Assignment 6: Public Key Cryptography

Monday, November 8, 2021 3:14 PM

Very helpful tool: https://gmplib.org/manual/Concept-Index

RSA:

D	Private key
N & e	Public key

P and q are LARGE NUMBERS!

N = p*q

N should be a 4096 bit number.

- We cant uint32 anymore so we use the GMP library.
- Large numbers are harder to decrypt.

RSA Algorithm

- The RSA Algorithm is about factoring the product of two large prime numbers. It is a factoring problem.
- There are no published methods to defeat the system if a large enough key is used, but RSA would be vulnerable to be attacked by a quantum computer. I did a whole research paper in quantum computers and they are very interesting.
- RSA involves a public & private key:
  - Everyone knows the public key. It is used to encrypt messages.
  - Files and messages can only be decrypted by using the private key.

N	Public key (int)
E	Public key (int)
D	Private key (int)

Public key	Modules n & public exponent e.
Private key	Private exponent d (kept secret) p, q p(n) kept secret

In this assignment I created three main() files. The first main file (keygen.c) generates the public and private key pairs. These keys are put into their specified files when the program is run. The second main file (encrypt.c) encrypts files using the public key. The third main file (decrypt.c) decrypts the encrypted message. There are two other files (numtheory.c & rsa.c) that contain functions that are used in keygen, encrypt, and decrypt. There is also one more file that implements a small random state module that contains an extern declaration to a global random state variable called state. This implementation is located in (randstate.c).

Example -> Making a Public and Private Keys:

Alice: primes p and q

N = pq (n is a very large number)

Totient of n = (p - 1) (q - 1)

- Totient: for any number n, the totient of n are the number of integers lower than it that are co prime with it.

  7, 3 = co prime 7 is co prime to 3

  4, 2 = not co prime

Totients mathematically explained and worked out:

Totient(5) = 4 (4, 3, 2, 1)

Totient(p) = p - 1

Totient(q) = q - 1

Totient(pq) = (p - 1)(q - 1) = n

e = 65537

- Calculate the inverse of e which is d.

  D = e^-1 mod totient(n)

  D is congruent to 1 mod totient(n).

Encryption:

E(m) = m^e mod n

Decryption:

D(E(m)) = (E(m))^d mod n

$e^{16} + 1 = 65537$

- How many 1 bits are in that number
- Using e = 65537 RSA is an extra precaution against a variety of attacks that are possible when bad message padding is used.
  - These attacks tend to be more likely or devastating with a much smaller e.
- Relatively large prime, so it is easier to arrange that gcd(e, p(n) = 1 and it is one more than a power of 2, so raising a number to the 65537th consists mostly of squarings.

  https://www.johndcook.com/blog/2018/12/12/rsa-exponent/

P	Function is called the totient of n

- Denotes the number of positive integers up to a given integer n that are relatively prime to n

Public key	<e, n>

$$D(m) = m^e \ (mod\ n) \quad and \quad E(c) = c^d (mod\ n)$$
$$\bigvee m \in \{0, \dots, n-1\} \quad that\ D(E(m)) = E(D(m)) = m$$

- Since D and E are mutual inverses.
- This program perform not only encryption but also digital signatures.

Task (3 binaries)

1. **A key generator: keygen**
   - In charge on key generation: producing RSA public and private key pairs.
2. ~~**An encryptor: encrypt**~~
   - Encrypt files using a public key.
3. **A decryptor: decrypt**
   - Decrypt the encrypted files using the corresponding private key.

- Implement two libraries and a random state module that will be used in each of your programs
  1. Library 1: holds functions relating to the math behind RSA.
  2. Library 2: contain implementations of routines for RSA.
  3. GNU multiple precision arithmetic library - need to learn how to use this

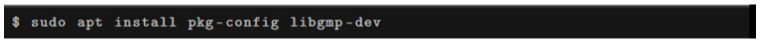
GNU Multiple Precision Arithmetic

- C unlike Python, does not natively support arbitrary precision integers
- RSA relies on large integers
  - This is why we use the GNU multiple precision arithmetic library (referred to as GMP)
  - https://gmplib.org/manual

-

Install:	gmp	pkg-config

  - Latter is a utility used to assist in finding and linking libraries instead of having the program hard code where to find specific headers and libraries during program compilation



```
$ sudo apt install pkg-config libgmp-dev
```

- Attend section for assistance on using pkg-config in a Makefile to direct the compilation process of your programs.
  - GMP already provides number theoretic functions that could be used in RSA
    - You CANNOT use any GMP-implemented number theoretic functions, you must implement these functions yourself.

