# Assignment 6
# Public Key Cryptography

Mikayla Win

Fall 2021

## 1 Introduction

In this assignment, a keygen program, encrypt program, and decrypt program will be implemented. The keygen program generates a key, produces RSA public and private key pairs. The RSA algorithm is the basis of a cryptosystem which enables public key encryption. The encrypt program encrypts files using a public key. The decrypt program decrypts encrypted files using a corresponding private key.

## 2 Definitions

**public key**: a public key is a cryptographic key that is used by anyone to encrypt messages intended for a particular recipient.
**private key**: a private key is a ccryptographic key that is use to decrypt, sometimes encrypt, messages
**RSA Algorithm**: This algorithm makes use of both public and private keys with the idea that the encrypted message can only be decrypted by a certain private key.

## 3 Files in the "asgn6" Directory

- encrypt.c : Implementation of the encrypt program. This program, as explaind in the introduction, encrypts files using a public key.
- decrypt.c : Implementation of the decrypt program.
- keygen.c : Implementation of the keygen program.
- numtheory.c : Implementation of the number theory functions.
- randstate.c : Implementation of the random state interface for the RSA library amd number theory functions.
- rsa.c : Implementation of the RSA library.

- numtheory.h : Header file that contains the interface of the number theory implementation.
- randstate.h : Header file that contains the interface of the random state interface.
- rsa.h : Header file contains the interface of the RSA library.
- Makefile : Used to format the source code file so it will run correctly as well as building the program correctly. Can also be used to clean the directory by removing the files generated by the compiler.
- README.md : A markdown file that contains a brief description of the program and directions for how to build and run the program. Also contains any reported errors by the command scan-build. As well as an explanation for said errors.
- DESIGN.pdf : Provides a more in depth description of the program as well as an explanation for how the program functions. Essentially the design of the program.

## 4 Pseudocode and Explanations

All images of pseudocode were provided by the assignment document.

### decrypt.c

```
define main function:
    create infile with default stdin
    create outfile with default stdout
    create pvfile with default rsa.priv

    create booleans for printing verbose and printing help

    use getopt
        -i : specifies the input file to encrypt
            fopen(optarg, "r")
        # default: stdin
        -o : specifies the output file to encrypt
            fopen()
        # default : stdout
        -n : specifies the file containing the public key
        # default : rsa.priv
        -v : enables verbose output
        -h : help message
    call rsa_read_priv()
    if verbose == true
        print public modulus n
        print key e
        # these values should be printed in decimal
    call rsa_decrypt_file()
    close private key file
    clear all mpz_t variables
```

**encrypt.c**

```
define main function:
    create infile with default stdin
    create outfile with default stdout
    create pvfile with default rsa.pub

    create booleans for printing verbose and printing help

    use getopt
        -i : specifies the input file to encrypt
            fopen(optarg, "r")
        # default: stdin
        -o : specifies the output file to encrypt
            fopen()
        # default : stdout
        -n : specifies the file containing the public key
        # default : rsa.pub
        -v : enables verbose output
        -h : help message
    call rsa_read_pub()
    if verbose == true
        print username
        print signature s
        print public modulus n
        print public exponent e
    # mpz_t valuse should be printed in decimal
    expected = convert username to mpz_t
    # expected value of the verified signature
    call rsa_verify()
    if expected != rsa_verify()
        error
        exit
    call rsa_encrypt_file()
    close public key file
    clear all mpz_t variables
```

**keygen.c**

```
define main function:
    use getopt
```

```
        -b : specifies the minimum bits needed for the public modulus n
        -i : specifies the number of 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
        # default : the seconds since the UNIX epoch, given by time(NULL)
        -v : print verbose
        -h : help message
    fopen() public and private key
    if failed:
        print error message
        exit()
    use fchmod() and fileno() to check private key permissions = 0600
    call randstate_init(seed)

    call rsa_make_pub()
    call rsa_make_priv()

    use getenv("USER")    # this gives the current user's name as a string
    convert username into mpz_t using mpz_set_str()
    # base 62
    call rsa_sign()
    call rsa_write_pub()
    call rsa_write_priv()
    if verbose == true
        print username
        print signature s
        print p
        print q
        print n
        print e
        print d
    # all mpz_t values should be printed as decimal
    close public and private key files
    call randstate_clear()
    clear all mpz_t variables
```

