A Fun-key Program

Angela Shen

February 2023

1 Introduction

To implement Schmidt Samoa encryption, there needs to be a way to generate keys public and private and then use these keys to encrypt and decrypt. These are implemented as keygen.c, encrypt.c, and decrypt.c using randstate.c, numtheory.c and ss.c.

2 randstate

randstate functions are called to initialize the seed for generating random numbers.

void randstate_init(uint64_t seed): Includes calls to srandom(), which sets the seed based on the parameter seed, gmp_randinit_init, which initializes state, and gmp_randseed_ui, which defines the state.

void randstate_clear(void): Contains a call to gmp_randclear() to clear the state.

3 numtheory

numtheory.c implements the following functions:

void pow_mod(mpz_t o, const mpz_t a, const mpz_t d, const mpz_t n): (Based on the pseudocode in assignment 5 spec) Set a variable equal to the base, another equal to the exponent, and another variable v equal to the 1. Create a while loop that continues while the exponent is greater than 0. Since any positive integer can be represented by powers of 2, check for the parity of the exponent. If the exponent is odd, multiply v by the current value of the base and mod by modulus, since this may produce a very large number. Outside of the if statement, multiply the base by itself to square its value (powers of two) and mod by modulus, since this might also have a very large number as a result. Do floor division on the the exponent, and then repeat the while loop. When the loop terminates, set parameter o equal to the final value of v and then clear initialized variables.

bool is_prime(const mpz_t n, uint64_t iters): (Based on the pseudocode in assignment 5 spec) Use the Miller Rabin test to check if n is prime by finding an int r where n - 1 = 2^sr and r is odd. Creating a for loop starting at int i = 0 and terminating when i equals the number of iters. Then, choose a random integer a that is less than n - 2 and greater than 2. One of the conditions to test if a number is composite is if a^r is not equal to 1 or n - 1, so use pow_mod to create a variable equal to a^r mod n and check if it is not equal to 1 or n - 1. If this holds true, then check the other condition for a composite integer. Make a variable j = 1 and a while loop that terminates when j is greater than s - 1 and 2^sr is not equal to n - 1. Then test if 2^sr equals 1 within the while loop. If 2^sr equals 1 or n - 1, then return false (meaning the number is composite). If iters number of test are performed and never returns false, then the number is most likely prime.

Pseudocode to find int r where n - 1 = 2^sr and r is odd.

```
pow_two = 2
s = 0
r = 0
while (r is even)
    r = (n - 1) / pow_two
```

```
s <- s + 1
pow_two <- pow_two
```

This continuously divides n - 1 by powers of two until a power of two where r is odd is found.

void make_prime(mpz_t p, uint64_t bits, uint64_t iters): Generate a random number and test it against is_prime using iters amount of iterations until a prime is found.

void gcd(mpz_t g, const mpz_t a, const mpz_t b): Create a while loop that terminates if b equals 0. Assign b to a temp variable and then assign b to a mod b, then assign a to temp. Continue this loop until b equals 0 and return a (the greatest common denominator).

void mod_inverse(mpz_t o, const mpz_t a, const mpz_t n)): To calculate mod inverse, create variables r and r_prime and set them equal to a and n respectively. Create variables t and t_prime and then set them to 0 and 1 respectively. In a while loop, assign q to r / r_prime. Using auxiliary variables, swap r with r_prime and set r_prime equal to r_prime - q times r_prime. Do the same with t and t_prime, with t replacing r and t_prime replacing r_prime. If r is greater than 1, then there is no inverse for a mod n, and o is set to 0. If there is a mod inverse, it may be negative, so the value of n is added to the mod inverse t. If the mod inverse exists, set o equal to t.

