FCC Assignment 2 – RSA Encryption
Jhi Morris (19173632) – 17/05/19

Part 1 – gcd

gcd(*a*, *b*) is implemented in keygen.c as gcdEx(*a*, *b*, *&x*, *&y*) and returns the greatest common denominator of *a* and *b*; as well as the coefficients for *a* and *b* as *x* and *y*.

gcd(12543, 1682) = 1. Thus, 12543 and 1682 are co-prime.

gcd(*a*, *b*) = gcd(*b*, *a* mod *b*) in case of a > b is true as if *a* is greater than *b*, the remainder will be a reduced by *n***b* for *n***a*/*b* <= 1. Thus the greatest common denominator will be the same as they share the same common denominators of *b* and its factors OR *a* mod *b* == *a* and there is no difference.

Part 2 - RSA

The public and private key pair is generated by the keygen program, either from selected or random primes and public key exponent (*e*) value. The number of tests done to ensure generated primes are in-fact prime can be configured and is defined in keygen.h also. The maximum size for generated primes and *e* values are configurable; defined in keygen.h and set at compile-time. These limits do not apply to manually selected values, though a warning may be printed for larger values as they may exceed datatype limits during calculation; which will cause an integer overflow and create invalid data.

Keys are generated by getting the prime values (*p*, *q*), the exponent value (*e*), ensuring *p* and *q* are indeed both prime, then generating *phi* of *p***q* using a short-cut version of Euler's Totient function that operates on the assumption that *p* and *q* are co-prime (which they should be if they are both prime). After this, *e* is checked to be above 1 and below *phi* using the Extended Euclidean Algorithm with the coefficient of *e* saved as the private key exponent (*k*) value. If negative, *k* may need to be corrected by adding the value of *phi* to it until it is positive. Finally, the public key is printed to the user in the form *n*#*e*, and the private key is printed in the form *n*#*k*. The symbol '#' is literal in this case; not a placeholder.

The public and matching private keys can then be used to encrypt and decrypt files respectively using the rsa program. The block size (number of characters encrypted at a time) is defined in rsa.h and can be configured then set at compile-time. The rsa program loads data from a file and then writes the output to a file. It writes the encrypted data as comma-separated long integers, each representing a block. If the block-size when decrypting is set differently to when the data was encrypted, the output will be invalid. A valid matching public/private key pair must be used too, or data will not be able to be decrypted properly. A larger block size requires larger key values to work correctly.

The rsa program takes the filename of the input and output files, as well as a flag defining if it is to encrypt or decrypt the input into the output and a public or private key respectively. The keys are input in the form *n*#*e*. The symbol '#' is literal in this case; not a placeholder. It is used as a simple delimiter of the modulo and exponent values of each key.

When encrypting, the rsa program loads each character from the input file and copies it as a unsigned integer to a buffer variable. Multiple characters to a total of the configured block size will be added to this variable in sequence, offset based on their order in the input file. This buffer variable is then treated as a single integer and encrypted by finding the value of ($buffer^e$ mod n) and writing it to the output file. This is done using the Fast Modular Exponentiation Algorithm detailed page 269 of *Cryptography and Network Security – Principals and Practise* by William Stallings. This process is repeated until

every character in the input file has been encrypted in this way, with each block written separated by a comma. If it reaches the end of the file whilst in the middle of filling a block it will write it as-is.

When decrypting, the same method is used although now from an integer back to characters, and with using the private key as the exponent – not the public key as before. The integers are loaded from the file one at a time, decrypted with the Fast Modular Exponentiation Algorithm, then unrolled back into a number of characters equal to the block size. If this block had less than the block size written to it, that part of the block will be all zeroes and represent a null character if printed. These null characters are not printed.

Part 3 – Usage & Example

The keygen and rsa programs can be compiled using the provided makefile by running *make*. Keygen can be run either with the *-r* flag (examples below), in which case it will generate a public and private key from primes randomly generated using the current system time. If you run the program multiple times in under a second, it may return the same results. It will also return the values of *p*, *q*, and *e* used to generate the key. The keygen program can also be run using manually specified primes and *e* value by running it with arguments *p*, *q*, *e*. Checks are still run to ensure that *p* and *q* are prime, and that *e* is a valid value.

Once a valid public and private key pair have been generated, the rsa program can be run. You must specify -e (for encrypt) or -d (for decrypt), the input file, output file, and relevant key (public for encryption, private for decryption). The program will then run and encrypt or decrypt the input file into the output file.

The behaviour of keygen and rsa may change based on certain settings only configurable during compile; for example the block size or the number of tests a prime is tested. If you wish you change these settings, you must recompile the program with *make* again.

The below example was run with *BLOCK_SIZE* set to 2.

```
$ make
  gcc -Wall -lm -0fast rsa.c -o rsa
  gcc -Wall -lm -0fast keygen.c -o keygen
$ ./keygen -r
  p:1409, q:2311, e:358619
  Public key: 3256199#358619
  Private key: 3256199#2893139
$ ./keygen 1409 2311 358619
  Public key: 3256199#358619
  Private key: 3256199#2893139
$ cat testFile
  abcdef1234
$ ./rsa -e testFile testFileEnc 3256199#358619
$ cat testFileEnc
  20431,2121348,1501681,2551783,2237350,1591050,
$./rsa -d testFileEnc testFileOut
$ cat testFileOut
  abcdef1234
```

Attached are copies of a larger test file, after encryption and then decryption, as testFile, testFileEnc, and testFileDec. These were encrypted and decrypted with the same key pair as the example above.

Part 4 – Further

> Assuming that Alice signed a document m using RSA signature scheme. (You should describe RSA signature structure first with a diagram and explain the authentication principle). The signature is sent to Bob. Accidentally Bob found one message **m'** ($m \neq m'$) such that **H(m)=H(m')**, where **H()** is the hash function used in the signature scheme. Describe clearly how Bob can forge a signature of Alice's with such **m'**. **Justify your forgery with the knowledge you learned from this unit.**

The process of creating a signature for a message is simple but relies on the fact that not only does decryption reverse the process of encryption, but that encryption reverses the process of decryption. Because of this, someone with a private key can encrypt a message that any person with the public key can decrypt. This can be used to digitally sign messages by first hashing the message, then "decrypting" the hash using the private key. This will create a unique value which (in theory) can only be generated by the unique combination of message hash and private key. A problem arises when a different message produces the same hash, as in this scenario. If this hashed message is known to be identical to the hash generated by the person signing their message, then the signature for that message could be used to sign the different message. This problem could be somewhat avoided by using a hashing algorithm that reduces collisions, and further the pre-decryption message should not be public; so that even in the case of a collision it is impossible to ascertain that the collision message hash is equal to the genuine message hash.

> In a group of 23 randomly selected people, the probability that two of them share the same birthday is larger than 50%.

With a group of 23 people, there are (22*23/2) = 253 possible combinations. The chances that one person does not share a birthday with another person (assuming no leap years) is 364/365. Therefore the odds that any two people in a group of 23 *do not* share a birthday is $(364/365)^{253} = 0.4995228$. The chance that any pair share a birthday (and thus some two people in the group share a birthday) is $1 – 0.4495228 = 0.5004772 \approx 50\%$. Slightly higher than fifty percent odds.