**Comp-6370 Project Summary**

**William Christopher White**

**Github link:** <https://github.com/bootlessxfly/RSA-UDP-Client-Server-Project>

**Design:**

I decided to write this project in C++. I felt that this would give me efficient code without some of the complexity that C would afford. Also, C++ seemed like the more common choice for networking when scanning through the languages used in the Linux Kernel. I broke my code into 6 classes that I will outline below:

* Key.cpp:
  + This is the main class for all of the key information for the public and private keys. This is where most of the magic happens. The class contains many methods for building out different aspects of the private and public key. At its heart is a random number generator which randomly calculates two prime numbers p and q. The two are then used to calculate n and phi (z). If n is generated and not higher than 50, then the random number generator would run again. From z and n, the public exponent (e) is calculated. Using z and e, the private exponent (d) is calculated. Once the key was generated, it is ran through a validation process to verify that the key can properly encrypt and decrypt the entire ASCII decimal range with no errors. If there is an error in the validation, the entire process is reran until a key is generated that can pass this process. Important method descriptions are below.
    - generatePQ()
      * For each p and q, a random number is generated from 2 to 1024. This number is checked to see if it is prime, and is iterated by two until it becomes prime.
    - findPublicExponent(int z, int n)
      * The public exponent was generated by finding a value e that started off as being one of the first five ferment numbers (3, 5, 17, 257, 655537). A random number is generated between 1 in 2, allowing for one of the ferment values to be randomly selected. The selected ferment number is then checked to see if it is relatively prime with z. If e is not, then e is iterated and rechecked to see if it is relatively prime with z. Additionally, there is a one and 6 chance that e will start start off at fermant number + 1. This added randomness is an attempt to the strength of the key.
    - findPrivateExponent(int e, int z)
      * Once a value for e is found, d is calculated. It is found using the formula d = (1 + (k \* z)) / e where k is a random value between 1 and 7. Once again, I attempted to randomize the calculation of d to add to the strength of the key.
    - buildKey()
      * Simply facilitates the building process. Once the key has been validated, the key is set and the system is informed that the key was generated.
    - validateKey()
      * This is an important part to the key generation process. My RSA key generation algorithm did not always return a key that could encrypt and decrypt the entire range of the ASCII chart. If the entire range can not be encrypted/decrypted, then a regeneration of they keys is performed until a valid key is found. Once this step passes, the private/public key generated are guaranteed to work with each other on any ASCII step. While this is probably due to an error in my algorithm, this step adds very little noticeable overhead(See more in implementation issues).
  + Notes
    - All numbers involved in the process are pseudorandom(More on the issues with this later). Since e and d are random, using the same values for p and q will not always give you the same value for d and e. This, in theory, should make cracking the key even harder, specifically for d. If you are able to crack p and q, you may not immediately be able to generate the same d. I tested this by providing the same value for p and q multiple times. While e only changed occasionally, d would almost always change. And if e did change, the range in which d could change would increase even higher. This establishes confusion since knowing p and q will not necessarily allow you to read the cipher text.
* RSA.cpp
  + This class is a wrapper around the key class and acts as the director of the RSA process. It contains two key objects. One for the actors private/public key pair, and one for the other actors public key. Additionally it contains the encryption and decryption methods. One more important method is the testASCIIEncDec method. While the validation method above tested for encrypting and decrypting using a single key pair, this test encrypting and decrypting using both sets of keypairs. Important method descriptions are below.
    - encrypt(int m)
      * Takes an integer m that represents a single ASCII character. It first encrypts it using the actors private key, and then encrypts it using the other actors public key. It returns this encrypted number.
    - decrypt(int m)
      * Takes in an integer that represents a single encrypted ASCII character. It first decrypts it using the actors private key. Then decrypts it using the other actors public key. It returns an integer that represents an ASCII character.
    - testASCIIEncDec(RSA rsa, RSA rsa1)
      * This method takes in two sets of keys and validates that they can encrypt and decrypt using both sets of keys. While the key classes validation guarantees that a single key will be able to do encryption, it does not guarantee that both sets of keys will be able to encrypt and decrypt each other. This method was mostly used for testing, as the server and client have there own way of validating keys with each other. This step does add overhead and is the most expensive part of the entire program.
* Utility.cpp
  + This class adds facility between the actors and the RSA process. This class offers the interface that the client and server use to generate and validate RSA keys. Additionally, it contains the logic to write and read the contracts. Important methods are below
    - encryptMess(char\* mess);
      * This takes in the contract message. It then loops through each individual ASCII character and converts it into its integer form. The integer is encrypted. Each individual integer is delimited with “:”. Returns the cipher text
    - decryptMess(char\* cipher)
      * This takes in the cipher text. It delimits the text by “:”, and converts each delimited char into an int. The int and sent through to the RSA decryption process and received back its decrypted ASCII character.
    - checkSameKey()
      * This is another validation process. It checks to see if the actors public key and other actors public key are the same, and if they are returns false. This is used to allow the client and server to know when they are using the same key, so they can regenerate their keys.
* Client.cpp
  + This class represents the client side actor. The client picks out a file to send, and then provides the server information. It then attempts to send the server its public key. The server will respond with its public key. The client then sends the server a validation the string, and waits for a response. The server sends its validation string, and the client confirms it. If one can not validate the string, the keys are regenerated. This process is repeated until a valid key pair is confirmed. Once the key exchange is completed, the contract is encrypted and sent to the server. It waits to receive the encrypted signed contract. Once received, the client decrypts the contract, validating that the server has sent the signed contract.
    - validateKeys()
      * This method is performed after each key exchange. It generates a string that represents each ASCII character, and sends it to the server. If the server responds with a NACK, then both sides begin to regenerate their keys, and the client sends a new validation string. If the server responds with an ACK, then it will send back its validation string that the client then validates. This process can add significant overhead. Also, if the last validation cipher is intercepted, then the attacker essentially has a key he can use to decipher a ciphertext. This method does allow me to generate the perfect set of key pairs for encryption and decryption.
* Server.cpp
  + This class represents the server side actor. The server performs a very similar set of tasks as the client, so I will not go into much detail here.
* RSA\_UDP\_Client\_Server\_Project
  + This is the driver class. It contains an interface that allows you to launch either the client or the server side application, Additionally, it launches an application that will generate keys that have gone through and passed all validation processes. The key generator is used for statistical analysis and testing of the key generation.

