

Assignment 5 Design

Arnav Nepal

February 15, 2023

1 Introduction

In this assignment we will be using a public-key cryptography system to implement a cryptography program for encrypting and decrypting messages. More specifically, we will be implementing the Schmidt-Samoa (SS) algorithm for encryption and decryption. Our implementation of the algorithm will require us to be able to handle arbitrary precision integers. Towards that end, we will be making use of the GNU multi-precision arithmetic (GMP) library to handle large integers. The files we will be writing are:

- decrypt.c
- encrypt.c
- keygen.c
- numtheory.c
- randstate.c
- ss.c

2 Design and Psuedocode

The main programs we will be writing in this assignment (keygen, encrypt, and decrypt) require the usage of 2 libraries and a random state module. I will be using regular, pure C operators and arithmetic in my design, as GMP is hard to read and understand. However, it is easier to use if you just use it replace C operations and functions line by line with the GMP operations. All mentions of “mpz_t” in function paramters should be considered intgers.

2.1 randstate.c

The random state module will simply be used to initialize a global random “state” variable to use with GMP. It will seed both the GMP random function and the C library random function with the passed in seed.

```
func randstate_init(int seed):  
    seed gmp_randstate_ui  
    seed srandom()
```

```

    gmp_randinit_mt(state) //state is a global extern variable

func randstate_clear(void):
    clear state with gmp_randclear()

```

2.2 numtheory.c

This file will contain the library for mathematic functions that are required for implementing the SS algorithm. For this program, we were provided psuedocode, so my design will closely resemble it.

```

void gcd(mpz_t g, mpz_t a, mpz_t b);

void mod_inverse(mpz_t o, mpz_t a, mpz_t n);

//paramters: o is the output variable, a is the base that will
//be raised to exponent d, and n is the modulus
void pow_mod(mpz_t o, mpz_t a, mpz_t d, mpz_t n);
    int v = 1 //holds calculation results
    int p = a //p holds "a" to avoid changing "a"
    while exponent d > 0;
        if d is odd:
            v = (v * p) % n
            p = (p*p) % n
            d = floordiv(d/2)
    return v

```

For the primality testing function (the Miller-Rabin) test, we must evaluate the first step of the test: *write $n - 1 = 2^s r$ such that r is odd*. This equation can be rewritten as: $r = \frac{n-1}{2^s}$. This essentially means that we must divide $n - 1$ by 2 s times until it results in a positive number. We then hold on to the number of times required to reach an odd number as well as resultant odd number.

Secondly, we need to choose a random number a where $a \in \{2, 3, \dots, n - 1\}$. This is relatively simple. Since we need to exclude 0 and 1, we can just add 2 to the random number. Furthermore, gmp contains the function `mpz_urandomm(output, state, n)` which returns a random integer in the range $[0, n)$. Thus we need to just pass in $n-3$ and then add 2 to the output of the RNG to get a random number within the desired range. (the design uses $\%(n-4)$ due to how modulus in C works).

```

//in the SS implementation, the parity of the number will most likely
//be checked before being passed into is_prime. Thus, for now, assume
//that all arguments passed in are at least odd
bool is_prime(mpz_t n, uint64_t iters);

```

```

int r = (n-1)/2
int s = 1 //first division done above. 2 divides all evens at least once
while (is_odd(r) == false){
    r = r/2
    s++;
}

```

```

int a
for int i from 1 to iters:
    a = random()%(n-4)
    a += 2;
    int y = power_mod(a,r,n)
    if y != 1 AND y != n - 1:
        int j = 1
        while j <= (s-1) AND y != (n-1):
            y = power_mod(y,2,n)
            if y == 1:
                return false
            j +=1
        if y != (n-1)
            return false
return true;

```

```

//this function generate a random number, then uses a while loop
//to check and regenerate random numbers until a prime is found
void make_prime(mpz_t p, uint64_t bits, uint64_t iters);
    use mpz_urandomb(output, state, bit_cnt) to generate a random number
    bit_cnt should be set to the passed in "bits" argument
    while (is_prime(output, iters)):
        mpz_urandomb(output, state, bits)

```

The next 2 functions in the number theory library deal with modular inverses. However, to implement a modular inverse function we first need to create a function to find the greatest common divisor (GCD). Furthermore, mod-inverse as shown on the assignment spec contains a lot of parallel assignments. Since C cannot do parallel assignments, temporary variables will be needed to do the math.

```

//this function finds the greatest common divisor of a and b, and
//stores the result in d
func GCD(output, a, b):
    int t
    while b != 0;
        t = b

```

```

        b = a % b
        a = t
    return a

//compute the inverse i of a modulo n
func mod-inverse(inverse i, int a, modulo n):
    int r = n
    int r' = a
    int t = 0
    int t' = 1
    int q;

    //each parallel assignment will be spaced out to clarify the code
    while r' != 0:
        q = r/r'

        temp_r = r
        r = r'
        r' = temp_r - (q * r')

        temp_t = t
        t = t'
        t' = temp_t - (q * t')
    if r > 1:
        i = 0
    if t < 0:
        t = t + n
    return t

```