Assignment 5: “Public Key Cryptography”
WRITEUP

Vincent Liu
2/25/23
Prof. Veenstra

---

1. Introduction

Assignment 5, titled “Public Key Cryptography” allows for students to familiarize themselves with the basics of Cryptography as well as the implementation of multiprecision integers within C. The particular type of cryptosystem within this assignment is the Schmidt-Samoa cryptosystem. This method heavily relies on the factorization of large integers, which allows for better security of encrypted messages. The purpose of this assignment is for students to understand the process in which encryption, decryption and key generation is done, as well as program the cryptosystem themselves. Students are expected to create three main files for this assignment; A key generator, an encryptor, and a decryptor. Students will also be creating supplemental c files that contain the required functions for this assignment. These consist of ss.c, randstate.c and numtheory.c. Additionally, a Makefile must be created by students to compile their code, create executable files, and clean the directory when requested. (note:students are expected to install gmp as well as pkg-config. Both are necessary for the completion of this assignment. “pkg-config” is a utility used to assist in finding and linking libraries. “Gmp” is used for precision integers.)

---

2. Example of Encryption and Decryption of a text file

In this example of my code, I will show a text file containing a message that will be encrypted and decrypted. I will also be showing the public and private key used for encryption and decryption.

Public Key:

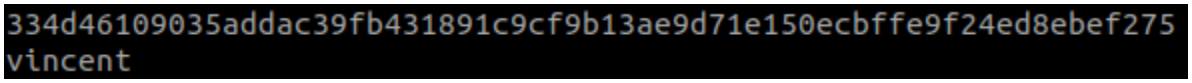

Figure 1: ss.pub (Public Key)

This figure shows the contents of ss.pub, which contains a public key generated from two large primes (p and q) that are a specified *nbits* long. The public key also contains the user's username. The function used to create this key is "ss_make_pub", and can be found in ss.c.

The function used to write the public key file to ss.pub is "ss_write_pub" and can also be found in ss.c. This function writes the key as hexstring and writes it onto ss.pub followed by a newline. It then writes the username, followed by another newline.

Private Key

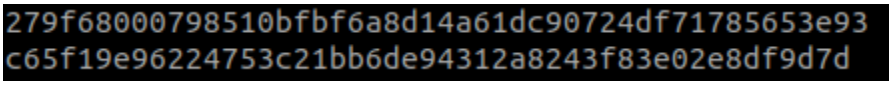

Figure 2: ss.priv (Private key)

This figure shows the contents of ss.priv, which contains a private key generated by two primes and a public key. The function used to create the key is "ss_make_priv". It creates the first line seen in figure 2, which is done by calculating the inverse of n modulo $\lambda(pq) = \text{lcm}(p-1,q-1)$. Writing to ss.priv is done by "ss_write_priv". It writes the key, followed by pq (calculated by other function). Both are written as hexstrings and have following newlines.

Input.txt:

