Assignment 5 Design Document

randstate.c

randstate_init(seed):
    call gmp and srandom functions to initiliaze random number generation.
randstate_clear():
    call gmp_randclear to clear the random state.

numtheory.c

```
gcd(d, a, b):
    while b != 0:
        temp = b
        b = a % b
        a = temp
    d = a

mod_inverse(i, a, n):
    r = n
    r' = a
    t = 0
    t' = 1
    while r' != 0:
        q = r/r'
        r = r'
        r' = r - q*r'
        t = t'
        t' = t - q*t'
    if r > 1: return no inverse
    if t < 0: t = t+n
    i = t
pow_mod(out, base, exponent, modulus):
    v = 1
    p = a
    while exponent > 0:
        if exponent is odd:
            v = v*p % modulus
        p = p^2 % modulus
        d = d/2
    out = v
is_prime(n, iters):
    write n−1= 2srsuch that ris odd
```

```
        for i from 1 to k:
                a = random number from 2 to (n-2)
                y;
                pow_mod(y, a, r, n)
                if y != 1 and y != n-1:
                        j = 1
                        while j <= s-1 and y != n-1:
                                pow_mod(y, y, 2, n)
                                if y == 1: return 0
                                j = j+1
                        if y != n-1: return 0
        return 1
make_prime(p, bits, iters):
        p = (2^(bits/2))^2 + 1
        while (is_prime(p, iters)):
                bits += 2
                p = (2^(bits/2))^2 + 1
```

rsa.c

```
rsa_make_pub(p, q, n, r, nbits, iters):
        random_p = random number from nbits/4 to (3*nbits)/4
        random_q = 2^nbits - random_p
        make_prime(p, random_p, iters)
        make_prime(q, random_q, iters)
        d;
        gcd(d, p-1, q-1)
        n = |(p-1)(q-1)|/d
        while 1:
                rand_num = generate a random number from 2^nbits to 2^(nbits+1)
`               gcd(d, rand_num, n)
                if d == 1:
                        e = d
                        break
rsa_write_pub(n, e, s, username, pbfile):
        write to pbfile hex strings of n, e, and s and then username
rsa_read_oub(n, e, s, username, pbfile):
        Read pbfile and separate by newline character.
        n = first part of file
        e = second part of file
        s = third part of file
        username = end of file
rsa_make_priv(d, e, p, q):
        t;
```

```
        gcd(t, p-1, q-1)
        d = 1/e % |(p-1)*(q-1)|/t
rsa_write_priv(n, d, pvfile):
        write to pvfile hex strings of n and d
rsa_read_priv(n, d, pvfile):
        Read pvfile and separate by newline character.
        n = first part of file
        d = second part of file
rsa_encrypt(c, m, e, n):
        c = m^e % n
rsa_encrypt_file(infile, outfile, n, e):
        k = (log_2(n)-1)/8
        make space for an array that can hold k bytes
        set the first byte of the array to 0xFF
        while 1:
                j = k-1
                if there are less than k-1 bytes in infile: j = number of bytes in infile
                read k-1 bytes from infile
                place bytes into array starting at index 1
                mpz_tm m[length of array]
                for the items in the array:
                        m[index in array] = mpz_import(item)
                encrypt m with rsa_encrypt
                write to outfile
                if j != k-1: break
rsa_decrypt(m, c, d, n):
        m = c^d % n
rsa_decrypt_file(infile, outfile, n, d):
        k = (log_2(n)-1)/8
        make space for an array that can hold k bytes
        array
        while 1:
                split hexstrings by newlines
                mpz_t c = next hexstring
                array add mpz_export(c)
                write j-1 bytes starting from index 1 to outfile
                break when reaching end of file
rsa_sign(s, m, d, n):
        s = m^d % n
rsa_verify(m, s, e, n):
        m = s^e % n
```

Key Generator

program with these 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: rsa.pub).
• -d pvfile: specifies the private key file (default: rsa.priv).
• -s: specifies the random seed for the random state initialization (default: the seconds since the UN I X epoch, given by time(NULL)).
• -v: enables verbose output.
• -h: displays program synopsis and usage.
open public and private key files
set permissions for private key file to 0600
run randstate_init(seed)
use rsa_make_pub() and rsa_make_priv() to make public and private keys
get current users name as string with getenv()
convert username to mpz_t
write public and private keys to their files
if verbose : print
(a) username
(b) the signature s
(c) the first large prime p
(d) the second large prime q
(e) the public modulus n
(f) the public exponent e
(g) the private key d
close files and clear random state and mpz_t variables

Encryptor

program with these 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: rsa.pub).
• -v: enables verbose output.
• -h: displays program synopsis and usage.
open public key file
read public key from file
if verbose: print
(a) username
(b) the signature s
(c) the public modulus n
(d) the public exponent e
convert the username into an mp_t

verify signature with rsa_verify()
encrypt file
close files and clear mpz_t variables

Decryptor

program with these 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 private key (default: rsa.priv).
• -v: enables verbose output.
• -h: displays program synopsis and usage.
open private key file
read private key from file
if verbose: print
(a) the public modulus n
(b) the private key e
decrypt file
close files and clear mpz_t variables