FUNCTIONS TO IMPLEMENT
(these functions require the use of random, arbitrary-precision integers)

Random State Variable

randstate.h -> randstate.c

GMP (multiple precision arithmetic library) requires use to explicitly initialize a random state variable and pass it to any of the random integer functions in GMP

Dedicated function that cleans up any memory used by the initialized random state

cp files ~/cse13s/asgn6

SO: randstate.h -> randstate.c

Implement a small random state module:

- Contains a single extern declaration to a global random state variable called state and two functions:
    1. Initialize the state.
    2. Clear the state.

void randstate_init(uint64_t seed)

- Initialize the global random state (state).
- Use Mersenne Twister algorithm:
    - Use the 'seed' as the random seed.
    - Use gmp_randinit_mt() and gmp_randseed_ui().

void randstate_clear(void)

- This function clears and frees the memory used by the initialized global random state 'state'.
    - Single call to gmp_randclear()

Number Theoretic Functions
numtheory.h -> numtheory.c

- Branch of math that studies the nature and properties of numbers.
- Most important contribution to the field for public-key cryptography are Fermat and Euler
    1. Implement the functions that drive the mathematics behind RSA before you can table your RSA library:
       numtheory.h & numtheory.c

Modular Exponentiation

Compute a^n

This Is the WRONG way to compute a^n because this is VERY inefficient.

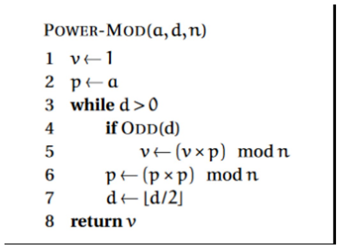
$$a^n = \overbrace{a \times a \times \cdots \times a \times a}^{n}.$$

The better way to compute a^n in $O(\log_2 n)$)

$$a^n = a^{c_m 2^m} \times a^{c_{m-1} 2^{m-1}} \times \cdots \times a^{c_1 2^1} \times a^{c_0 2^0} = \prod_{0 \le i \le m} a^{c_i 2^i}.$$

$$a^{13} = a^{2^3+2^2+2^0} = a^{8+4+1} = a^8 \ x \ a^4 \ x \ a^1$$
$$a^{15} = a^{2^3+2^2+2^1+2^0} = a^{8+4+2+1} = a^8 \ x \ a^4 x \ a^2 x \ a^1$$

void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus);



```
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
```

This function performs fast modular exponentiation, computing base raised to the exponent power modulo modulus and storing the computed result in out.

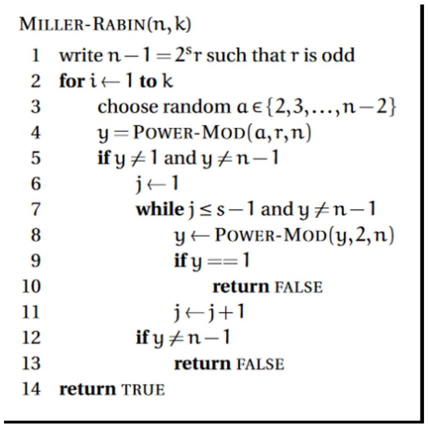
Primality Testing

- The simplest test is trial division.
  Given an input number n, check whether it is evenly divisible by any prime number between 2 and square root of n.

Millar - Rabin primality test:

This test detects all composites (for every composite number n, there are at least 3/4 numbers a that are witnesses of compositeness of n).

