# Assignment 5 Design: Public Key Cryptography

Sep Nasiriany

Due: February 26, 2023

## 1 Description:

This program will simulate asymmetric Public-Public Encryption using the `gmp` library. What exactly is asymmetric Public key Cryptography? If Person A were to be the sender and Person B were to be the recipient, Person A would broadcast a message encrypted in Person B's public key, which everybody may or may not know about. The recipient and only that specific recipient (Person B) will be able to unlock it using their own private key, which makes them uniquely able to receive and decipher the message. Below is a diagram explaining this phenomenon:

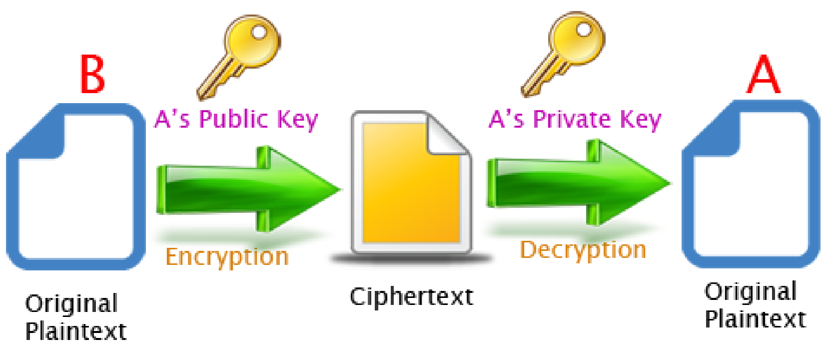


**Figure 1**
Diagram explaining Public-Private Key Encryption. Only Person A can receive the message from Person B because their private key is not known to outsiders. Image from
`https://community.ibm.com/community/user/ibmz-and-linuxone/blogs/subhasish-sarkar1/2020/06/23/understanding-the-rsa-asymmetric-encryption-system`

Knowing that, how is this program going to emulate that? Well it will be divided into 3 parts:

1. Generating a public and private key with `keygen`
2. Encrypting a message using a public key with `encrypt`
3. Decrypting a message using a private key with `decrypt`

There will be one C file that deals with initializing and clearing `gmp` random functions and that will be `randstate.c`. The two C files that do most of the algorithmic and mathematical computation are `numtheory.c` and `ss.c`. There contain be 3 C files that will contain a main function and they are: `keygen.c, encrypt.c, and decrypt.c`. Public and private keys are generated by `keygen.c`. There will be only one C file, `encrypt.c`, for encrypting using the public key and also only one C file, `decrypt.c` for decrypting using the private key. The purpose and the pseducode for each of the C files will be discussed in greater detail below.

# 2 `randstate.c`

This C file will be written to process random numbers that will later be used as public and private keys. This C file will use the `gmp.h` library because C does not natively provide arbitrary precision integers. There will be a global variable called `state` which will have memory allocated and freed by the `gmp` library functions. Let's go over the functions for it.

## 2.1 `void randstate_init(uint64_t seed)`

This function will have memory allocated to the global variable `state` using `calloc`. It will be using the Mersenne Twister algorithm to initialize `state`. Here is the following the function needs to do:

1. Call `gmp_randinit_mt()` function to intialize the state. Pass the `state` variable as the parameter.
2. Once initialized, call `gmp_randseed_ui()` in order to set the seed value into `state`.
3. Call the `srandom()` function and pass the seed into its parameter. This will set the seed of `random` to the same seed given to `gmp` random functions.

## 2.2 `void randstate_clear(void)`

This function will free all memory used by `state` by calling `gmp_randclear()`.

