Artie Nazarov
anazarov@ucsc.edu
11/9/21

CSE 13S Program 6.
DESIGN document

OVERVIEW

In Program 6 we will implement the RSA encryption algorithm for arbitrary data encryption and decryption. This algorithm is widely used today for secure data transmission. RSA heavily relies on the concept analogous to the *factoring problem*, stating that it is very difficult to efficiently factor the product of 2 very large numbers.

APPROACH

RSA encryption and decryption methods rely on the use of public keys and private keys, hence we will first implement a key generator program. In RSA a public key is used to encrypt a message, which can then only be decrypted using a private key. A private key is used for message decryption that was encrypted using a shared public key of the same user. We will then implement programs for message encryption and decryption using the RSA public and private keys.

TOP-LEVEL

randstate.c

```
// Using GMP library
RandomState state

// Initialize random state
void randstate_init(int seed):
    gmp_randinit_mt(state)
    gmp_randseed_ui(state, seed)

// Clear random state memory
void randstate_clear(void):
    gmp_randclear(state)
```

---

numtheory.c

```
// Using the GMP library
// Modular exponentiation function
void pow_mod(int out, int base, int exponent, int modulus):
	res = 1
	acc = base
	while exponent > 0:
		if exponent is ODD:
			res = (res*acc) mod modulus
		acc = (acc*acc) mod modulus
		exponent /= 2
	out = res

// Primality testing function
// Using Miller-Rabin test
bool is_prime(int n, int iters):
	write n - 1 = 2^(s) * r such that r is odd
	for i in [1..k]:
		choose random a in [2..n-2]
		y = primality testing determiner
		pow_mod(out = y, base = a, exponent = r, modulus = n)
		if y != 1 and n - 1:
			j = 1
			while j <= s - 1 and y != n-1:
				pow_mod(out = y, base = y, exponent = 2, modulus = n)
				if y == 1:
					return False
				j += 1
			if y != n-1:
				return False
	return True

// Generate a prime number
void make_prime(int p, int bits, int iters):
	Generate a random prime number
	Check if a randomly generated number is prime
	Check primality using is_prime()
```

```
// Compute the GCD of 2 numbers
void gcd(int g, int a, int b):
	while n != 0:
		t = b
		b = a mod b
		a = t
	Store result in variable g
	g = a

// Compute Mod-Inverse
void mod_inverse(int o, int a, int n):
	r, r' = n, a
	t, t' = 0, 1
	while r' != 0:
		q = floor(r/r')
		r, r' = r', r - q * r'
		t, t' = t', t - q * t'
	Update the value o
	if r > 1:
		o = 0
	if t < 0:
		t = t + n
	o = t
```

---

rsa.c

```
// Generate RSA public key
void rsa_make_pub(int p, int q, int n, int e, int nbits, int iters):
   generate p of size p_bits long (in bits), where p_bits in [nbits/4, (3*nbits)/4)
   generate q of size q_bits long (in bits), where q_bits = nbits - pbits

   totient of n = (p-1)(q-1)
   n = p*q

   // Compute public exponent e
   While e is not a valid exponent:
      generate a random number e of size nbits
      if gcd(e, totient of n) == 1:
         e is a valid exponent
```

```
void rsa_write_pub(int n, int e, int s, string username, file pbfile):
   Write n, e, s, username to pbfile

void rsa_read_pub(int n, int e, int s, string username, file pbfile):
   Read values n, e, s, username from pbfile

void rsa_make_priv(int d, int e, int p, int q):
   totient of n = (p-1)(q-1)
   // Compute Modular inverse
   mod_inverse(d, e, totient_n)

void rsa_write_priv(int n, int d, file pvfile):
   Write n, d to pvfile

void rsa_read_priv(int n, int d, file pvfile):
   Read values n, d from pvfile

void rsa_encrypt(int c, int m, int e, int n):
   //Compute cyphertext c
   // E(m) = c = m^e (mod n)
   pow_mod(c, m, e, n)

void rsa_encrypt_file(file infile, file outfile, int n, int e):
   // Compute block size k
   k = (log2(n) - 1) / 8
   Allocate memory for block
   block[0] = 0xFF
   While more unprocessed bytes to read:
      Read from file and write into block starting at index 1
      Convert read text to suitable integer format
      rsa_encrypt(c, m, e, n)
      print cyphertext c to outfile

void rsa_decrypt(int m, int c, int d, int n):
   // Compute message m from cyphertext c
   pow_mod(m, c, d, n)

void rsa_decrypt_file(file infile, file outfile, int n, int d):
   // Compute block size k
   k = (log2(n) - 1) / 8
```

```
    Allocate memory for block
    While more unprocessed bytes to read:
        read cyphertext c from infile
        rsa_decrypt(m, c, d, n)
        convert message m to text format
        write decrypted message m to outfile

void rsa_sign(int s, int m, int d, int n):
    pow_mod(s, m, d, n)

bool rsa_verify(int m, int s, int e, int n):
    int t
    pow_mod(t, s, e, n);
    if t == m:
        return true
    return false
```