Given an integer n, choose some positive integer a < n.
Let 2^s d = n -1 where d is odd.
If a^d does not equal 1 (mod n) and a ^2rd does not equal 1 (mod n) for all 0 <= r <= s - 1.



```
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
```

Generate a number, check its size base 2
Mpz_sizeinbase(x)

void is_prime(mpz_t n, uint65_t iters);

Conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations.

This funciton is needed when creating the two large prime p and q in RSA, verifying if a large integer is prime.

MILLER-RABIN(n, k)

First check if n is 2 or 2:

In the case that they are, return true because they are prime.
In this case you would return true.

Then check if n is 1, 0, or even:

If they are, this means that n cannot be a prime number.
In this case you would return false.

Then you must initialize and set your temporary variables for:

R, a, y, lower, upper, n_1, and mp2

First you must guess r.
Write n - 1 = -1 2^s r such that r is odd.

S is some exponent.
N is going to be 2 the something times r plus 1.

This involves a loop to make sure that it is odd if r is odd to begin with then you stop.

Compute s and r and r should end up odd.

```
If n % 2 equals 0:
    Return false.
Otherwise, for I from 1 to k (inclusive).
    Choose a random subset of {2, 3, …, n - 2}.
    Y = power_mod(y, 2, n).

    If y does NOT equal 1 and y.
```

void make_prime(mpz_t p, uint64_t bits, uint64_t iters);

This functions generates a new prime number stored in p.

The generated prime should be at least bits number of bits long.

The primality of the generated number should be tested using is_prime() using iters numbers of iterations.

Generate a random p by using mpz_urandomb().

While p is not prime, generate numbers until you find a prime number.

```
while the number is not prime:
    p  is set to gmp_urandomb.
    If you generate a prime number:
        Stop the loop.
```

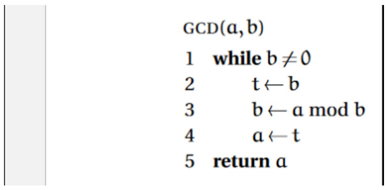
Modular Inverses

Euclidean algorithm (Euclid's algorithm): compute greatest common divisor of two integers
(largest number that divides them both with a zero remainder.

Compute the remainder (subtract the smaller number from the larger until it is no longer larger).

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

Compute the greatest common divisor of two integers (a and b).
Store the value in d.



```
GCD(a, b)
1  while b ≠ 0
2      t ← b
3      b ← a mod b
4      a ← t
5  return a
```

The extended Euclidean algorithm is particularly useful when a and b are coprime.

X = a mod (b)
Y = b mod (a)

Bezout's identity asserts that a and n are coprime if and only if there exist integers s and t such that:

Ns + at = 1
At == 1 (mod n)

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

Pseudo code from assignment document.

```
The use of temporary variables is very important in this function!

(r, r_prime ) <-- (n, a)
(t, t_prime) <-- (0, 1)

While r_prime doesn't equal 0:
    Set q to the floor of r divided by r_prime.
        Q <-- floor( r / r_prime )
    Then set r to r_prime and r_prime to r minus q times r_prime.
        ( r, r_prime ) <-- ( r_prime, r - q * r_prime )
    Then set t to t_prime and t_prime to t minus q times t_prime.
        ( t, t_prime ) <-- ( t_prime, t - q * t_prime )

If r is greater than 1:
    Return no inverse.

If t is less than zero:
    Set t to t plus n.

Return t.
```

RSA Library
rsa.c rsa.h

void rsa_make_pub(mpz_t p, mpz_t q, mpz_t n, mpz_t e, uint64_t nbits, uint64_t iters);

Explaining parameters and functionality:
- The return p is passed in as empty for you to eventually fill in in this function.
- This function ideally makes your public key.
- Not only is p empty, so are q n and e.
- Nbits is at least how large n should be.

First thing you should do is make your primes.
- Generate a pseudo random number with srandom() and time(NULL).
- Then calculate your b bits with random(). The range to calculate your pbits in is below:
  - [nbits / 4, 3 * nbits / 4]
  - You just need a way to offset it.
  - Random() % 7 = [0 -> 6]
  - [3 to 9] you need an offset by three so: 3 + (random() % 7) = [3 to 9]

- qbits = nbits - pbits

- To make p and q, you just call your make_prime() function on both.
- Set p to make_prime().
- Set q to make_prime().