4 ss

void ss_make_pub(mpz_t p, mpz_t q, mpz_t n, uint64_t nbits, uint64_t iters):

First, calculate the number of bits for p (a random number between nbits / 5 and (2 * nbits) / 5). Then generate a prime p, calculate the bits in p squared, and subtract the number of bits from nbits to find the number of bits for q. Then generate a prime q with at least qbits and check that q $\nmid$ p - 1 and p $\nmid$ q - 1.

```
pbits = rand(0, (nbits / 5) - 1)
pbits = pbits + (nbits / 5)
while (true)
    p = make_prime(pbits, iters)
    qbits = nbits - (bits in p^2)
    q = make_prime(qbits, iters)
    if q ∤ p - 1
        if p ∤ q - 1
            break
n = p^2 * q
```

void ss_write_pub(const mpz_t n, const char username[], FILE *pbfile): Use the gmp fprintf function to print the value of n (as a hexstring) and the username into pbfile.

void ss_read_pub(mpz_t n, char username[], FILE *pbfile): Use the gmp fscanf function to scan the value of hexstring n and the string username from pbfile.

void ss_make_priv(mpz_t d, mpz_t pq, const mpz_t p, const mpz_t q): Creates private key d by calculating the mod inverse of n mod λ(pq). Also sets pq equal to p * q. λ(pq) is equal to ((p - 1) (q - 1)) / gcd(p - 1, q - 1)

```
lambda = ((p - 1)*(q - 1)) / gcd(p - 1, q - 1)
d = mod_inverse(n, lambda)
pq = p * q
```

void ss_write_priv(const mpz_t pq, const mpz_t d, FILE *pvfile): Use the gmp fprintf function to print the value of pq and d into pvfile as hexstrings.

void ss_read_priv(mpz_t pq, mpz_t d, FILE *pvfile): Use the gmp fscanf function to read the hexstrings of pq and d from pvfile.

void ss_encrypt(mpz_t c, const mpz_t m, const mpz_t n): Calculates $m^n mod n$ and assigns to c.

void ss_encrypt_file(FILE *infile, FILE *outfile, const mpz_t n):

To encrypt the input from file, use blocks to ensure the the value is less than n. First, calculate the value of k = (log2($\sqrt{n}$) - 1) / 8 and dynamically allocate an array of k amount of uint8_t *. Then, set the byte at index 0 to 0xFF. Then, create a while loop. Use feof(infile) == 0 as the condition so that the loop will terminate when there are

no more bytes to read. Then use fread() to read bytes from infile and assign the number of bytes read to j. Then import j + 1 amount of bytes to a variable m, then encrypt m using the parameter n. Finally, print the encrypted bytes using gmp_fprintf.

```
k = (log2($\sqrt{n}$) - 1) / 8
j = 0
while (feof == 0)
    j = fread k - 1 amount of bytes from infile
    mpz_import j + 1 bytes from bytes to m
    ss_encrypt(m, n)
    gmp_fprintf encrypted key from ss_encrypt
```

void ss_decrypt(mpz_t m, const mpz_t c, const mpz_t d, const mpz_t pq): Calculates $c^d mod pq$ and assigns it to m.

void ss_decrypt_file(FILE *infile, FILE *outfile, const mpz_t d, const mpz_t pq):

Like encrypt, decrypt uses blocks when reading from infile. First, calculate the value of k = (log2($\sqrt{pq}$) - 1) / 8 (since pq must be greater than sqrt n) and dynamically allocate an array of k amount of uint8_t *. Then use gmp_fscanf to read from infile, then decrypt c back to m. Then use mpz_export to write to export m to bytes and use fwrite to write the message into the outfile.

```
k = (log2($\sqrt{pq}$) - 1) / 8
j = 0
while (true)
    gmp_fscanf encrypted message from infile
    if (feof)
        break
    m = ss_decrypt(c, d, pq)
    mpz_export m to bytes, j = the number of bytes exported
    fwrite j - 1 amount of bytes to the outfile
```

5 Other C files

keygen.c: Makes calls to ss_make_pub() and ss_make_priv() to generate public and private keys in different files.

encrypt.c: Accepts input from a file and writes the encrypted message into the outfile.

decrypt.c: Accepts input from a file and decrypts the message and writes the original message to outfile.