1)$
            - $\varphi(p)$ = # of numbers < p that are coprime w/ p
                - coprime meaning that if you do $gcd(x, p) = 1$
            - In this case d and e are multiplicative inverses (modular inverses) in a modular ring

                - In order to guarantee that there is a modulo inverse of e, you need to make sure that the $gcd(e, \varphi(n)) = 1$
                    - This also guarantees an inverse d
    - To encrypt and decrypt a message:
        - encryption $c = m^e(mod(n))$
        - decryption $m = c^d(mod(n))$
            - In this case m is some message
                - For example: given “abc” which is 3 bytes, when converted into its ASCII value it becomes [97, 98, 99]. That then becomes some long binary (1100010…. …… ….) that is evaluated as a single number, which is what m is, that is then raised to e, becoming $m^e(mod(n))$.
            - n is the public modulus
        - Rather than using φ, during this assignment we are instead using lambda λ, which is Carmichale’s quotient
    - p and q are pseudo randomly generated, same with e
    - Have some external global variable that u can use across all your files

<u>`randstate.c` functions:</u>

`randstate_init()` description & pseudocode:

The `randstae_init` function aims to initialize a global random state with a Mersenne Twister algorithm. In this function, I use `seed` as the random seed, and then call `srandom(seed)`, using seed as well. Afterwards, I call the `gmp_randinit_mt()` and `gmp_randseed_ui` functions with state and seed as needed.

```
void randstate_init(uint64_t seed) {
     initializes a global random state named state with a
     Mersenne Twister algorithm;
     using seed as the random seed;
     call srandom() using this seed as well;
     call gmp_randinit_mt() and gmp_randseed_ui();
     should be two or three lines MAX;
}
```

`randstate_clear()` description & pseudocode:

The `randstate_clear` function in randstate.c aims to call and free all the memory that was used by the initialized global random state. This function only needs a single call.

```
void randstate_clear(void) {
     calls and frees all memory used by the initialized;
     global random state, state;
     should be a single call to gmp_randclear();
     should be two or three lines MAX
}
```

<u>`numtheory.c` functions:</u>

`power_mod()` description & pseudocode:

The power_mod function computes modular exponentiation. Modular exponentiation is the remainder when an integer b (the base) is raised to the power e (the exponent), and divided by a positive integer m (the modulus). In my function below, I have implemented modular exponentiation by squaring as shown in the assignment document, and then I reduce my results by multiplying them by modulo n, after each operation. By doing this, I am ensuring that the numbers produced will not be excessively large in size.

```
void power_mod() {
     initialize v at 1;
     initialize p as a;
     while d > 0 {
          if d is odd;
               v = (v * p) mod(n);
          p = (p * p) mod(n);
          d = d / 2;
     }
     return v;
}
```

`is_prime()` description & pseudocode:

The is_prime function is the implementation of the Miller-Rabins primality test which checks the primality of a large number. When coding this function be sure to make clones of all the parameterized variables. This is so any original values that have been fed into the function are not edited or changed with the computations made. This is extremely important as we may need to reference the original values later on in the code. The general implementation of the is_prime function I begin by checking if my r value

```
bool is_prime() {
     if r is odd
          n - 1 = 2^s(r);
     for (i = 1 to k)
          choose random a ε {2, 3,..., n - 2}
          y = pow_mod(a, r, n);
          if y does not equal 1 & y does not equal n - 1
```

```
                    j = 1;
                    while j <= s - 1 and y does not equal n - 1
                         y = pow_mod(y, 2, n);
                         if y == 1
                              return false;
                         j = j + 1;
                    if y does not equal n - 1
                         return false;
          return true;
     }
```

`make_prime()` description & pseudocode:

The make_prime function in assignment 5 aims to generate a new prime number stored in p. It is specified in the assignment document that the generated prime should be at least bits number of nits long, and that the primality of the generated number should be tested using the is_prime() function, with iters number of iteration.

```
     void make_prime() {
          initialize a var with value of 2^bits use mpz_pow_ui()
          while loop that iterates as long as var1 isn't a prime
          integer;
               call mpz_urandomb(var1, state, bits);
               equate var1 to var1 + var;
          clear all mpz_t variables used;
          return;
     }
```