---

keygen.c

```
// RSA Key generator program
int main():
    Parse command-line arguments

    Open Private and Public key files
    pbfile = public key file
    pvfile = private key file

    Set private key file permissions (read/write for the user only)

    Create a public key
    rsa_make_pub(p, q, n, e, nbits, iters)

    Create a private key
    rsa_make_priv(d, e, p, q)

    username = name of current USER running the program

    Create a signature
    rsa_sign(s, u, d, n)
```

```
  Write public and private keys
  rsa_write_pub(n, e, s, username, pbfile)
  rsa_write_priv(n, d, pvfile)
```

<u>**encrypt.c**</u>

```
// RSA Encryption program
int main():
  Parse command-line arguments

  Open public key file
  pbfile = public key file

  Read RSA public key
  rsa_read_pub(n, e, s, username, pbfile)

  Verify signature
  if rsa_verify(u, s, e, n) == false
    Error: signature could not be verified

  Encrypt file
  rsa_encrypt_file(infile, outfile, n, e)
```

<u>**decrypt.c**</u>

```
// RSA Decryption program
int main():
  Parse command-line arguments

  Open private key file
  pvfile = private key file

  Read private key
  rsa_read_priv(n, d, pvfile)

  Decrypt file
  rsa_decrypt_file(infile, outfile, n, d)
```

DESIGN PROCESS

Working with large numbers in C:

The standard C library integer types will not be sufficient for the purposes of the RSA algorithm. We want to make use of numbers with sizes over 1000 bits in length, which C doesn’t natively support. We will instead use the GMP library for generating and operating on numbers of arbitrary lengths. Additionally, we will use the random number generation function from the same library to produce large numbers of the same data type native to GMP.

Primality approximation:

The RSA encryption algorithm is dependent on the use of very large numbers the product of which is extremely difficult to factor given currently known algorithms. Developing such an approach requires the use of prime numbers; we want to be able to determine whether a given number is prime.
This task becomes extremely difficult for very large numbers, so we want to optimize our primality testing function to ensure efficient RSA performance.
We will use a randomized algorithm that implements probabilistic tests for primality approximations; we will implement the Miller-Rabin test

GMP Library

The GMP library has served as a great solution to handling very large numbers. While very useful, extraneous libraries have their learning curves and unspoken intricacies that need to be taken into account.
mpz_t is the main abstract data type that was used to represent integer values. Although passing mpz_t values to functions acts as a call by value (programming in c), it is important to note that calling setter functions on those values will update that specific entry’s value outside the scope of a function.
Not being aware of particularly attentive to this fact resulted in several bugs in my program. For instance, most numtheory functions would update the actual passed values whereas their original values needed to be preserved outside the function call; a solution to this problem was to create alias temp variables that would serve the purpose of mirroring their parent values.