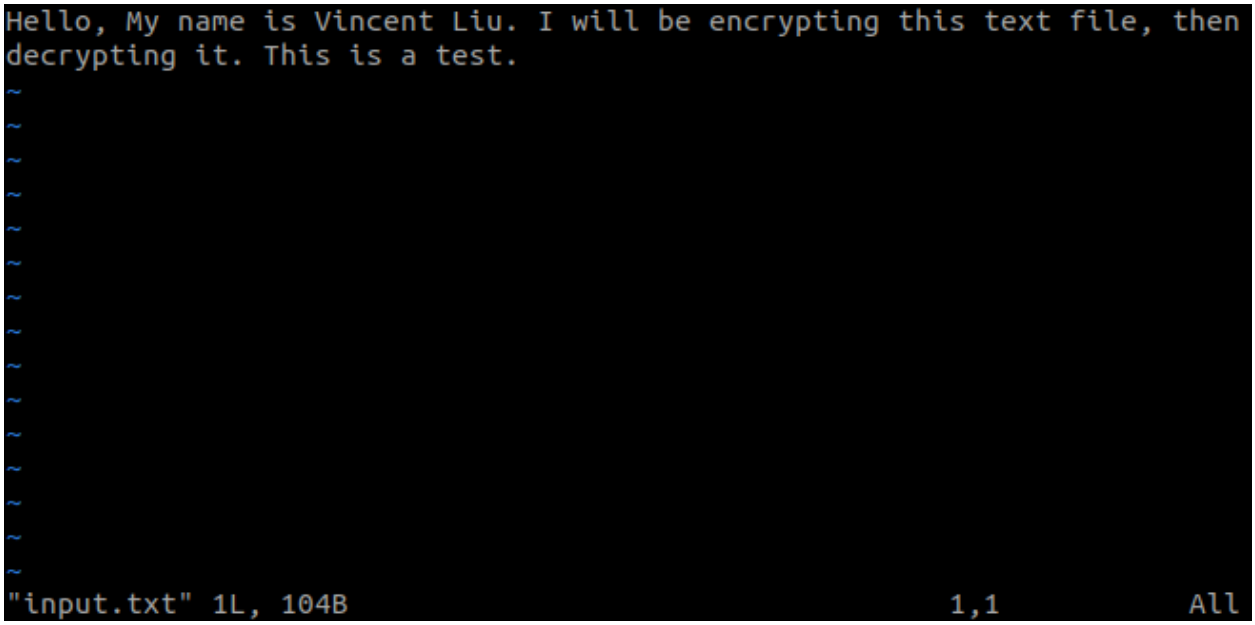

Figure 3: input.txt

In this image, we are able to see what is included in the input text file that will be encrypted.

Output after encrypting:

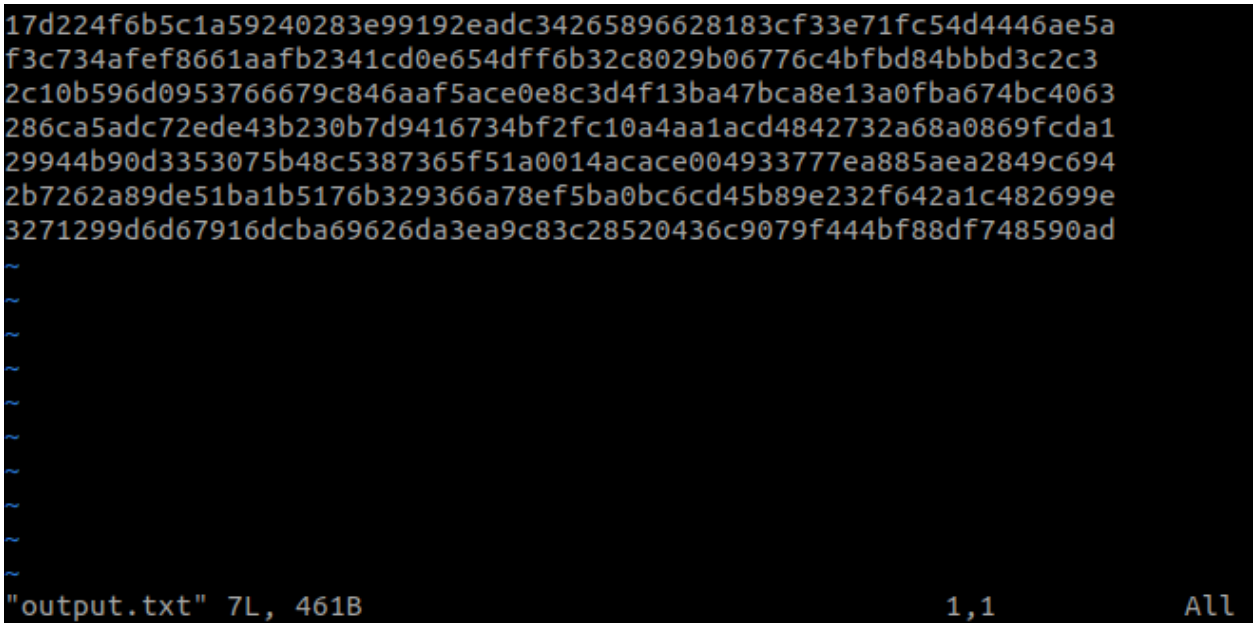

Figure 4: output.txt

In this figure, we are able to see the encrypted output of the encrypt.c main file after running the command: "**./encrypt -i input.txt -o output.txt**". As can be seen, the program separates the encrypted data into blocks.