`gcd()` description & pseudocode:

The gcd function implemented in assignment 5, computes the greatest common divisor of two values a and b, and then stores the computed divisor in d. The function begins by initializing all variables and creating clones of a and b, so that any original values that have been fed into the function are not edited or changed with the computations made. The function uses a while loop to compute the gcd, so long as b does not equal zero.

```
void gcd() {
    initialize all vars;
    mpz_inits(all vars, NULL);
    while b does not equal 0 {
        set t to b;
        set b to a (mod b);
        set a to t;
    }
    mpz_clears(all vars, NULL);
    return a;
}
```

`mod_inverse()`* *description & pseudocode:

The mod_inverse function in assignment 5, computes the inverse i of modulo n. The assignment document specifies that if the inverse cannot be found, i must be set to 0. The mod_inverse function requires parallel assignments, which is a concept that C does not support and thus, auxiliary variables are created in order to fake it.

```
mod_inverse() {
    (r,r_prime) ← (n,a)
    (t,t_prime) ← (0,1)
    while r_prime does not equal 0 {
        q ← the floor division of r and r_prime
        (r,r_prime ) ← (rprime ,r−q×r_prime)
        (t,t_prime) ← (tprime,t−q×t_prime)
    }
    if r is greater than one
        do not return an inverse
    if t is less than zero
```

```
          t ← t+n
     return t
}
```

<u>`rsa.c` functions:</u>

***`rsa_make_pub()`* description & pseudocode:**

The rsa_make_pub function, implemented in rsa.c, creates parts of a new RSA public key. The generates the following, two large primes, p and q, as well as their product n, and the public exponent e.

```
void rsa_make_pub() {
     make a random number in the range of [nbits/4,
     (3xnbits)/4) and begin generating the bits;
     calculate the remainder of the bits for qbits;
     use make_prime to generate a large prime number of
     pbits length;
     use make-prime to generate a large prime number of
     qbits length;
     calculate the gcd(p - 1, q - 1);
     multiply (p - 1) * (q - 1);
     divide the absolute value of (p - 1) * (q - 1) with
     the gcd of (p - 1) * (q - 1);
     while the exponent is not coprime {
          generate a random number of nbits
          compute the gcd of each random number and
          lambda(n);
     }
     clear all vars using mpz_clears(all vars, NULL);
     return;
}
```

`rsa_write_pub()` *description & pseudocode:*

The function rsa_write_pub, writes a public RSA key to pbfile in the specified format of n, e, s, and then the username. Each of these are supposed to be written with a trailing newline as specified in the assignment document. The values of n, e, and s should be written as hexstrings rather than integers. Using %Zx will specify that the values should be printed out as hexstrings to pbfile.

```
void rsa_write_pub() {
     gmp_fprintf(pbfile, %”%Zx\n‟, n);
     gmp_fprintf(pbfile, %”%Zx\n‟, e);
     gmp_fprintf(pbfile, %”%Zx\n‟, s);
     gmp_fprintf(pbfile, %”%s\n‟, username);
}
```

`rsa_read_pub()` *description & pseudocode:*

The function rsa_read_pub, reads a public RSA key from pbfile. The format should be n, e, s, then username, each of which should have been written with a trailing newline. The values of n, e, and s should have been written as hexstrings. This can be specified using %Zx

```
void rsa_read_pub() {
     gmp_fscanf(pbfile, %”%Zx\n‟, n);
     gmp_fscanf(pbfile, %”%Zx\n‟, e);
     gmp_fscanf(pbfile, %”%Zx\n‟, s);
     gmp_fscanf(pbfile, %”%s\n‟, username);
}
```

