Acer Cristea
Cruzid: acristea
Winter 2023 - CSE 13S
Dr. Veenstra

Assignment 5 DESIGN

Description of Program

In this assignment, we will be creating three programs. A keygen program that will be in charge of key generation, producing SS public and private key pairs. An encrypt program that will encrypt files using a public key. A decrypt program that will decrypt the encrypted files using the corresponding private key. We also need to implement two libraries, one holding functions for the mathematics of SS, and the other containing implementation of routines for SS. We also need a library for the GNU multiple precision arithmetic library.

NOTE: You may not use any GMP-implemented number theoretic functions. You must implement those functions yourself.

Key Generator

Your key generator program should accept the following command-line options:

- -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.

The program should follow these steps:

1. Parse command-line options using getopt() and handle them accordingly.

2. Open the public and private key files using fopen(). Print a helpful error and exit the program in the event of failure.

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.

4. Initialize the random state using randstate_init(), using the set seed.

5. Make the public and private keys using ss_make_pub() and ss_make_priv(), respectively.

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

7. Write the computed public and private key to their respective files.

8. If verbose output is enabled print the following, each with a trailing newline, in order:
    a. username
    b. the first large prime p
    c. the second large prime q
    d. the public key n
    e. the private exponent d
    f. the private modulus pq

All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal.

9. Close the public and private key files, clear the random state with randstate_clear(), and clear any mpz_t variables you may have used.

Encryptor

Your encryptor program should accept the following command-line options:

- 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.

The program should follow these steps:

1. Parse command-line options using getopt() and handle them accordingly.

2. Open the public key file using fopen(). Print a helpful error and exit the program in the event of failure.

3. Read the public key from the opened public key file.

4. If verbose output is enabled print the following, each with a trailing newline, in order:
    a. username
    b. the public key n

All of the mpz_t values should be printed with information about the number of bits that constitute them, along with their respective values in decimal.

5. Encrypt the file using ss_encrypt_file().

6. Close the public key file and clear any mpz_t variables you have used.

Decryptor

Your decryptor program should accept the following command-line options:

- specifies the input file to decrypt (default: stdin).
- specifies the output file to decrypt (default: stdout).
- specifies the file containing the private key (default: ss.priv).
- enables verbose output.
- displays program synopsis and usage.

The program should follow these steps:

1. Parse command-line options using getopt() and handle them accordingly.

2. Open the private key file using fopen(). Print a helpful error and exit the program in the event of failure.

3. Read the private key from the opened private key file.

4. If verbose output is enabled print the following, each with a trailing newline, in order:
    a. the private modulus pq
    b. the private key d

Both these values should be printed with information about the number of bits that constitute them, along with their respective values in decimal. See the reference decryptor program for an example.

5. Decrypt the file using ss_decrypt_file().

6. Close the private key file and clear any mpz_t variables you have used

The Deliverables
(files that need to be included in the directory "asgn4")

1. decrypt.c contains the implementation and main() function for the decrypt program.

2. encrypt.c contains the implementation and main() function for the encrypt program.

3. keygen.c contains the implementation and main() function for the keygen program.

4. numtheory.c contains the implementation of the number theory functions.

5. numtheory.h specifies the interface for the number theory functions.

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

7. randstate.h specifies the interface for initializing and clearing the random state.

8. ss.c contains the implementation of the SS library.

9. ss.h specifies the interface for the SS library.

10. Makefile:
    - CC = clang must be specified.
    - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified.
    - pkg-config to locate compilation and include flags for the GMP library must be used.
    - make must build the encrypt, decrypt, and keygen executables, as should make all.
    - make decrypt should build only the decrypt program
    - make encrypt should build only the encrypt program
    - make keygen should build only the keygen program.
    - make clean must remove all files that are compiler generated.
    - make format should format all your source code, including the header files.

11. README.md: This must use proper Markdown syntax. It must describe how to use your program and Makefile. It should also list and explain any command-line

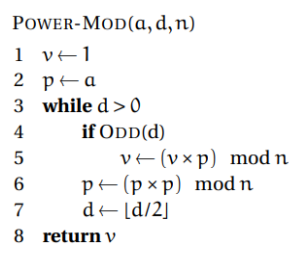
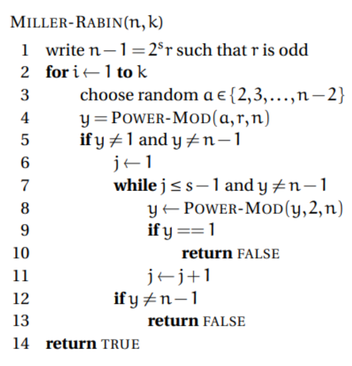
options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

12. DESIGN.pdf: What I am writing right now.

13. WRITEUP.pdf: This document must be a proper PDF. This writeup document must include everything you learned from this assignment. Make sure to mention everything in detail while being as precise as possible. You should also discuss the applications of public-private cryptography and how it influences the world today. Be sure to describe at least one way in which you personally take advantage of public-private cryptography on a day-to-day basis.