Output after Decrypting

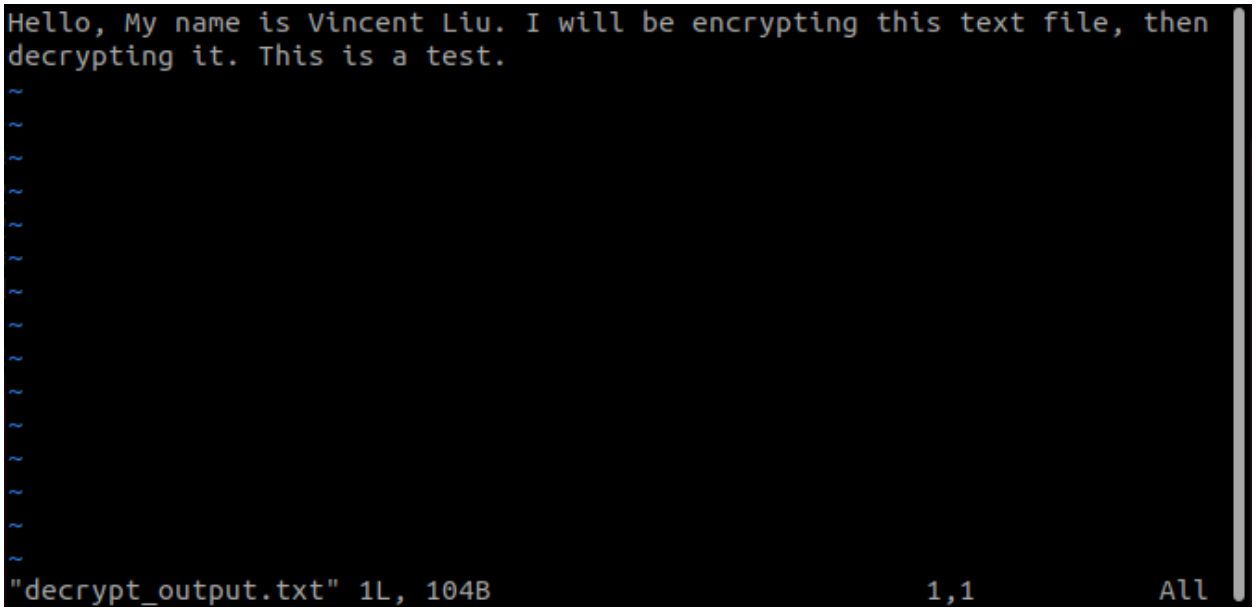

Figure 5: decrypt_output.txt

In this figure, we are able to see the decrypted version of figure 4. The command used to do this is: "**./decrypt -i output.txt -o decrypt_output.txt**". The function used to decrypt *output.txt* was "ss_decrypt" and "ss_decrypt_file". These function essentially are the reversed versions of their encrypting counterparts

---

3. Functions used in this assignment

This assignment requires students to create three main files (written in c) in conjunction with three supporting '.c' files as well as a Makefile. The functions included in these files will be elaborated on below. Note: The steps taken in each of the main files will not be elaborated on here, but on the DESIGN.pdf.

3.1 keygen.c

Is one of the main files that is used to generate a public and private key for cryptography. This main file (unless otherwise specified) creates ss.pub and ss.priv for their respective keys. These keys are used to encrypt and decrypt messages within encrypt.c and decrypt.c. Keygen automatically gets the user's username and saves it to be printed on the public key file. Keygen also sets the private key permissions such that only the user has read and write permissions. This

file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -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: ss.pub).
- -d pvfile : specifies the private key file (default: ss.priv).
- -s : specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL)).
- -v : enables verbose output.
- -h : displays program synopsis and usage.

3.2 encrypt.c

encrypt.c one of the main files, and is used to encrypt a text file or an input from stdin. Utilizes the public key (ss.pub) generated by keygen.c (or a specified public key file) to properly encrypt the desired message. This main file uses stdin and stdout as input and output files unless otherwise specified. If an output file is stated but does not exist, encrypt.c will create said file and print to it. This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -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: ss.pub).
- -v : enables verbose output.
- -h : displays program synopsis and usage.

3.3 decrypt.c

Is one of the main files required for this assignment. As the name suggests, this main file is used to decrypt an encrypted text file (that was encrypted by encrypt.c) or from a stdin input. Additionally, this file will use stdout unless a desired (and valid) output file is stated. If an output file is stated but does not exist, decrypt.c will create said file and print to it. Decrypt.c utilizes the key generated by keygen.c (or a specified private key file) to properly decrypt the encrypted message. This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -i : specifies the input file to decrypt (default: stdin).
- -o : specifies the output file to decrypt (default: stdout).
- -n : specifies the file containing the private key (default: ss.priv).
- -v : enables verbose output.

- -h : displays program synopsis and usage.

3.4 randstate.c

Initializes the randomizer state. Also clears the randomizer state.

- **void randstate_init(uint64_t seed)**
  - Initializes the global random state named state using ***seed*** as the random seed.
- **void randstate_clear(void)**
  - Clears and frees all memory used by the initialized global random state named state.

