Neil Bisht
nbisht@ucsc.edu
20 November 2021

CSE13S Fall 2021
Assignment 6: “Public Key Crypotgraphy”
DESIGN.pdf

Key:

-A GREEN “**” indicates integral pieces of pseudocode that will be explained further in the Discussion section

Project Description:

The goal of this project, “Public Key Crypotgraphy”, is to create a public and private key generator, an encryptor, and a decryptor. We will be implementing this asymmetric cryptography method with the RSA algorithm, named after its founders Adhi Shamir, Ronald Rivest, and Leonard Adelmen. The security of this encryption method depends on the difficulty of factoring a very large product of prime numbers into its constituents. RSA cryptography involves public and private keys. The idea behind the encryption is to encrypt a message with a receiver’s public key which can only be decrypted by a receiver’s private key. For that reason, private keys live up to its namesake and should be kept private, while public keys can be known to the public. The mathematics behind RSA are based on modulo arithmetic and an extremely large composite number **n = pq**, where **p** and **q** are two large prime numbers. Our encryption algorithm can be deduced to $\mathbf{E(M) = M^e \pmod n = C}$, while our decryption algorithm can be deduced to $\mathbf{D(C) = c^d \pmod n = M}$. The variable **M** in this case would be a message where **M** raised to the totient of a natural number **n** is congruent to **1 (mod n)**. An important property of these two algorithms is that they’re mutual inverses, meaning that: **E(D(x)) = D(E(x)) = x**. This will prove to be useful because it allows us the ability to provide *digital signatures*. Our key generator implementation will be provided in “keygen.c”. Our encryptor and decryptor implementations will be provided in “encrypt.c”, with its interface defined in “encrypt.h”, and “decrypt.c”, with its interface defined in “decrypt.h” respectively. The mathematical functions that will be required for our encryption and decryption methods to work are implemented in “numtheory.c”, with its interface

defined in “numtheory.h”. In addition, the functions that will be performing the RSA encryption and decryption will be provided in “rsa.c”, with its interface defined in “rsa.h”. Finally, “randstate.c” will contain our implementation for the random state interface with its interface defined in “randstate.h”. The purpose of “randstate.c” is to initialize and clear the random state interface necessary to generate large random numbers. We will be using the GNU multiple precision arithmetic library, referred to as GMP, to support arbitrary precision integers. This is necessary because **C** does not natively support arbitrary precision despite us relying on extremely large numbers for this project.

Pseudocode:

**-numtheory.c:

void pow_mod()

```
# Performs modular exponentiation
def pow_mod(base, exponent, modulus):
   v = 1
   p = base
   # While the exponent is greater than 0...
   while(exponent > 0):
       # If the exponent is odd...
       if (exponent % 2) != 0:
           v = ((v * p) % modulus)
       p = ((p * p) % modulus)
       # Floor division
       exponent = (exponent // 2)
   return v
```

bool is_prime()

```
# An implementation of Miller-Rabin's primality test where a number can definitely
# be said to be not-prime or probably prime.
# The more iterations there are to this test, the likelier it will be accurate.
def is_prime(n, iters):
   # Finds an odd r that satisfies this equation
   n - 1 = 2^s * r
   for i in range(1, iters):
       choose a random number a in {2, 3,...,(n - 2)}
       # Performs modular exponentiation with base a, exponent r, and mod n
       y = pow_mod(a, r, n)
       if y != 1 and y != (n - 1):
           j = 1
           while j <= (s - 1) and y != (n - 1):
               # Performs modular exponentiation with base a, exponent r, and mod n
               y = pow_mod(y, 2, n)
               if y == 1:
                   return False
               if y != (n - 1):
                   return False
   return True
```

void gcd()

```
# Finds the greatest common divisor between two numbers a and b
def gcd(a, b):
   while (b != 0):
       temp = b
       b = a % b
       a = temp
   return a
```