# 3 `numtheory.c`

This C file will be responsible for computing and using number theory that is used for the Schmidt-Samoa (SS) Algorithm. Here are the following functions for it:

### 3.1 `void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus)`

This function is responsible for modular exponentiation. This is critically important in encrypting plaintexts and decrypting ciphertexts. Here are the folliwng steps:

1. Create 3 `mpz_t` variables `v`, `p`, and `i`. Initialize them using `mpz_inits()`.
2. Set the value of `v` to the value of 1 using `mpz_setui`.
3. Set `p` equal to `a` and `i` equal to `d` using `mpz_set`.
4. Create a `while` loop with the condition that `i` is greater than 0 using `mpz_cmp_ui`.
   (a) Make sure if `d` is odd, `v` is equal to `v` * `p` mod `n`.
   (b) Make sure `p` is equal to $p^2$ mod `n`.
   (c) Make sure d is equal to half its current value.

### 3.2 `bool is_prime(mpz_t n, uint64_t iters)`

This function will conduct the Miller-Rabin primarily test to check whether `n` is is prime or not. This function is important to ensure all number being generated are prime numbers, it will be especially helpful for `make_prime()`. Here are the following steps:

1. Create an `if` condition that will check if the number is equal to 0 or 1. These are special cases where the Miller-Rabin primarily test cannot determine. Return `false` if the condition is satisfied.
2. Create an `if` condition that will check if the number is equal to 2 or 3. These are also special cases where the Miller-Rabin primarily test cannot determine, but they are prime numbers. Return `true` if the condition is satisfied.
3. Create a `mpz_t n_copy` that will be equal to `n`-1.
4. Create a `while` loop that will divide `n_copy` such that it will be odd.
5. Create a `for` loop that will iterate until `iters` number of times.
6. Inside the loop, it will check whether or not there are any values divisible to it using the modulus operator. If so, it will return `false`.
7. If the loop has completed, then by definition it has not found any divisible numbers, so it will return `true`.

### 3.3 `void make_prime(mpz_t p, mpz_t bits, mpz_t iters)`

This function will generate a new prime number in `p`. We will use bit-wise operators to make sure the number of bits is greater than or equal to `bits`. Here are the following steps:

1. Create an `mpz_t addition` and initialize it. This will be used to add to the value the random `mpz_t` returns.

2. Create a boolean `primeSatisfied` that will determine whether or not we got a prime.

3. If `bits` is equal to 1, we will make `addition` equal to $2^{\texttt{bits}}$. Otherwise, make `addition` equal to $2^{\texttt{bits-1}}$. The reason for this is because 1 or 2 bits offer a very limited range of prime numbers.

4. Create a `while` loop that will run until `primeSatisfied` is `true`.

   (a) Generate random numbers from 0 to $2^{\texttt{bits-1}}$ using `mpz_urandomb`.

   (b) Add `addition` to the value generated above.

### 3.4 `void gcd(mpz_t d, mpz_t a, mpz_t b)`

This function computes the greatest common divisor of a and b and stores it in d. Here are the following steps:

1. Create copies of `a` and `b`. These copies are what will be used below.

2. Run a `while` loop where it runs while `b` is not equal to 0.

   (a) Create an `mpz_t t` equal to `b`.

   (b) Make `b` equal to `a` mod `b`.

   (c) Make `a` equal to `t`.

3. After the loop is complete, return `a`.

### 3.5 `void mod_inverse(mpz_t i, mpz_t a, mpz_t n)`

This function computes the inverse `a` of modulo `n` and stores that value into `i`. This function will be necessary and crucial to making a private key. Here are the following steps:

1. Create copies of `a` and `n`.

2. Create variables `t` and `t_prime` and their values will be 0 and 1 respectively.

3. Create a `while` loop that iterates until the copy of `a` is equal to 0.

    (a) Make a variable called `q` that will hold the quotient of `n_copy` and `a_copy`.

    (b) Create a temporary variable that will hold the copies of `n_copy` and `t`.

    (c) Make sure to set the value of `n_copy` to `a_copy`.

    (d) Make sure to set the value of `t` equal to `t_prime`.

    (e) Set the value of `a_copy` to the temporary variable of `n_copy` subtracted from the quotient of `q` and `a_copy`.

    (f) Set the value of `t_prime` to the temporary variable of `t` subtracted from the quotient of `q` and `t_prime`.