Pseudocode / Structure

*Code will be spread out across multiple files

Number Theoretic Functions $\prod_{0\le i\le m} a^{c_i 2^i}$ 	void pow_mod(mpz_t out, mpz_t base, mpz_t exponent, mpz_t modulus) - Performs fast modular exponentiation, computing base raised to the exponent power modulo modulus, and storing the computed result in out. - We need this a part of a step for SS
	bool is_prime(mpz_t n, uint64_t iters) - Conducts the Miller-Rabin primality test to indicate whether or not n is prime using iters number of Miller-Rabin iterations. This function is needed when creating the two large primes p and q in SS, verifying if a large integer is a prime. In addition to the is_prime() function, you are also required to implement the following function:

```
POWER-MOD($a, d, n$)
$v \leftarrow 1$
$p \leftarrow a$
while $d > 0$
    if ODD($d$)
        $v \leftarrow (v \times p) \mod n$
    $p \leftarrow (p \times p) \mod n$
    $d \leftarrow \lfloor d/2 \rfloor$
return $v$
```

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

GCD(a, b) **while** $b \neq 0$ $t \leftarrow b$ $b \leftarrow a \bmod b$ $a \leftarrow t$ **return** a MOD-INVERSE(a, n) $(r, r') \leftarrow (n, a)$ $(t, t') \leftarrow (0, 1)$ **while** $r' \neq 0$ $q \leftarrow \lfloor r/r' \rfloor$ $(r, r') \leftarrow (r', r - q \times r')$ $(t, t') \leftarrow (t', t - q \times t')$ **if** $r > 1$ **return** no inverse **if** $t < 0$ $t \leftarrow t + n$ **return** t	void make_prime(mpz_t p, uint64_t bits, uint64_t iters) - 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 number of iterations Euclid's algorithm, is an efficient method for computing the greatest common divisor (gcd) of two integers, the largest number that divides them both with a zero remainder void gcd(mpz_t d, mpz_t a, mpz_t b) - Computes the greatest common divisor of a and b, storing the value of the computed divisor in d. a and n are coprime if and only if there exist integers s and t such that ns+at = 1 void mod_inverse(mpz_t i, mpz_t a, mpz_t n) - Computes the inverse i of a modulo n. In the event that a modular inverse cannot be found, set i to 0. Note that this pseudocode uses parallel assignments, which C does not support. Thus, you will need to use auxiliary variables to fake the parallel assignments
SS Library	void ss_make_pub(mpz_t p, mpz_t q, mpz_t n, uint64_t nbits, uint64_t iters) - This creates a new SS public key, using two large prime numbers (p and q) and n computed like p*p*n. We have to create primes using a function make_prime(). The number of bits for the prime number p be a random number in the range

	[nbits/5,(2×nbits)/5]. The bits from p will be contributed to n twice, the remaining bits will go to q. Iters is obtained using random() and checking that p does not divide by q-1 and q does not divide by p-1. void ss_write_pub(mpz_t n, char username[], FILE *pbfile) - Writes a public SS key to pbfile. Formathould be n, then the username, each with a newline. N should be a *hextring* (GMP). 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. To compute d, compute the inverse of n modulo $\lambda(pq) = \text{lcm}(p-1,q-1)$. void ss_write_priv(mpz_t pq, mpz_t d, FILE *pvfile) - Writes a private key to pvfile - Format, pq then d both followed by newline, written in hextring. void SS_read_priv(mpz_t pq, mpz_t d, FILE *pvfile) - Read a private SS key from pvfile. Same format as write void ss_encrypt(mpz_t c, mpz_t m, mpz_t n) - Performs SS encryption, computing the ciphertext c, encrypting message m using public key n. void ss_encrypt_file(FILE *infile, FILE *outfile, mpz_t n) - Encrypts contents of infile, writing contents to outfile. Encrypt data in blocks, not the whole file as we're working with modulo n.

- The value of a block cannot be 0: E(0) ≡ 0 ≡ 0 n (mod n).
- The value of a block cannot be 1. E(1) ≡ 1 ≡ 1 n (mod n).

- Calculate the block size k. This should be k = b(log2 (p n)−1)/8c.

- Dynamically allocate an array that can hold k bytes. This array should be of type (uint8_t *) and will serve as the block.

- Set the zeroth byte of the block to 0xFF. This effectively prepends the workaround byte that we need

- While there are still unprocessed bytes in infile:
    - Read at most k−1 bytes in from infile, and let j be the number of bytes actually read. Place the read bytes into the allocated block starting from index 1 so as to not overwrite the 0xFF.
    - Using mpz_import(), convert the read bytes, including the prepended 0xFF into an mpz_t m. You will want to set the order parameter of mpz_import() to 1 for most significant word first, 1 for the endian parameter, and 0 for the nails parameter.
    - Encrypt m with ss_encrypt(), then write the encrypted number to outfile as a hexstring followed by a trailing newline

void ss_decrypt(mpz_t m, mpz_t c, mpz_t d, mpz_t pq)

- Performs SS decryption, computing message m, decrypting ciphertext c using private key d and public modulos n.

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

	- Decrypts the contents of infile, writing the decrypted contents to outfile. The data in infile should be decrypted in blocks to mirror how ss_encrypt_file() encrypts in blocks. To decrypt a file, follow these steps: - Need to dynamically allocate an array that can hold k = (log2 (pq) −1)/8 rounding down bytes. Type (uint8_t *) - Iterating over the lines in infile: - Scan hextring, saving it as c. - Decrypt c back into it's original value m. Use mpx_export(), convert m back into bytes, storing them into block. J is the number of bytes converted. - Write out j−1 bytes starting from index 1 of the block to outfile. This is because index 0 must be prepended 0xFF. Do not output the 0xFF
Step 2: Creating the Makefile	Since no Makefile is provided, we need to make it ourselves to be able to compile and run our files. 1. Make sure to use CFLAG and CLANG 2. pkg-config to locate compilation and include flags for the GMP library must be used.
Making Keygen, Encrypt, and Decrypt	Follow the steps above to create these files.