Assignment 5

James Contini

2022 October 26

General information

Citations

None.

Description

This design is split into three parts. Numtheory, encrypt and decrypt. Numtheory pertains to all the keygen functions and keygen itself while encrypt and decrypt pertain to all of the rsa reading and writing functions.

1 Miscellaneous

1.1 *randstate init()*

The parameter for this function is parsed from the command line. It is a number which is set as the global seed for all deterministic random number generating functions.

1.2 *rand clear()*

This function clears the global random seed variable.

2 Numtheory

2.1 *pow mod()*

This function takes four parameters, an out, a base, an exponent and a modulus. Using the fact that any integer, can be represented as a polynomial, we can calculate the exponent out of a sum of base two polynomials. Similar to how decimal numbers are represented in binary. The function keeps calculating powers within *mod(n)* until we've looped all the way to the original exponent in *mod(n)* range.

2.2 *is prime()*

This function uses the Miller-Rabin primality test. For n iterations of checking the number against a witness, the test has a $1/4^{-n}$ chance of being wrong. Therefore the more iterations the more likely, the pseudo-prime will be truly prime. To find s and r we have to make some ameliorations to the equation. To start lets re-write the equation so $\frac{n-1}{2^s} = r$. Now lets iterate s until r is odd. And because we are dividing n-1 by a some power of two, if n-1 is odd, the function will never find an integer r. Thus at the top of the function we must check first to see if n is even, and if it is return 0.

2.3 *make prime()*

This functions takes three parameters *mpz o*, *nbits* and *iters* iterations. The gmp function *mpz urandomb* allows us to make a number $2^n - 1$ which will be exactly. Then we will mask the most significant bit and the first bit to guarantee a random number of nbits bits that is odd. After we'll use *is prime()* to check if the number is indeed prime and loop until making one of them is prime.

2.4 *gcd()*

This function calculates the greatest common divisor of two numbers using the Euclidean Algorithm. Essentially while one of the numbers b is not zero take the remainder of other number a mod b until the remainder b is zero. And every loop new a is old b.

2.5 keygen.c

This function parses options using *getopt()*. I used an uint8 bit set to keep track of my options. I would set bits past bit one if there was some kind of error. And Verbose options was represented by the first bit so I knew if my set v equaled one then verbose had been chosen but if v was greater than one it meant no matter what I had to print the help/synopsis. The rest of keygen as a main function is rather uninteresting and straightforward.

3 encrypt

3.1 *rsa make pub()*

A public key n is be made by multiplying two large primes from make prime(). Prime p is a random number within the range 1/4 *nbits* to 3/4 *nbits* and q is the difference between n and p. In my implementation, if *nbits* is not divisible by four there is a possibility that p p may be out of range because any number not divisible by four will be rounded down. To offset this, if nbits is not divisible by four and the random number to be added to it is zero. Then I add nbits modulo four to round up. Next I choose 65537 as the public exponent because its prime and $2^16 + 1$ which is useful.

3.2 *rsa write pub()*

This writes the public keys to a .pub file. First we open a file and then get the public key in hexstrings by calling *mpz get str()* and formatting their strings into an fprintf statment.

3.3 *rsa read pub()*

This function parses the public key components to be used for encryption. I used a counter to keep track of which line the program is on and *mpz set str() getline()* to set each mpz. To set the username I had to realloc the username to the length of the username in the file.

3.4 *rsa encrypt file()*

This parses chunks sized (log2(n) - 1) bytes from an infile. A piece of heap of that size is allocated to handle that many bytes. A chunk is read using *fread()* to the block. *fread()* returns the bytes it read and transferred. I use this value j, as the count for *mpz import()*. The block is overwritten again and again until j is less than (log2(n) - 1) - 1 at which point the block was not fully loaded with bytes. Thus only j bytes of block will be imported and turned into hexstring and written to the ciphertext file... at which point the loop ends because j less than (log2(n) - 1) - 1. However the count seemed to not be working for *mpz import()* so to outsmart this bug, just before the function writes its last hexstring, at j + 1 index of the block, I put a null terminator so that the decrypt function's fprintf doesn't print the junk out. Also I made sure the padding wasn't overwritten by using pointer arithmatic to say block begins at block + 1.

3.5 *rsa encrypt()*

One chunk of data can be encrypted at a time. The data is of size n. The encryption is performed by raising the data to the public exponent e and then taking the modulus by n. This will render c, the ciphertext which will be accessible in the form of a mpz t variable.

3.6 encrypt.c

In encrypt.c I also used an 8 bit unsigned integer set to keep track of my command options. The output basically comes down to four combinations made of two options input from stdin/a file and output to stdout/a file. Using if statments and the set I would print and take input from one of those options accordingly.

4 decrypt

4.1 rsa make priv()

This function calculates the private key using the Euler-Fermat theorem to come up with a private key d that when multiplied to e in the exponent will neutralize it and turn C back into M.

4.2 *rsa write priv()*

This function is similar to *rsa write pub()*, it just writes the private key in the exact same way but to a .priv file which will be used for decryption.

4.3 *rsa read priv()*

The same way *rsa read pub()* parses hextrings using a counter and *mpz set str()* is done here

4.4 rsa decrypt file()

This function does about the same thing as *rsa encrypt file()* but backwards. But this time the block is of the previous size plus one byte to account for a null terminator. This function also parses the infile using *getline()* like *rsa read pub()* and *priv*. Each line is read *rsa decrypted()* and printed to outfile. Also I make sure to print starting at block + 1 to avoid the padding.

4.5 rsa decrypt()

Similar to rsa encrypt() this function assumes a chunk sized n has been fed to it. The function will then compute the original message M using the private key d to neutralize the public exponent e.

4.6 decrypt.c

For decrypt.c I literally copied and then pasted encrypt.c and then changed where the output was going to and the public file related variables' names. And the last difference is that it prints private key d and public n with command option -v instead.