Design Document - Public Key Cryptography

Lev Teytelman

February 6, 2022

1 Program Description

This assignment consists of three main programs:

- `keygen` - Creates public/private keys
- `encrypt` - Encrypts files using an RSA key
- `decrypt` - Decrypts files with the specified key

2 Included Files

- decrypt.c - The implementation of the `decrypt` program (contains a `main` function, flag processing, and file decryption).
- encrypt.c - The implementation of the `encrypt` program - also has a `main` function and flag processing; encrypts files.
- keygen.c - The implementation of the `keygen` program - generates RSA public/private keys, has a `main` function, and does flag processing.
- numtheory.c - Contains various useful functions for key creation and file processing.
- numtheory.h - The declaration of the number theory functions, to be included and used in other files.
- randstate.c - The implementation of random number generation functions; mostly used for setting the seed and ensuring no memory leaks.
- randstate.h - The declaration of the functions to be included and used in other files.
- rsa.c - The implementation of a variety of RSA-related functions, including key generation and file encryption and decryption functions.
- rsa.h - The declarations of the RSA functions, used for inclusion in other files.

- Makefile - The logic to compile and link the files into a binary.
- README.md - The file describing program compilation, usage, and flags.
- DESIGN.pdf - This file, describing the general structure of certain functions to be implemented.

3 Structure

This assignment has three main executable programs, each with their own sets of flags. The `keygen` program allows the user to specify the number of bits to use for the public modulus, the number of iterations for testing primes, the output files for the public and private keys, and a verbose flag to print the values to standard output. Both `encrypt` and `decrypt` have a similar set of flags: optional input and output file specification, a flag to specify the location of the public/private key, and a verbose flag. The `keygen` program will generate randomized prime numbers and a public exponent, and use them to create a public and private key, writing to the specified file names and modifying the permissions as necessary for the private key. The `encrypt` and `decrypt` programs will encrypt and decrypt files using the provided public/private keys, respectively.

4 Pseudocode

Encrypting and decrypting files will be an interesting process: the data will need to be converted into numbers and subsequently encoded using the public key, and decoded with the private key and converted back to the original data when decrypting. Encryption is done using the `pow_mod` function with $c \equiv m^e \bmod n$, where m is the message to encrypt and c is the encrypted message. However, this won't be able to encrypt an infinite-size number, as the modulus would make the result overflow; which is why when encrypting files we will break it into smaller pieces and encrypt those instead. Our decryption function will go in the opposite direction, taking each encrypted block from the file and decrypting it with the private key ($m \equiv c^d \bmod n$, where m is the message, c is the encrypted message, and d is the private key). Going by the assignment's description of the functions, the code in `rsa.c` will look similar to the following:

```
def encrypt_file():
    k = block_size
    data = ""
    j = 0
    array = []
    while ((data = read(input, 1) != EOF):
        array[++j] = data
        if (j == k - 1):
```

```
            mpz_import(m, array)
            rsa_encrypt(m)
            fwrite(output, m)
def decrypt_file():
    k = block_size
    mpz_t c
    array = []
    while ((c = read(input, 1) != EOF):
        rsa_decrypt(c)
        mpz_export(c, array)
        fwrite(output, array)
```

We will need to ensure that the private key file's permissions are properly set, which we can do using `fchmod` and octal representation (done by starting a number with a 0):

```
    fchmod(pvfile, 0600)
```

When writing GMP integers to file/standard output, we will need to use special formatting functions and characters; the documentation provides guidance on formatting the large integers in strings:

```
    gmp_fprintf(file, "%Zd\n", n); // will print the number in decimal
    gmp_fscanf(file, "%Zx\n", n); // will scan the number in as hex
```

These special formatting strings are necessary as the `mpz_t` types are not normally in C and therefore do not work in default `printf` formatting. In order to ensure no memory leaks and efficient memory usage, the `mpz_t` variables will need to be properly initialized and cleared before and after use; this will also need to be done in a smart way so they're not being initialized and cleared over and over again. We don't want to initialize or clear inside of a loop, such as a while or for loop; we want to do that at the start of the function, and clear at the end once we're done using them:

```
def function():
    mpz_t a, b, c
    initiate(a, b, c)
    for i in range(100):
        pow_mod(a, b, c)
    clear(a, b, c)
```

This way, we only create the variables when using them, but we ensure they're not initialized and cleared 100 times, as that is inefficient.