void mod_inverse()

```
# Computes the modular inverse of a mod n
def mod_inverse(a, n):
   (r, r_prime) = (n, a)
   (t, t_prime) = (0, 1)
   while r_prime != 0:
       # Floor division
       q = r // r_prime
       (r, r_prime) = (r_prime, r - q * r_prime)
       (t, t_prime) = (t_prime, t - q * t_prime)
   # If r is greater than 1, return no inverse.
   # Returning no inverse should mean setting the output variable to 0.
   if r > 1:
       return no inverse
   if t < 0:
       t = t + n
   return t
```

void make_prime()

```
# Generates a prime number that is at least "bits" numbers of bits long.
# Stores the prime number in "p".
def make_prime(p, bits, iters):
   # Generate a random number from 0 - 2^n-1 inclusive.
   # While that number isn't prime OR its size in bits is less than "bits",
   # continue generating a number until it's  prime and at least "bits" number of bits
long.
   do {
       generate random number p at least “bits” number of bits long
   } while ((p is not prime) or (size of bits < bits))
   return
```

**-rsa.c:

void rsa_make_pub()

```
def rsa_make_pub(p, q, n, e, nbits, iters):
   # Calculate the prime numbers p and q.
   # p is calculated with a number of bits inbetween the range
   [nbits / 4, (3 × nbits) / 4).
   # The rest of the bits go to q.
   p = prime number with a number of bits inbetween range [nbits / 4, (3 × nbits) / 4)
   q = prime number with the remaining number of bits

   # Calculates the totient of n.
   totient = (p - 1)(q - 1)

   # Find the public exponent.
   do {
       e = random number that
       gcd  = gcd(e, totient);
   } while (gcd != 1);
```

void rsa_write_pub()

```
# Writes the public key to pbfile in the order n, e, s, and the username, each of
which have a trailing new line after.
# n, e, and s are written as hexstrings.
# Each element has a trailing newline after.
def rsa_write_pub(n, e, s, username[], FILE pbfile):
   print (n \n, e \n, s \n username \n) to pbfile
```

void rsa_read_pub()

```
# Reads the public key from pbfile.
def rsa_write_pub(n, e, s, username[], FILE pbfile):
   scan (n \n, e \n, s \n username \n) from pbfile
```

void rsa_make_priv()

```
# Generates a private key "d" by computing the modular inverse of "e" mod totient(n).
# totient(n) = (p - 1)(q - 1) where "p" and "q" are prime numbers.
def rsa_make_priv(d, e, p, q):
   # Calculates the totient, totient(n) = (p - 1)(q - 1).
   totient = (p - 1)(q - 1)
   # Computes the inverse of e mod totient(n).
   mod_inverse(d, e, totient)
```

void rsa_write_priv()

```
# Writes the private key to pvfile in the order n, then d, each of which have a
trailing new line after.
# n and d are written as hexstrings.
# Each element has a trailing newline after.
def rsa_write_priv(n, d, FILE pvfile):
   print (n \n, d \n) to pvfile
```

void rsa_read_priv()

```
# Reads the private key from the file pvfile.
def rsa_read_priv(n, d, FILE pvfile):
   scan (n \n, d \n) to pvfile
```

void rsa_encrypt()

```
# Encrypts a message "m" by taking "m" to the power of the public exponent "e" mod
"n".
# Stores the ciphertext in "c".
def rsa_encrypt(c, m, e, n):
   pow_mod(c, m, e, n)
```

void rsa_encrypt_file()

```
# Encrypts an infile in k byte blocks.
def rsa_encrypt_file(FILE infile, FILE outfile, n, e):

   # Calculate the block size k.
   k = floor(log_base_2(n) - 1) / 8)

   # Dynamically allocate an array that can hold k bytes.
   # Set the 0th byte to 0xFF.
   array = dynamically allocate array that can hold k bytes of type uint8_t
   array[0] = 0xFF

   # While we haven't reached EOF, read in k bytes at a time, import them into an
mpz_t, encrypt the mpz_t, then print to the outfile.
   while (unprocessed bytes in the infile):
       j = number of bytes read from infile (should be at most k - 1 bytes)
       m = mpz_import(j)
       rsa_encrypt(m)

       # m should be printed as a hexstring to the outfile with a trailing newline.
       print (m \n) to outfile
```