Compute the totient of n:
- totient = (p - 1) x (q - 1)

Coprime:
- A and b are coprime if gcd(a, b) = 1.
  - These are relatively prime to eachother.

Loop
- Generate a random e.
- Check if it is coprime to the totient.
  - (to check if something is co prime, you need to use gcd)
  - A and b are coprime if and only if gcd(a, b) == 1.
  - This is all for the purpose of making sure we can decrypt as well.

void rsa_write_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile);

This funciton writes the public key to the public key file.
Use gmp_fprintf(pbfile "%Zd\n", n);
The same thing for e and then s.
fprintf(pbfile "%s\n", username);

void rsa_read_pub(mpz_t n, mpz_t e, mpz_t s, char username[], FILE *pbfile);

This function reads from the pbfile the public key.
Use fscanf!
Get n, e, s.
fscanf(pbfile, "%s\n", username);

void rsa_make_priv(mpz_t d, mpz_t e, mpz_t p, mpz_t q);

This function makes a private key.
d is empty, e p q are filled in.

First, you create mpz_t variables for the totient,(p - 1), and (q - 1).
Then you calculate (p - 1) and (q - 1).
Multiply these two variables and set them to the totient.
Calculate the mod_inverse() of e and the totient and set that to d (the output).

void rsa_write_priv(mpz_t n, mpz_t d, FILE *pvfile);

This function writes the private key to the file.
Use gmp_fprintf().

void rsa_read_priv(mpz_t n, mpz_t d, FILE *pvfile);

This function reads the public key into the program.
Reads a private RSA key from pvfile.
The format should key should be n then d.
Both of which should have been written with a trailing newline.
HEXSTRING!

void rsa_encrypt(mpz_t c, mpz_t m, mpz_t e, mpz_t n);

This file performs RSA encryption, computing ciphertext c by encrypting message m using public exponent e and modulus n.

C = cypher text
M = message
E = public key
N = modulos

Modular exponentiation:
    This function is just one call pow_mod().

Takes a message (raise power e and mod by n).
write that to the file.

Remember, encryption with RSA is defined as

$$E(m) = c = m^e \ (\ mod\ n\ )$$

void rsa_encrypt_file(FILE*infile, FILE*outfile, mpz_t n, mpz_t e);

Scan in bytes and convert to mpz_t, but how many bytes?
    2 % 5 = 2
    7 % 5 = 2
    12 % 5 = 2

Value of a block cannot be 0 or 1.

0000 0000 0000 0000
    Convert to integer -> 0
    From seeing zero, how would anyone know how many bytes it was originally?
    This is why we must prepend a single byte to the front of the block (0XFF (255))

1111 1111 0000 0000 0000 0000
    Convert integer and now we can always go back to original form.
    This is the act of prepending.

fread() <- stdio.h
    To get bytes
Mpz_import(bytes) to convert it back into a large int m
C = Rsa_encrypt(m)
gmp_fprintf(c to the output file)

Calculate the block size k.

$$k = \lfloor(\log_2(n) - 1)/8\rfloor$$

Dynamically allocate an array that can hold k bytes.

Type ---> uint8_t *
This array serves as a block.

Set the zeroth byte of the block to 0xFF.
This will prepend the workaround byte that we need.

While there are still unprocessed bytes in infile:
Read k to 1 bytes from the infile.
This portion is similar to encode and decode in assignment 5.
START AT INDEX 1 SO YOU DON'T OVERWRITE 0xFF.

mpz_import()

- This is used to convert the read bytes into an mpz_t m.

Encrypt m with rsa_encrypt(), then write the encrypted number to outfile as a hex string followed by a trailing newline.

void rsa_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t n);
The difference between this and rsa_encrypt is exponent.
You take the exponent it to power of d.

