# Assignment 5: Public Key Cryptography Writeup

Shirin Rokni
shrokni

November 3, 2022

---

### Citations

This PDF was generated using a LaTeXtemplate made by Jessie Srinivas (CSE13s tutor).

---

## Purpose

This assignment creates three programs called keygen, encrypt and decrypt. Keygen is a program that is in charge of key generation, producing RSA public and private key pairs. Security requires keeping the private key private, and the public key can be distributed widely. Any person can encrypt a message using the intended receiver's public key, but that encrypted message can only be decrypted with receivers private key. The encrypt program will encrypt files using a public key. The decrypt program will decrypt the encrypted files using the corresponding private key which consists of the public modulus and public exponent.

Testing each file and each function within them is integral in compartmentalizing your code and finding where potential errors are coming from. Testing also aids in understanding how each function works and how it will work with it's environment. Thus, implementing test files and functions are essential in this assignment.

---

## 1 Testing numtheory

### 1.1 File generation and Makefile changes

In order to properly test the functionality of the various numtheory functions in numtheory.c, I made a file named numtheory-test.c In it, there are several headers including the numtheory.h library in order to call the functions and the randstate.h library in order to generate random values. Then, I defined the randstate using gmp_randstate_t in order to access the state variable.

Then, I defined my main function. In it, I would first initialize the randstate by using randstate_init with a random value in the parameter. Then I would have various chunks of code representing the test of various functions.

For example, in order to test if gcd is functionally effective and correct, I initialized some random variables x, y and z. Then I set them to unsigned integers like 5, 10 and 15. I would then call the gcd() function with x, y and z as the arguments. Finally, I would print what x is using gmp_printf and compare the output with an online calculator. I would use such steps to test mod_inverse, pow_mod, make_prime, and is_prime.

However, in order to run this file, I would need to include it in the Makefile, so I altered the Makefile to allow me to run this file. In the Makefile, I added the numtheory-test executable to the all and clean sections. Then, I made a section for numtheory-test which needs numtheory-test.o, numtheory.o and randstate.o. I would then include the necessary clang and lflag portions below. This step allows me to make and clear numtheory-test and properly test the functionality of numtheory.c.

---

# 2 Testing rsa

## 2.1 File generation and Makefile changes

In order to properly test the functionality of the various rsa functions in rsa.c, I made a file named rsa-test.c In it, there are several headers including the rsa.h library in order to call the functions and the randstate.h library in order to generate random values. I would also call the numtheory.h library since several functions in rsa.c use its functions. Then, I defined the randstate using gmp_randstate_t in order to access the state variable.

Then, I defined my main function. In it, I would first initialize the randstate by using randstate_init with a random value in the parameter. Then I would have various chunks of code representing the test of various functions.

for example in order to check if rsa_write_pub is functionally effective and correct, I initialized some random variables n, e, and s. I also defined a character type variable named 'user' which was set to some random string such as 'shirin'. Then, I set n, e and s to random unsigned integers such as 100, 150 and 103. Then, I defined another variable 'files' which is a FILE type and is set to fopen(). The arguments of fopen are an already existing yet empty file named 'files.dat' and the mode 'w+' so that the file can be read and written from. Finally, I called rsa_write_pub with the arguments n, e, s, user and files. Then, I would go into the 'files.dat' file and determine if anything was written to the file and if it was the appropriate contents (if the first three items are integers and the last one a string). I would use such steps to test the other various rsa functions - by creating variables, files and characters to input into the functions and then accessing them later to deem if they work.

However, in order to run this file, I would need to include it in the Makefile, so I altered the Makefile to allow me to run this file. In the Makefile, I added the rsa-test executable to the all and clean sections. Then, I made a section for rsa-test which needs rsa-test.o, rsa.o, randstate.o and numtheory.o. I would then include the necessary clang and lflag portions below. This step allows me to make and clear rsa-test and properly test the functionality of rsa.c.

---

# 3 Testing keygen

## 3.1 Command-line Checks

The keygen file itself has various checks and systems in place for the implementor to ensure that their file is running properly. One such check is to simply run the verbose flag and see if the number of bits and output match the input. If the user specified that the bit value should be 100, then the information about the number of bits for each mpz_t value should reflect similar numbers.

Another check is to manually go into the 'rsa.pub' and 'rsa.priv' files that the function changes. If, when you visit these files, they are filled with a string of randomly generated values and numbers, then it can be safely assumed that the functionality of keygen is accurate.

---

# 4 Testing encrypt

## 4.1 Command-line checks

One resource that is readily available to us is an executable for keygen, encrypt and decrypt named 'keygen-dist', 'encrypt-dist', and 'decrypt-dist'. This resource is incredibly useful in testing the functionality and successful passing of our own encrypt file. One such way is to generate a key with 'keygen-dist' and use it with our own encrypt program and see if no helpful error messages were brought up. Then, we can access the file used as the output file and see if the contents of the file appear to be encrypted. We can assume that it is if there are various values and variables that are nonsensical.

---

# 5 Testing decrypt

## 5.1 Command-line checks

One resource that is readily available to us is an executable for keygen, encrypt and decrypt named 'keygen-dist', 'encrypt-dist', and 'decrypt-dist'. This resource is incredibly useful in testing the functionality and successful passing of our own decrypt file. One such way is to generate a key with 'keygen-dist' and use it with our own decrypt program and see if no helpful error messages were brought up. Additionally, we can use the reference encrypt to encrypt a file, and then use our own decrypt program to decrypt it with the same key.

Another way to test the funtionality and succesful passing of the decrypt program is to use it with our own already-existing programs. We can create a key with ./keygen, encrypt and command-line written message with ./encrypt, copy and paste the message into decrypt (using the same private key value) and see if the decrypted message is the same as what was encrypted.

---