void rsa_decrypt()

```
# Decrypts a ciphertext "c" by taking "c" to the power of private exponent "d" mod
"n".
# Stores the message in "m".
def rsa_decrypt(m, c, d, n):
   pow_mod(m, c, d, n)
```

void rsa_decrypt_file()

```
# Encrypts an infile in k byte blocks.
def rsa_decrypt_file(FILE infile, FILE outfile, n, d):

   # Calculate the block size k.
   k = floor(log_base_2(n) - 1) / 8)

   # Dynamically allocate an array that can hold k bytes.
   array = dynamically allocate array that can hold k bytes of type uint8_t

   # While we haven't reached EOF, scan in hexstrings from infile, decrypt them,
export them as bytes into the array, then write them to the outfile.
   while (unprocessed bytes in the infile):
       # Scan in a hexstring "c" from the infile.
       scan (c) from infile
        m = rsa_encrypt(c)
        mpz_export(m)
        write to the outfile
```

void rsa_sign()

```
# Performs RSA signing.
def rsa_sign(s, m, d, n):
   pow_mod(s, m, d, n)
```

bool rsa_verify()

```
# Performs RSA verification.
def rsa_verify(m, s, e, n):
   pow_mod(t, s, e, n)
   # If t isn't the same as the expected message m, return false.
   if t != m:
       return False
   return True
```

**-keygen.c:

```
# Generates a public and private key, sending them to their specified outfiles.
# The public key will be sent to 'rsa.pub' by default.
# The private key will be sent to 'rsa.priv' by default.
main(int argc, char **argv):

   # Default values for the command line options.
   num_bits = 256
   mr_iters = 50
   seed = time(NULL)
   username[]
   verbose = false

   # Initialize all mpz_t variables that we'll be using in keygen.
   # p: prime number 1
   # q: prime number 2
   # n: product of p and q
   # e: public exponent
   # d: private key
   # user: username of type mpz_t
   # s: signature
   mpz_t p, q, n, e, d, s

   # Files for public and private keys.
   FILE pbfile
   FILE pvfile

   # Variables to store file names specified by the user.
   # The public key is sent to 'rsa.pub' by default.
   # The private key is sent to 'rsa.priv' by default.
   pbfile_name[] = "rsa.pub"
   pvfile_name[] = "rsa.priv"

   while ((opt = getopt(argc, argv, OPTIONS)) != -1):
       switch (opt):
       case 'b': num_bits = user specified bits
       case 'i': mr_iters = user specified number of iterations for Miller-Rabin
primality test
       case 'n': pbfile_name = user specified file
       case 'd': pvfile_name = user specified file
       case 's': seed = user specified seed
       case 'v': verbose = True
```

```
        case 'h': print manual page and end program


    # Opening of files to print the public and private keys to.
    open pbfile
    open pvfile

    # Setting the permissions of the private key file to 0600, or read and write
permissions for the USER ONLY.
    # Use fileno() to get file descriptor, then use chmod() to set the permissions.
    set pvfile permissions to 0600

    # Initialize the random state with the given seed.
    randstate_init(seed);

    # Make both public and private keys.
    rsa_make_pub(p, q, n, e, num_bits, mr_iters)
    rsa_make_priv(d, e, p, q)

    # Retrieve the user's username.
    # Use getenv()
    username = getenv(username)

    # Sign the username.
    rsa_sign(s, username, d, n)

    # Write both the public and private keys to their respective files.
    rsa_write_pub(n, e, s, username, pbfile)
    rsa_write_priv(n, d, pvfile)

    # If the user wants verbose output, print out all of the following to stdout...
    if (verbose):
        print username
        print s (signature)
        print p (prime number 1)
        print q (prime number 2)
        print n (public modulus)
        print e (public exponent)
        print d (private exponent)

    close pbfile and pvfile
    return 0
```