4. After the loop finishes, check if `n_copy` is greater than 1. If so, set `i` to 0 and return.

5. If `t` is negative, make `t` equal to the sum `t` of `n`.

6. Set `i` equal to the value of `t`.

# 4 `ss.c`

This C file exists to apply all the math functions created in `numtheory.c` into the SS algorithm. This C file will do everything from generating public and private keys to encrypting them into files and decrypting them.

## 4.1 `void ss_make_pub(mpz_t p, mpz_t q, mpz_t n, uint64_t nbits, uint64_t iters)`

This function serves to generate the public key that will be used for encryption. There are two conditions to a public key `n` to be valid:

1. The $log2(\mathtt{n}) >= \mathtt{nbits}$

2. `p` and `q-1` cannot be divisible and `q` and `p-1` cannot be divisible.

Thus, in order to achieve these two goals, a `do-while` loop will be created. Inside the loop, the `random()` function will be used to generate the number of bits that will be generated for p in making a prime nummber. The number will be in the range of [`nbits`/5, (2 * `nbits`) / 5). Once that is done, the number of bits for `q` will just be ($\mathtt{nbits} - (2 * \mathtt{pbits})$). The function `make_prime()` will be called to generate the p and q prime numbers.

## 4.2 Write Functions: `void ss_write_pub()` `void ss_write_priv()`

These functions will call `gmp_fprintf()` in order to write the necessary contents into their respective files. It will use `"Zx"` in order to write the contents as a hexstring.

## 4.3 Read Functions: `void ss_read_pub()` `void ss_read_priv()`

These functions will call `gmp_fscanf()` in order to read the necessary contents and store them into the variables of the parameter passed. It will use `"Zx"` in order to read the contents as a hexstring.

## 4.4 `void ss_encrypt(mpz_t c, mpz_t m, mpz_t n)`

This function will create encrypt a message `m` by calling `pow_mod` to calculate $m^n$ mod `n` and store it in `c`.

## 4.5 `void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n)`

This function will encrypt the contents of `infile` and store it into `outfile`.

1. It will do so in blocks of size `k`. `k` is calculated by the following formula: $(log2(\sqrt{\mathtt{n}}) - 1)/8)$.
2. Allocate an array of type `uint8` dynamically using `calloc()` and make it of size `k` bytes.
3. Once `k` has been calculated, create a `do-while` loop that will use the function `fread()` to read the contents of `infile` in `k-1` byte chunks and store the result into `j`.
   (a) Make sure to include an `if` condition that will check if `j` is less than (k-1) or if `j` is equal to 0. If so make sure that it breaks, after calculating its binary data, encrypting, and writing to `outfile`.
   (b) Make sure to call `mpz_import()` to convert the message into binary data.
   (c) Call the `encrypt()` function to get the ciphertext version of the message.
   (d) Call `gmp_fprint()` and print the contents onto `outfile`.
4. Free the array we dynamically allocated by calling `free()`.

## 4.6 `void ss_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t pq)`

This function will create decrypt a ciphertext `c` by calling `pow_mod` to calculate $c^d$ mod `pq` and store it in `m`.

### 4.7 `void ss_decrypt_file(FILE *infile, FILE *outfile, mpz_t pq, mpz_t d)`

This function will decrypt the contents of `infile` and store the decrypted plaintext into `outfile`.

1. An array will be dynamically allocated of size `k` using `calloc()`. The value of `k` will be calculated by the taking $(log2(\texttt{pq}) - 1)/8)$.
2. Next, create a `do-while` loop that will scan each block using `gmp_fscanf() and store the contents into mpz_t c`.
   (a) It will break from the loop whenever we reach the end of the file, indicated by the `EOF` variable.
   (b) Decrypt `c` by calling `decrypt()` and store the plaintext into `m`.
   (c) Call the `mpz_export()` function which serves to convert the binary data back and store it in the array we dynamically created earlier.
   (d) Write the contents of the decrypted message by calling `fprintf()`.
3. Free the array we dynamically allocated by calling `free()`.