**numtheory.c**

```
POWER-MOD(a, d, n)
v ← 1
p ← a
while d > 0
    if ODD(d)
        v ← (v × p) mod n
    p ← (p × p) mod n
    d ← ⌊d/2⌋
return v
```

```
void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)
    v = 1
    p = a
    while d > 0
        if d % 2 > 1
            v *= p % n
        p = (p * p) % n
        d = floor(d / 2)
```

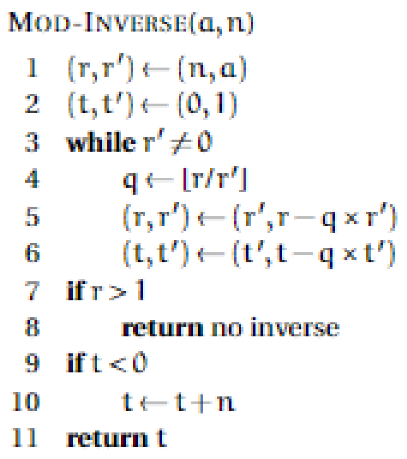
```
MILLER-RABIN(n, k)
write $n - 1 = 2^s r$ such that r is odd
for i ← 1 to k
    choose random $a \in \{2, 3, \ldots, n-2\}$
    y = POWER-MOD(a, r, n)
    if $y \neq 1$ and $y \neq n - 1$
        j ← 1
        while $j \leq s - 1$ and $y \neq n - 1$
            y ← POWER-MOD(y, 2, n)
            if y == 1
                return FALSE
            j ← j + 1
        if $y \neq n - 1$
            return FALSE
return TRUE
```

```
bool is_prime(mpz_t n, uint64_t iters)
    n = 2^sr + 1     # r must be odd
    # this uses a loop
    # compute s and r
    # r should end up as odd
    temp n = n - 1
    while (r is even)
        r = temp n / 2
        s += 1
        # s will increase with counts
        # loop will stop when r is odd
    if n % 2 == 0 or n == 0 or n == 1
        clear mpz
        return false
    for (1, iters)
        upper bound = n - 2
        a = (random % upper bound) + 2
        y = power_mod(a, r, n)
        if y != 1 and y != n - 1
            j = 1
            while j <= s - 1 and y != n - 1
                y = power_mod(y, 2, n)
                if y == 1
                    clear mpz
                    return false
                j += 1
            if y != n - 1
                clear mpz
                return false
    return true

void make_prime(mpz_t p, uint64_t bits, uint64_t iters)
    check = false
    while check == false
        call mpz_urandomb
        check = is_prime(p, iters)
```

```
GCD(a, b)
1  while $b \neq 0$
2      $t \leftarrow b$
3      $b \leftarrow a \bmod b$
4      $a \leftarrow t$
5  return a
```

```
void gcd(mpz_t d, mpz_t a, mpz_t b)
    temp b = b
    while b != 0
        t = b
        b = a % temp b
        a = t
    d = a
```



```
MOD-INVERSE(a, n)
(r, r') ← (n, a)
(t, t') ← (0, 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
return t
```

```
void mod_inverse(mpz_t i, mpz_t a, mpz_t n)
    r = n
    r_prime = a
    t = 0
    t_prime = 1
    temp_r = r
    temp_rprime = r_prime
    temp_t = t
    temp_tprime = t_prime
    while r_prime != 0
        q = temp_r / temp_rprime
        r = temp_rprime
        r_prime = temp_r - (q * temp_rprime)
        t = temp_tprime
        t_prime = temp_t - (q * temp_tprime)
    if r <= 1
        return t
    if t < 0
        t += n
    return t
```

