Liam Murray
CSE13s Fall 2022
November 6, 2022
Assignment 5

Public Key Cryptography

General Idea:

The general idea of this assignment is to implement a basic cryptography library using the RSA algorithm, that can encode and decode messages based on numerical values.

The RSA algorithm will choose two random, arbitrary prime numbers p and q,, and multiply them together to get n. Then, an encryption key E will be chosen, such that E is co prime (does not evenly divide) n. Then, our decryption key D is calculated, based on the equation (d * e) is identical to 1 (mod totient(n)). Totient is the number of integers between 1 and N that are coprime to n.

Pseudo Code:

Greatest Common Denominator

The point of the following function is to compute the greatest common denominator or divisor of the two numbers inputted. This is used in several computations throughout the RSA algorithm.

```
GCD(storage_var, a, b) -computes the GCD of a and b, storing it in the storage variable
While (b != 0)
    T = b
    B = a % b //(a mod b)
    A = t
Set storage_var = a
```

Modular Exponentiation

Modular Exponentiation is a form of calculating exponents that is more efficient than simply repeated multiplication. This works by repeatedly squaring a number, starting at n^1, and multiplying each result by (mod n) to prevent absurdly large numbers.

```
Modular_exponents(a,d,n)
    V = 1
```

```
P = a
Counter  = d
While counter> 0
    If counter is odd;
        V = (v*p) % n
    P = (p*p) % n
    D = d /2
Return v
```

Miller-Rabin PseudoPrime Test

This function is designed to test, with a great deal of accuracy, whether a number is prime or not. It uses the Miller-Rabin pseudoprime algorithm, which works by testing whether or not a number is composite, and repeating this process enough times to say that it is in all likelihood prime.

```
is_prime(n,k);
// this first sequence is calculating r and s, such that 2^s * r = n - 1
F = 4
While (((n-1) % f) % 2 = 0) // while r is not odd
    S = f
    r= ((n-1) % f)
    F = f * 2  //increase S
For (i =1, until i = k)
    Rand = random(from 2 to n -2)
    Y = power_mod(a, r, n)
    If y != 1 and y != n - 1
        J = 1
        While j <= s-1 and y != n-1
            Y = power_mod(y, 2, n)
            If y = n
                Return false
            J = j +1
```

```
		If y != n-1
			Return false
	Return true
```

Modular Inverses

Computes a value i such that ai = 1 % n.

```
mod_inverse(i, a,n)
	R = n
	rp = a
	T = 0
	tp = 1
	While rp != 0
		Q = r / rp
		Temp = rp
		 Rp = r - (q * rp)
		R = temp
		Temp2 = tp
		 tp = t - (q * tp)
		T = temp2
	If r > 1
		Deref i = 0 //no inverse, i = 0
	If t < 0
		T += n
	Set i = t
```

Make Prime

Generates a random prime number, with a specified bit size, putting it into P address. This takes advantage of the fact that prime numbers can be expressed as 6(n) +1 or 6(n) -1.

```
Make_prime(pointer P, bits, iters)

	R_int = modular_exponents(2,  bits) // raise 2 to the power of bits
```

R_int -= 1 //compute maximum number held in that many bits, thus giving us our bottom range

Rand = random number from r_int to (r_int * 100)

Prime = (rand * 6) - 1

If is_prime(prime, iters) == True

Pointer P = Prime

Else if is_prime((prime+2), iters) == True

Pointer P = (prime + 2)

RSA Make Pub

Makes an RSA public key

rsa_make_pub(p , q, n, e, bits, iters)

Iters = random() //random number

P_Bits = random() in range (bits/4, 3*bits / 4)

Q_bits = (3 * bits / 4) - P_bits

make_prime(p, P_bits)

make_prime(q, Q_bits)

Lam_n = ((P-1) * (Q-1) / gcd(P-1, Q-1)

For(int e, while gcd(e, lam_n) != 1)

E = mpz_randomb(bits)

Return E

Rsa Make Priv

Makes a new private key based off of public key

rsa_make_priv(d,e,p,q)

Lam_n = ((P-1) * (Q-1) / gcd(P-1, Q-1))

Declare d = 0

modular_inverse(d,e, lam_n)

Return d

Writes a public key to a file

rsa_write_pub(n , e, s, username, filename)

FILE = fopen(filename, w)

gmp_fprintf(FILE, %X, n)

fprintf(FILE,%c, \n)

gmp_fprintf(FILE, %X, e)

fprintf(FILE,%c, \n)

gmp_fprintf(FILE, %X, s)

fprintf(FILE, %c, \n)

fprintf(FILE, username)

fprintf(FILE, %c, \n)

fclose(FILE)

Reads public key from a file

rsa_read_pub(n , e, s, username, filename)

FILE = fopen(filename, r)

n = gmp_fscanf(FILE, %X, n)

e = gmp_fscanf(FILE, %X, e)

s = gmp_fscanf(FILE, %X, s)

username = fscanf(FILE, %s, username)

fprintf(FILE, \n)

fclose(FILE)

Writes a private key to a file

rsa_write_priv(n , d, filename)

FILE = fopen(filename, w)

gmp_fprintf(FILE, %X, n)

fprintf(FILE,%c, \n)

gmp_fprintf(FILE, %X, d)

fprintf(FILE,%c, \n)

Reads private key from a file

rsa_read_priv(n , d, filename)

FILE = fopen(filename, r)

n = gmp_fscanf(FILE, %X, n)

e = gmp_fscanf(FILE, %X, d)

username = fscanf(FILE, %s, username)

fprintf(FILE, \n)

fclose(FILE)

Encrypts a block of data using the c, m, e, n values passed in.

rsa_encrypt(c, m, e, n)

C = modular_exponent(m to the e)

C = c % n

Return c

Encrypts a whole file using the key information provided.

```
Rsa_encrypt_file(FILE *infile, FILE *outfile, n, e)

        //k is the block size

        K = log(n) / log(2)

        K = (k - 1) / 8

        Arr = Calloc(uint8_t array of size k)

        Arr[0] = 0xFF

        FILE = fopen(infile, r)

        OUTFILE = fopen(outfile, r)

        for(j = 1 ; j  <= k -1; )

                fscanf(FILE,  %c, char)

                M = mpz_import(char, 1,1,0)

                M = rsa_encrypt(m)

                GMP_fprintf(OUTFILE, m)

        Close both files
```