D is the private key.
Modular exponentiation:
So this function is just one call to pow_mod().

void rsa_decrypt_file(FILE *infile, FILE *outfile, mpz_t n, mpz_t d);

Allocate memory for the uint8_t buffer using the size of a unit8_t.
If we want to fill up this buffer starting from index 2 and onwards
Set the first index of the block array to 0xFF:
buffer[1] = 0XFF

Loop while you haven't reached the end of the file:
Set c equal to gmp_fscanf().
Set m equal to rsa_decrypt.
mpz_export(m)
Use fwrite() to write the file to the output file.

Use pointer arithmetic:

- This is because array index 1.

void rsa_sign(mpz_t s, mpz_t m, mpz_t d, mpz_t n);
write pow pod correctly its only one line
store value in s (s initially empty)

Performs RSA signing producing signature s by signing message m using private key d and public modulus n. Signing with RSA is defined as

$$S(m) = s = m^d \ (\, mod\ n\,)$$

bool rsa_verify(mpz_t m, mpz_t s, mpz_t e, mpz_t n);
This function performs RSA verification.
This function returns true if the signature s is verified and false otherwise.
Verification is the inverse of signing.

$$t = V(s) = s^e \ (\, mod\ n)$$

The signature is verified if and only if t is the same as the expected message m

Pow_mod
m is message

Key Generator
keygen.c

```
#define OPTIONS "b:i:n:d:s:vh"

For the switch cases:
        Define the public file as STDIN_FILENO;
        Define the private file as STDOUT_FILENO;

        Set the verbose as false.

        While ((opt = getopt(argc, argv, OPTIONS)) != -1) {
                Switch (opt) {
                        case 'b': specifies the minimum bits needed for the public mod n

                        case 'i':  specifies the number of Miller-Rabin iterations for testing primes (default: 50)

                        case 'n': specifies the public key file (default rsa.pub)
                                Use open()

                        case 'd': specifies the public key outfile.

                        case 's': specifies the random seed for the random state initialization (default: the seconds since the UNIX epoch,
                        given by time(NULL))

                        case 'h': print the help message.

                        default: in the case that the user enters the wrong input, print the help statement.
```

1. Parse command-line options using getopt()

2. Open public and private key using fopen(). Print a help error and exit the program in the event of failure
    Open private and public key files
    .
3. Using fchmod() and fileno() make sure that the private key file permissions are set to 0600 indicating read and write permissions for the user, and no permissions for anyone else.
    PERMISSIONS

4. Initialize the random state using randstate_init(), using the set seed
    Use seed that is parsed in command, and if the user doesn't give a seed, the default is 50.

5. Make the public and private keys using rsa_make_pub() and rsa_make_priv()
    These functions are from rsa.c

6. Get the current user's name as a string. You will want to use getenv()
    getenv() gets the file path.
    Use USERNAME

7. Convert the username into an mpz_t with mpz_set_str() (base 62)
    Then use rsa_sign() to compute the signature of the username.

8. Write the computed public and private key to their respective files.
    These functions are in rsa.c

9. If the user wants the verbose statistics, print those out.

10. CLOSE FILES!

MAKEFILE:

```
CC=clang
CFLAGS= -Wall -Werror -Wextra -Wpedantic $(shellpkg-config --cflags gmp) -g
LDFLAGS=$(shellpkg-config --libs gmp) -lm
COMMON_OBJS=rsa.o randstate.o numtheory.o

all: keygen encrypt decrypt

keygen: keygen.o $(COMMON_OBJS)
	$(CC)-o$@$^$(LDFLAGS)$(CFLAGS)

encrypt: encrypt.o $(COMMON_OBJS)
	$(CC)-o$@$^$(LDFLAGS)$(CFLAGS)

decrypt: decrypt.o $(COMMON_OBJS)
	$(CC)-o$@$^$(LDFLAGS)$(CFLAGS)

%.o:%.c
	$(CC)$(CFLAGS)-c$<

clean:	rm-fkeygen encrypt decrypt *.o

cleankeys:
	rm-f*.{pub,priv}

format:
	clang-format -i-style=file *.[ch]
```

Encryption
encrypt.c

./encrypt [-hv] [-i infile] [-o outfile] -n pubkey -d privkey

I included these header files and libraries in order to run my implementation.

```
#include "rsa.h"
#include "numtheory.h"
#include "randstate.h"
#include <unistd.h>
#include <inttypes.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <gmp.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdbool.h>
#include <limits.h>
```

Defines the option that take inputs:

```
#define OPTIONS "i:o:n:vh"
```

In the main function:

Define int opt = 0.
Then set the FILE infile and outfile to stdin and stdout.