`rsa_make_priv()` *description & pseudocode:*

The function rsa_make_priv creates a new RSA private key d, given the primes p and q as well as the public exponent e. To compute d, it is necessary to take the inverse of e modulo $\lambda(n) = \mathrm{lcm}(p - 1, q - 1)$.

```
void rsa_make_priv() {
     initialize all vars needed;
     mpz_inits(all vars, NULL);
     set a var for p-1, mpz_sub_ui(p-1 var, p, 1);
     set a var for q-1, mpz_sub_ui(q-1 var, q, 1);
     compute the gcd of (p-1) & (q-1), gcd(gcd pq var, p-1
     var, q-1 var);
     set a var for (p-1)*(q-1), mpz_mul(var for p-1*q-1,
     p-1 var, q-1 var);
     set a var for the lcm
     abs((p-1)*(q-1))/gcd((p-1)*(q-1)), mpz_fdiv_q(var for
     lcm, p-1*q-1 var, gcd pq var);
     compute the mod_inverse and set it to a var,(d, e, var
     for lcm);
     mpz_clears(all vars, NULL);
}
```

rsa_write_priv() description & pseudocode:

The function rsa_write_priv, writes a private RSA key to pvfile. The format of a private key should be n then d, both of which are written with a trailing newline. Both of these values should also be written as hex strings.

```
void rsa_write_priv() {
     gmp_fprintf(pvfile, "%Zx\n", n);
     gmp_fprintf(pvfile, "%Zx\n", d);
}
```

`rsa_read_priv()` description & pseudocode:

The function rsa_read_priv, reads a private RSA key from pvfile. The format of the private key should be n and then d, both of which should have been written with a trailing new line. Both of these values should have been written as hex strings.

```
void rsa_read_priv() {
     gmp_fscanf(pvfile, "%Zx\n", n);
     gmp_fscanf(pvfile, "%Zx\n", d);
}
```

`rsa_encrypt()` description & pseudocode:

The function rsa_encrypt, performs RSA encryption, computing ciphertext c by encrypting message m using the public modulus exponent e as modulus n. Encryption with RSA is defined as s E(m) = c = me (mod n).

```
void rsa_encrypt() {
     use the pow_mod() function to perform rsa encryption
     E(n)= c = m^e (mod n), pow_mod(out, base, exponent,
     modulus)
}
```

`rsa_encrypt_file()` description & pseudocode:

The function rsa_encrypt_file encrypts the contents of infile, writing the encrypted contents to outfile. The data inside the file should be encrypted in blocks. Furthermore, the value of a block cannot be 0, E(0) ≡ 0 ≡ 0 e (mod n), and the value of a block cannot be 1 either, E(1) ≡ 1 ≡ 1 e (mod n). Because of these restrictions it is necessary to prepend an additional byte to the front of the block that we want to encrypt, The value of the prepended byte must be set to 0xFF.

```
void  rsa_encrypt_file() {
        initialize all vars necessary;
        mpz_intis(all vars, NULL);
        store log(n)-1 in a var, mpz_sub_ui(log(n)-1 var,
        log(n), 1);
        divide log(n) by 8 and store value in a var,
        mpz_fdiv_q_ui(k, log(n), 8);
        dynamically allocate an array that can hold k bytes,
        block = (uint64_t *)calloc(k, sizeof(uint64_t));
        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;
             using mpz_import(), convert the read bytes into
             an mpz_t m, set the order parameter to 1, the
             endian parameter to 1, and the nails parameter to
             0;
        }
        mpz_clears(all vars, NULL);
   }
```

`rsa_decrypt()` description & pseudocode:

The function rsa_decrypt performs RSA decryption, computing message m by decrypting ciphertext c using private key d and public modulus n. Decryption with RSA is defined as s D(c) = m = c d (mod n).

```
   void rsa_decrypt() {
        use the pow_mod() function to perform rsa decryption
        D(n)= m = c^d (mod n), pow_mod(out, base, exponent,
        modulus);
   }
```