**-encrypt.c:

```
# Encrypts a file specified by the user, or stdin by default with a public key file.
# Sends the encrypted message to an outfile specified by the user, or stdout by
default.
int main(int argc, char **argv):

   username[]
   bool verbose = false

   # Initialize all mpz_t variables that we'll be using in encrypt.
   # n: product of p and q (public modulus)
   # e: public exponent
   # s: signature
   # user: username of type mpz_t
   mpz_t n, e, s, user

   # Sets default input and output to stdin and stdout respectively.
   FILE infile = stdin
   FILE outfile = stdout
   FILE pubkey

   # Variables to store file names specified by the user.
   infile_name
   outfile_name

   # The public key file is 'rsa.pub' by default.
   pubkey_name = "rsa.pub"

   while ((opt = getopt(argc, argv, OPTIONS)) != -1):
       switch (opt):
       case 'i': infile_name = user specified file name
       case 'o': outfile_name = user specified file name
       case 'n': pubkey_name = user specified file name
       case 'v': verbose = True
       case 'h': print manual page and end program

   # Opening of files...

   pubkey = open(pubkey_name)

   # Read the public key, username, and signature from pubkey.
   rsa_read_pub(pubkey)
```

```
  # If the user wants verbose output, print out all of the following to stdout...
  if (verbose):
      print(username)
      print(s)
      print(n)
      print(e)

  # Verify the signature.
  # If the signature isn't verified, throw an error and end the program.
  if (!rsa_verify()):
      exit program

  # Encrypt the infile and send the ciphertext to outfile.
  rsa_encrypt_file()

  close files
```

**-decrypt.c:

```
# Decrypts a file specified by the user, or stdin by default with a private key file.
# Sends the decrypted message to an outfile specified by the user, or stdout by
default.
int main(int argc, char **argv):

  bool verbose = false

  # Initialize all mpz_t variables that we'll be using in decrypt.
  # n: product of p and q (public modulus)
  # d: private key
  mpz_t n, d

  # Sets default input and output to stdin and stdout respectively.
  FILE infile = stdin
  FILE outfile = stdout
  FILE privkey

  # Variables to store file names specified by the user.
  infile_name
  outfile_name
```

```
# The private key file is 'rsa.priv' by default.
pubkey_name = "rsa.priv"

while ((opt = getopt(argc, argv, OPTIONS)) != -1):
    switch (opt):
    case 'i': infile_name = user specified file name
    case 'o': outfile_name = user specified file name
    case 'n': privkey_name = user specified file name
    case 'v': verbose = True
    case 'h': print manual page and end program

# Opening of files...

privkey = open(privkey_name)

# Read the private key from privkey.
rsa_read_pub(privkey)

# If the user wants verbose output, print out all of the following to stdout...
if (verbose):
    print(n)
     print(d)

# Decrypt the infile and send the message to outfile.
rsa_encrypt_file()

close files
```

Discussion:

This discussion will be explaining the parts of my pseudocode indicated by the "**" at the start of the block:

**-numtheory.c:

The purpose of "numtheory.c" is to hold the mathematical functions necessary for making RSA encryption possible; in essence, it's a math library with five mathematical functions. These functions include: **pow_mod()** for computing modular exponentiation, **is_prime()** for determining whether a number is prime or not, **gcd()** computing the greatest common divisor of two numbers, **mod_inverse()** for computing the modular inverse, and **make_prime()** for generating a random prime number that is at least *n* bits long.