**randstate.c**

```
void randstate_init(uint64_t)
    initialize state
    call gmp_randinit_mt()
    call gmp_randseed_ui()

void randstate_clear(void)
    call gmp_randclear()
    # this clears and frees all memory
```

**rsa.c**

```
void rsa_make_pub(mpz_t p, mpz_t q, mpz_t e, uint64_t nbits, uint64_t iters)
    p = make_prime(p, bits, iters)
    q = make_prime(q, bits, iters)
    n = p * q
    upperbound = (2 * nbits) / 4
    lowerbound = nbits / 4
    p_nbits = (random % upperbound) + lowerbound
    q_nbits = 1 - p_nbits
    totient = (p - 1) * (q - 1)
    divisor = 0
    while divisor == 1
        mpz_urandomb(coprime, state, nbits)
        # state : initialized in randstate_init()
        gcd(divisor, coprime, totient)

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile)
    gmp_fprintf(pbfile, "%Zx\n", n)
    fputs(pbfile, n)
    # do for e, s, username

void rsa_read_pub(mpz_t n, mpz_t e, mpz_t, s, char uusername[], FILE *pbfile)
    gmp_fscanf(pbfile, "%Zx\n", n)
    fgets(pbfile, n)
    # do for e, s, username

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q)
    d = (1 / e) % (p * q)

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile)
    gmp_fprintf(pvfile, "%Zx\n", n)
    fputs(pvfile, n)
    # do for d

void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile)
    gmp_fscanf(pvfile, "%Zx", n)
    fgets(pvfile, n)
    # do for n

void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n)
    call pow_mod(c, m, e, n)

void rsa_encrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t e)
    k = floor((log2(n) - 1) / 8)
    # mpz_sizeinbase(n, 2) -> use to find log base 2 of n
    BLOCK = allocate memory of an array that holds k bytes
    # this should be type (uint8_t *)
    # this will be the block
    set 0th bit = 0xFF
    while input != 0
        call fread to read the encrypt block
        if bytes read > 0
            mpz_import()    # converts the read bytes into mpz_t m
            # order of mpz_import: 1 most significant word, 1 for endian parameter
            # 0 for nails parameter
            call rsa_encrypt()
            call fprintf to outfile
    clear mpzs
    free BLOCK

void rsa_decrypt(mpz_t, mpz_t c, mpz_t d, mpz_t n)
    call pow_mod(m, c, d, n)

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d)
    k = floor((log(n) - 1) / 8)
    # mpz_sizeinbase(n, 2) -> use to find log base 2 of n
    BLOCK = allocate memory of an array that holds k bytes
    # this should be type (uint8_t *)
    index 0 = 0xFF
    while input != 0
        if c > 0            # c : content scanned from the input file
            call rsa_decrypt()
            mpz_export()    # converts the read bytes into mpz_t m
            # order of mpz_export: 1 most significant word, 1 for endian parameter
            # 0 for nails parameter
            call fwrite()
            # write out j - 1 bytes from [1, block] to the outfile
    clear mpzs
    free BLOCK

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n)
    call pow_mod(s, m, d, n)

void rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n)
    call pow_mod(t, s, e, n)
    if s is verified (t == m)
        return true
    else s isn't verified
        return false
```

## 5 Cite

- Makefile provided by TA Sloan during his in-person section on 11/10/21.
- Functions numtheory.c had psuedocode provided by the assignment document written by the Professor and TAs.
- During TA Eugene's in person sction on 11/18/21, he explained that we can use mpz-sizeinbase() to find log base 2 of something.
- During Tutor Miles' section on Friday 11/19/21, he explained how to implement feof() in the rsa-encrypt-file() and rsa-decrypt-file() while loop conditional. This function is used to check the end of file.

## 6 Error Handling

- If the user specifies a file that doesn't exist for the infile, outfile, pvfile, and pbfile, the program will have segmentation fault.