3.5 numtheory.c

Contains Number Theoretic Functions to be used in the main files as well as ss.c

- **void pow_mod(mpz_t o, mpz_t a, mpz_t d, mpz_t n)**
  - Performs fast modular exponentiation, computing ***a*** raised to the ***d*** power modulo ***n***, and storing the computed result in ***o***
- **void mod_inverse(mpz_t o, mpz_t a, mpz_t n)**
  - Computes the inverse ***o*** of ***a*** modulo ***n***.
- **void gcd(mpz_t g, mpz_t a, mpz_t b)**
  - Computes the greatest common divisor of ***a*** and ***b***, storing the value of the computed divisor in ***"mpz_t g"***.
- **bool is_prime(mpz_t n, uint64_t iters)**
  - Conducts the Miller-Rabin primality test and determines if ***'n'*** is a prime number
- **void make_prime(mpz_t p, uint64_t bits, uint64_t iters)**
  - Generates a new prime number and stores it in ***"mpz_t p"***

3.6 ss.c

Contains functions that are essential to performing the Schmidt-Samoa cryptosystem

- **void ss_make_pub(mpz_t p, mpz_t q, mpz_t n, uint64_t nbits, uint64_t iters)**
  - Creates parts of a new SS public key: two large primes ***p*** and ***q***, and ***n*** computed as p*p*q
- **void ss_write_pub(mpz_t n, char username[], FILE *pbfile)**
  - Writes a public SS key to ***pbfile***.

- **void ss_read_pub(mpz_t n, char username[], FILE *pbfile)**
    - Reads a public SS key from ***pbfile***.
- **void ss_make_priv(mpz_t d, mpz_t pq, mpz_t p, mpz_t q)**
    - Creates a new SS private key ***d*** given primes ***p*** and ***q*** and the public key ***n***.
- **void ss_write_priv(mpz_t pq, mpz_t d, FILE *pvfile)**
    - Writes a private SS key to ***pvfile***.
- **void ss_read_priv(mpz_t pq, mpz_t d, FILE *pvfile)**
    - Reads a private SS key from ***pvfile***.
- **void ss_encrypt(mpz_t c, mpz_t m, mpz_t n)**
    - Performs SS encryption, computing the ciphertext ***c*** by encrypting message ***m*** using the public key ***n***.
- **void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n)**
    - Encrypts the contents of ***infile***, writing the encrypted contents to ***outfile***.
    - Data in ***infile*** will be encrypted in blocks
- **void ss_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t pq)**
    - Performs SS decryption, computing message ***m*** by decrypting ciphertext ***c*** using private key ***d*** and public modulus ***n***.
- **void ss_decrypt_file(FILE *infile, FILE *outfile, mpz_t pq, mpz_t d):**
    - Decrypts the contents of ***infile***, writing the encrypted contents to ***outfile***.
    - Data in ***infile*** will be decrypted in blocks

3.7 Makefile

This file is used to compile the program. It also creates object files (‘.o’ files) and the necessary executable files for this assignment. The executable files created by Makefile for this assignment include: ‘keygen’, ‘encrypt’, and ‘decrypt’. The object files for this assignment are as follows:

- randstate.o:
- numtheory.o:
- ss.o
- decrypt.o:
- encrypt.o:
- keygen.o:

The Makefile also properly formats all ‘.c’ files to the c17 standard
Additionally, the Makefile includes the ’pkg-config’ required for the simplicity of library inclusion for files within this assignment.

---

4. UNIX commands used

- **./keygen**: Used to run the keygen executable main file. Expects a passed argument. Arguments are as follows:
    - -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: ss.pub).
    - -d pvfile : specifies the private key file (default: ss.priv).
    - -s : specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch, given by time(NULL)).
    - -v : enables verbose output.
    - -h : displays program synopsis and usage.

- **./encrypt**: Used to run the encrypt executable main file. Expects a passed argument. Arguments are as follows:
    - -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: ss.pub).
    - -v : enables verbose output.
    - -h : displays program synopsis and usage.

- **./decrypt**: Used to run the keygen executable main file. Expects a passed argument. Arguments are as follows:
    - -i : specifies the input file to decrypt (default: stdin).
    - -o : specifies the output file to decrypt (default: stdout).
    - -n : specifies the file containing the private key (default: ss.priv).
    - -v : enables verbose output.
    - -h : displays program synopsis and usage.