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

It's extremely inefficient to calculate a^n by the method defined above. This is the reason why our **pow_mod()** function is necessary; within this function, we will be computing the modular exponentiation of a number *a* by an exponent *b* mod *n*. Through this method, we can compute a^n in logarithmic time rather than linear time. To implement **pow_mod()**, we utilize the fact that all integers can be written as a polynomial,

$$n = c_m 2^m + c_{m-1} 2^{m-1} + \cdots + c_1 2^1 + c_0 2^0 = \sum_{0 \le i \le m} c_i 2^i,$$

where *n* is greater than or equal to 2^m, and c_i is in the set {0, 1}.We can rewrite this formula as,

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

however, the numbers get extremely large very quickly. This is why we mod these numbers by a given number *n*, meaning that all numbers will be inbetween {0,...,(n - 1)}. For the function **is_prime()**, we implement the Miller-Rabin primality test whose accuracy depends on the number of iterations. It is a pseudoprime test, or in other words, it can only tell us whether a number is

probably prime, but can tell us whether a number is definitely *not prime*. That being said, if we run the Miller-Rabin primality test 100 times, then our chances of being wrong is $(¼)^{100} = 2^{-200}$; we can assume this is more than good enough. The function **gcd()** simply computes the greatest common divisor of two numbers using Euclid's Algorithm. Computing the modular inverse requires **mod_inverse()**, which involves the extended Euclidian Alogrithm which computes the gcd as well as the coefficients to Bezout's identity,

$$ax + by = \gcd(a, b)$$

After some arithmetic, this boils down to,

$$at \equiv 1 \pmod{n}.$$

Finally, our last math function **make_prime()**, which finds a prime out of randomly generated numbers that is at least *n* bits long and prime. This is implemented through a simple do-while loop and the use of our **is_prime()** function..

**-rsa.c:

The purpose of "rsa.c" is to hold a library of functions that perform RSA encryption. This library utilizes the mathematical functions necessary for RSA encryption that were defined in "numtheory.c". Each function serves a purpose in three major categories: public keys, private keys, or verification/signing. The function **rsa_make_pub()** makes a public key that can encrypt messages; the public key is defined as the pair <*e*, *n*>, where *e* is the public exponent, and *n* is the public modulus. By extension, the functions **rsa_write_pub()** and **rsa_read_pub()** write and read the public key to and from a file. The public key is written to an outfile in the order: *n*, *e*, *s*, *username*. Every element has a trailing newline, with *n*, *e*, and *s* being written as hexstrings. The functions **rsa_make_priv()**, **rsa_write_priv()**, and **rsa_read_priv()** follow the same semantics as the functions for public keys expressed above. In this case, however, we are dealing with private keys where *n* and *d* are written and read to and from files. Following this, the function **rsa_encrypt()** actually performs RSA encryption, computing the ciphertext *c* by encrypting a message *m*. Encryption is defined as,

$$E(m) = c = m^e \pmod{n},$$

so we use our **pow_mod()** function from “numtheory.c” to compute *c*. To encrypt a whole file, we made a function called **rsa_encrypt_file()** which utilizes our **rsa_encrypt()**. In this function, we read in and encrypt *k* bytes at a time. The functions **rsa_decrypt()** and **rsa_decrypt_file()** follow the same semantics as the encrypt functions mentioned above. Finally, the functions **rsa_sign()** and **rsa_verify()** perform RSA signing and verifying respectively. Both of these functions also utilize our **pow_mod()** function.

**-keygen.c:

This file is 1/3 of the main programs required for this project. Specifically, “keygen.c” contains the program to generate public and private keys. Each key will be sent to a different file specified by the user; if no files are specified, files “rsa.pub” and “rsa.priv” will be created by default and hold the public and private keys respectively.

**-encrypt.c:

This file is 1/3 of the main program required for this project. Specifically, “encrypt.c” contains the program to encrypt a file, or stdin by default. The generated ciphertext will be sent to a file specified by the user, or stdout by default.

**-decrypt.c:

This file is 1/3 of the main program required for this project. Specifically, “decrypt.c” contains the program to decrypt a file, or stdin by default. The generated message will be sent to a file specified by the user, or stdout by default.