***`rsa_decrypt_file()`* description & pseudocode:**

The function rsa_decrypt_file decrypts the contents of infile, writing the decrypted contents to outfile. The data in infile should be decrypted in blocks to mirror the way rsa_encrypt_file encrypts blocks.

```
void  rsa_decrypt_file() {
     initialize all vars necessary;
     mpz_intis(all vars, NULL);
     store log(n)-1 in a var, mpz_sub_ui(log(n)-1 var,
     log(n), 1);
     divide log(n) by 8 and store value in a var,
     mpz_fdiv_q_ui(k, log(n), 8);
     dynamically allocate an array that can hold k bytes,
     block = (uint64_t *)calloc(k, sizeof(uint64_t));
     while there are still unprocessed bytes in infile {
          scan a hexstring and save it as a mpz_t c;
          use mpz_export() to convert c back into bytes,
          set the order parameter to 1 for the most
          significant word, the endian parameter is also 1,
          the nails parameter is 0;
          write out j - 1 bytes starting from index 1 of
          the block to outfile;
     }
     mpz_clears(all vars, NULL);
}
```

***`rsa_sign()`* description & pseudocode:**

The function rsa_sign performs RSA signing producing the signature s by signing message m using a private key d and public modulus n. Signing with RSA is defined as S(m) = s = md (mod n).

```
void rsa_sign() {
     use the pow_mod function to perform rsa signing S(m) =
     s = m^d (mod n), pow_mod(out, base, exponent,
     modulus);
}
```

***`rsa_verify()`* description & pseudocode:**

The function rsa_verify performs RSA verification, returning true if the signature s is verified and false if otherwise. Verification is the inverse of signing. Let t = V(s) = s e (mod n). The signature is verified if and only if the t is the same as the expected message m.

```
void rsa_verify() {
     initialize all vars necessary;
     mpz_inits(all vars, NULL);
     pow_mod(out, base, exponent, modulus);
     if (mpz_cmp(t, m,) == 0) {
          if the signature is verified return true;
     }
     else {
          return false otherwise;
     }
     mpz_clears(all vars, NULL);
}
```

Error Handling:

- I ran into an error while trying to code my make_prime where when I was computing s-1 was outside the necessary, therefore when I would test, I would run into an infinite loop because when the program iterated through the loop the calculations were not subtracting and thus not computing properly. I went to Brian Mak's tutoring session 02/14/2022 for help finding the error.

Citations:

- I watched the yuja recording of Eugene's section on 02/04/2022 where he went over how to start assignment 5.
    - Eugene is cited for the pseudocode implementation of pow_mod and is_prime. During his section on 02/04/2022 he went into detail about how to implement both these functions while going through the arithmetic for them.
    - Eugene is also cited in numtheory.c make_prime specifically for the idea of adding 2^bits + the random number generated from `urandomb()` that caps at the user input of bits given.
    - Eugene is cited in rsa_make_pub for explaining how to generate a random number in the range [nbits/4, (3×nbits)/4).
    - Eugene is cited in keygen.c for explaining how to properly implement `fchmod()` and `fileno()`
- I attended Audrey Ostrom's tutoring section on 02/09/2022 and during it she helped me understand where to begin with the program and how to implement randstate.c.
    - Audrey provided me with pseudocode for randstate_clear and randstate_init during the section

- I attended Omar's office hours on 02/10/2022 and while I was in it he went over how to properly use commands from the gmp library as well as how to format the Makefile for this assignment.
- I attended Brian's tutoring session on 02/14/2022 for help with an infinite loop in my make_prime function as well as guidance on how to best start rsa_make_pub.
- I attended Mile's tutoring session on 02/15/2022 for help with properly printing out the `username[]` to pbfile and pvfile.
- I attended Ben's tutoring session on 02/15/2022 for help with understanding the while loop in `rsa_encrypt_file().`
- I attended Brian's tutoring session on 02/16/2022 for help with my `rsa_encrypt_file().` I was running into an issue where my encrypt was not working properly because the end of the file was not being reached. Brain pointed out that my variables were out of order and thus the while loop was not quitting properly.
- I attended Audrey's tutoring session on 02/16/2022 for help with testing my encrypt.c and decrypt.c files. During the session Audrey advised me to test my encrypt.c against the decrypt.c file in the resources repo to make sure that it was correct, and then to test my decrypt against my correct encrypt.