- **make / make all** : Used to create executable files as well as compile the necessary '.c' files into objects files (".o" files)

- **make clean** : Used to remove the executable main file as well as remove the associated object files

- **make decrypt**: Used to build only the decrypt program

- **make encrypt**: Used to build only the encrypt program

- **make keygen**: Used to build only the keygen program

- **make format** : Used to properly format all '.c' files.

---

5. Files in the directory

- **randstate.c**
    - This contains the implementation of the random state interface for the SS library and number theory functions

- **randstate.h**
    - This specifies the interface for initializing and clearing the random state.

- **numtheory.c**
    - This contains the implementations of the number theory functions.

- **numtheory.h**
    - This specifies the interface for the number theory functions.

- **ss.c**
    - This contains the implementation of the SS library.

- **ss.h**
    - This specifies the interface for the SS library.

- **keygen.c**
    - This contains the implementation and main() function for the keygen program.

- **encrypt.c**
    - This contains the implementation and main() function for the encrypt program.

- **decrypt.c**
    - This contains the implementation and main() function for the decrypt program.

- **Makefile**
    - Used to compile all '.c' files into '.o' object files, create an executable file, or remove all object files and executables from the directory.

---

6. Steps taken to complete assignment?

As required by this assignment, I first had to learn what GNU MP was, and how to use it. By reading the gmp library manual, I was able to understand that this library provided functions that would allow me to use and accurately calculate arbitrarily large integers. As stated before in the DESIGN document, the SS cryptosystem heavily relies on the factorization of large integers, which assists in the security of the encrypted messages. Additionally, I was able to get a basic understanding of which functions would be required in order to complete each and every function.

I decided to first complete the Makefile required for my assignment to properly compile. Copying over previously used Makefiles and adjusting them to my needs was quite simple. As required by the assignment, I added "pkg-config" to the makefile. Following advice given by Professor Veenstra, I first began working on "randstate.c", as he stated that it was the easiest file to complete amongst all required files (which was true). All that was required of this file was the initialization of the random state, as well as the clearing of the gmp randomized state. In total, both functions only required 4 lines of code. Additionally, the instructions were already provided in the assignment documentation, making it quite simple and straightforward. The second file I decided to complete was "numtheory.c". For this file, I simply followed the general pseudo code and instructions provided in the assignment documentation. I found the pseudo code to be quite easy to understand after reading what the purpose of the function was, as well as what the function was expected to output. The third file I decided to complete was "ss.c". I found this one to be a bit of a challenge in comparison to the other C files, as I had to make use of many gmp library functions that were confusing to understand at first, such as "mpz_str_in" and "mpz_import". As I read through the manual over and over I got a better understanding of these functions and used them in "ss.c". In all honesty, I believe this was the most difficult file for me to complete within this assignment. The fourth file I worked on was "keygen.c". This one was pretty straightforward, as the instructions were given in the assignment documentation. Something I found interesting in this file was the use of "getenv()", because it was able to get a user's username. Of course I understood that this was possible, however what I did not know at the time was how easy it would be to obtain this information. Moreover, I found it interesting how I was able to ensure that the private key file permissions were set such that only the user could read and write/overwrite it. I then completed "encrypt.c" and "decrypt.c" simultaneously, as both were quite similar in structure. If given the command, both would read from and write to specified files. Otherwise, both would (by default) use stdin and stdout, as well as ss.pub and ss.priv for the public and private keys, respectively. encrypt.c used the public and decrypt.c used the private)

---

7. What I learned

From this assignment I learned how to create the Schmidt-Samoa cryptosystem. With the use of a Schmidt-Samoa library (ss.c) as well as theoretical number functions, I was able to encrypt and decrypt messages, as well as create public and private keys for users. My understanding of this cryptosystem is that a pair of keys is created for use between two users. These keys (public and private) are created using two arbitrary large primes. Said keys can be used to encrypt and decrypt messages sent between the holders. One set of keys cannot be used to encrypt or decrypt messages that utilize a separate pair of keys.

Something I found especially useful within this assignment was the usage of GNU Multuple Precision Arithmetic library. I was able to learn how to use this library to (what I believe was) a basic extent. I understand from this assignment that C does not (natively) support large precision integers and that SS cryptosystem is heavily reliant on it for the creation of keys and encryption. Therefore, usage of this library is necessary if one were to do some form of cryptography, such as the Schmidt-Samoa cryptosystem. To use the library, one must first (as anyone would) include the library header file. To use the mpz variables, one must follow these steps: declare, initialize, set values, use, clear.