Create the default public key file name.

Set the verbose to false so that you can later set it to true in switch case.

In a while loop, define your cases:

case'i': specifies the input file to encrypt (default: stdin)

Do this with fopen() and optarg.

case'o': specifies the output file to encrypt (default stdout)
Do this with fopen() and optarg.

case'n': specifies the file containing the public key (default rsa.pub)
Do this with optarg.

case'v': enables verbose output

case 'h': Displays a helpful message.

Default: In case everything else fails.

After the switch cases:

Open the public key file with fopen().
Make sure to account for the case that opening the file failed. Prints a help message as well.

Then, initialize a variables that will hold the username once the public key file has been read.
char username[_POSIX_LOGIN_NAME_MAX];

Create and initialize the variables that will later be used in rsa_read_pub():
m, s, n, e

Read the public key from the opened public key file.
Do this with rsa_read_pub()

Then, create an if statement for the verbose options.
Use mpz_sizeinbase(t, int) to calculate the size of each variable passed in.
Print the username read from the public file.
Print out the size and base of s.
Print out the size and base of n.
Print out the size and base of e.

Ex.

```
printf("s (%zu bits) = ", mpz_sizeinbase(s, 2));
gmp_printf("%Zd\n", s);
```

Then, inorder to verify the signature of the user, convert the username into an mpz_t with mpz_set_str(). Be sure this is in base 62.

Call rsa_verify() in order to ensure the correct user is encrypting.
If rsa_verify returns false, return 1 and exit. If not, continue to encrypt the file.

rsa_encrypt_file(infile, outfile, n, e);

Then close all the files you opened with fclose() and clear all your variables.

```
fclose(infile);
fclose(outfile);
fclose(pbfile);

mpz_clears(m, s, n, e, NULL);
```

Decryption

decrypt.c

./decrypt [-hv] [-i infile] [-o outfile] -n pubkey -d privkey

I included these header files and libraries in order to run my implementation.

```
#include "rsa.h"
#include "numtheory.h"
#include <unistd.h>
#include <inttypes.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdlib.h>
#include <gmp.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdbool.h>
```

Defines the option that take inputs:

```
#define OPTIONS "i:o:n:vh"
```

In the main function:

Define int opt = 0.
Then set the FILE infile and outfile to stdin and stdout.

Create the default private key file name.

Set the verbose to false so that you can later set it to true in switch case.

In a while loop, define your cases:

case'i': specifies the input file to decrypt (default: stdin)
Do this with fopen() and optarg.

case'o': specifies the output file to decrypt (default stdout)
Do this with fopen() and optarg.

case'n': specifies the file containing the private key (default rsa.pub)
Do this with optarg.

case'v': enables verbose output

case 'h': Displays a helpful message.

Default: In case everything else fails.

After the switch cases:

Open the private key file with fopen().
Make sure to account for the case that opening the file failed. Prints a help message as well.

Create and initialize the variables that will later be used in rsa_read_priv():

```
mpz_t n, e;
```

Read the public key from the opened public key file.
Do this with rsa_read_priv()

Then, create an if statement for the verbose options.
Use mpz_sizeinbase(t, int) to calculate the size of each variable passed in.
Print out the size and base of n.
Print out the size and base of e.

Then decrypt the file.

```
rsa_decrypt_file(infile, outfile, n, e);
```

Clear all your variables and close all your files.

```
mpz_clears(n, e, NULL);
fclose(infile);
fclose(outfile);
fclose(pvfile);
```

Type equation here.