**Implementation Issues and Results**

There are a number of issues with my implementation. The biggest issue is in key generation. While the key generated is guaranteed to work with the given private key/public key combination, it is not guaranteed to work in combination with another key pair. This meant that I had to add a large amount of overhead in validating and regenerate the keys. If my configured sizes for p and q where small (under 256), then this validation process would take less than 5 minutes. If I had larger p and q values, this process could take long. With p and q having a max size of 4096, this generation could take upwards of an hour. Sizes larger than that would become very impractical, I'm not fully sure if this problem occurs because my algorithm produces many bad key pair sets, or because generating a good key pair set that can do both integrity and confidentiality is hard. Regardless of the reason, this is the programs biggest flaw.

Additionally, if someone where to intercept my validation cipher, they would possibly be able to read any additional cipher sent during that session. All one would have to do is wait for the client to send an ACK, and the last used validation cipher could act as a key for the contract being sent out. This is because the validation cipher contains each character is the ASCII table, meaning you have the decrypted version of each ASCII character in order. This would completely defeat the purpose of using one actors private key and other actors public key to decrypt the message. Additionally, since the attacker would be able to guess what each decrypted number is in ASCII, he could run a statistical analysis and calculate the private exponent d. At this point he would have each actors private key, and would be able to not only read intercepted messages, but also create messages of his own. In this programs defense, the attacker would have to keep up with 100s of crackable ciphers, and determine which validation cipher was successful. If he were to miss the last sent validation, he would have no way easy way of decrypting the following messages. While this does allow me to generate a strong set of key pairs that can be used for integrity and confidentiality, the validation method should be seen as a major weakness.

One solution to this problem would be to use an interchange key. Using Cathy as a trusted third party, each actor would send cathy their public keys. When the client and server want to start sending contracts, they would get the others public key from Cathy. The public keys received from cathy would be used for an interchange key. At this point, each validation cipher would be encrypted with the other’s public key. Once a public/private key pair is validated, that generated public/private key would be the session key. Since the session key is encrypted using an interchange key, if the ACKed validation cipher was intercepted, the attacker would not know it unless he knew the session key. I wanted to simulate this process in my program to add on security and fix the issue that occurs from needing to send validation ciphers to validate the generated keys, but time did not permit this. Even if this program did not have the issue of being required to validate each actors key pair, this would be a smart move, because the public key exchange was vulnerable to a man in the middle attack. Adding the Needham-Schroeder Protocol with Sacco modification or the Otway-Rees protocol would limit Cathy from being affected by a man in the middle attack.

Ironically enough, using only a single key pair in this case would be more secure as the single key pair has already been validated to work and a crackable validation cipher would not need to be sent. Remember from above that once a single key pair is generated, it is validated to work with anything encrypted/decrypted with that pair, just not when combined with together with someone else’s key pair. In such a procedure, the client actor could use his private key to send a signed hash of the message. The server would validate the hashed message and sign it with his own private key. The client would receive the signed hash, and validate that he is talking to the server. He would use the server’s public key to encrypt the actual message and send it to the server. The server would decrypt the message with his private key pair and generate a hash. The hashes would be compared, and if they match, the server has validated the message and the sender. This would also meet both integrity and confidentiality. As with the above methods, this would also be subject to a man in the middle attack. Like the above method, using a trusted third party would limit this

The last problem with this implementation is with the minimum size of n. The minimum generated size is 50. 50 is a very small number to have for n. It is feasible to be able to calculate p and q using brute force math, at which point you could try to calculate d and e. While I did try to add a certain amount of randomness to the generation of e and d while also giving p and q a large random range, this would still be a problem. Increasing n to a size greater than 1000 would greatly reduced this. And increasing it even further would help even more. Additionally, the random number generators in computers are by nature pseudorandom and not truly random. This would open the public key open to statistical attacks that would allow you to possible calculate p and q, and then generate d.