To declare, I did "*mpz_t variable_name*" (*variable name* as an example). To initialize it, I did "*mpz_init(variable_name)*". To initialize AND set, I did "*mpz_set_init_ui(mpz_t rop, unsigned int)*" or "*mpz_set_init(mpz_t rop, const mpz_t op)*". To simply set a variable value after separate initialization, I did "*mpz_set(mpz_t rop, const mpz_t op)*" or "*mpz_set_ui(mpz_t rop, unsigned int)*". The former changes an mpz variable to an unsigned int, while the latter sets it to an already initialized value found within another mpz variable. To use the aforementioned variables, one could do a myriad of commands such as:

- **mpz_add** *(mpz_t rop, const mpz_t op1, const mpz_t op2)*
    - Adds two mpz variables and places the result in ***mpz_t rop***
- **mpz_add_ui** *(mpz_t rop, const mpz_t op1, unsigned long int op2)*
    - Adds an mpz variable to an unsigned integer and places the result in ***mpz_t rop***
- **mpz_sub** *(mpz_t rop, const mpz_t op1, const mpz_t op2)*
    - Subtracts an mpz variable from another mpz variable and places the result in ***mpz_t rop***
- **mpz_sub_ui** *(mpz_t rop, const mpz_t op1, unsigned long int op2)*
    - Subtracts an unsigned integer from an mpz variable and places the result in ***mpz_t rop***
- **mpz_mul** *(mpz_t rop, const mpz_t op1, const mpz_t op2)*
    - Multiplies two mpz variables and places the result in ***mpz_t rop***
- **mpz_mul_ui** *(mpz_t rop, const mpz_t op1, unsigned long int op2)*
    - Multiplies an mpz variable and an unsigned integer and places the result in ***mpz_t rop***
- **mpz_fdiv_q** *(mpz_t q, const mpz_t n, const mpz_t d)*

    - Divides two mpz variables and places the result in ***mpz_t q***
- **mpz_fdiv_q_ui** *(mpz_t q, const mpz_t n, unsigned long int d)*
    - Divides an mpz variable by an unsigned integer and places the result in ***mpz_t q***
- **mpz_inp_str** *(mpz_t rop, FILE *stream, int base)*
    - Place the value of mpz variable into a file and writes it as a string of *base* type. This assignment requires values to be written in hexstrings.
- **mpz_out_str** *(FILE *stream, int base, const mpz_t op)*
    - Reads a string of *base* type and from a file and places it into an mpz variable. This assignment requires values to be written in hexstrings.
- **mpz_clear** *(mpz_t x)*
    - Frees space occupied by variable *x*
- **mpz_import** *(mpz_t rop, size_t count, int order, size_t size, int endian, size_t nails, const void *op)*
- **mpz_export** *(void *rop, size_t *countp, int order, size_t size, int endian, size_t nails, const mpz_t op)*
- **mpz_sizeinbase** *(const mpz_t op, int base)*

Note that these are the main GNU MP functions used

I found it quite interesting that the factorization of integers makes the deciphering of encrypted messages difficult. On top of this, I never really realized how much mathematics was involved in cryptography, however in all honesty, I never gave it much thought before this assignment. Now I am able to better understand how mathematics plays a large role not only in encryption but also in cryptography as a whole.

Applications of public-private cryptography

My understanding is that the main use of cryptography is to ensure the security and confidentiality of messages being sent between two people, groups, or organizations. Today, cryptography is used worldwide by various organizations and people, all of whom have a task in which either the general public or other organizations are not to be informed of said messages. For example, messages being sent between two governments that contain classified information that no foreign adversary should know, otherwise a conflict could ensue. One such instance is the historical rivalry between US intelligence agencies (FBI, CIA, etc) and the USSR's intelligence agency(agencies) (KGB, FSS, etc). Both countries spied on each other and sent crucial information to their respective handlers (mostly) in encoded messages that their adversaries would attempt to decipher and read. Though not always talked about or thought about, this still occurs today, where countries such as the US still attempt to spy on foreign nations for intel and activity. The average person could similarly use cryptography to send messages without the need to excessively worry about others reading their communications. I myself could probably use this

if I really wanted to send a message to a friend without having to worry about other people taking my phone and reading my texts.

---

8. Credit:

https://gmplib.org/manual/index

-I learned how to use many gmp functions within this manual. Helped in understanding mpz functions as well as implementing them into this assignment. Most pages were read and understood, and functions within the number theoretic functions section were avoided, as the assignment explicitly states that they are not to be used.