CSE 30331 Final Project: *crypChat* Encrypted Chat Program
Christopher Durr, Matthew Mooney, Paul Zakas

**Abstract:** Secure communication over the internet is critical in today’s world. Our project is an encrypted chat program designed to ensure a safe mode of communication between two users. It makes use of RSA encryption to encode and decode messages in a terminal window. Clients exchange public keys and encrypt outgoing text with the intended recipient’s public key. Each user connects to a central server which forwards messages to the intended recipient. On the receiving end, the program utilizes a client’s private key in order to decrypt received messages.

1. **Keywords**

- RSA encryption: A public key cryptography algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT. The security of RSA is based upon the problem of factoring large numbers, for which no efficient algorithm exists.
- Public key cryptography: A form of cryptography in which the key used to encrypt a message is different from the key used to decrypt it. In public key encryption, a user has a pair of cryptographic keys – one public and one private key. The private key is kept secret, while the public key may be widely distributed.
- Public key: Key used to encrypt messages; can be shared with others.
- Private key: Key known only to the private user; used to decrypt messages.
- Co-prime: Two numbers are said to be co-prime if their only common factor is 1, namely that their greatest common divisor is 1 (e.g. 10 and 21).
- Relatively prime: Two numbers that are co-prime.
- Encrypt: To turn data into cipher text.
- Decrypt: To turn cipher text back into its original, meaningful form.
- Modulus (%): Mathematical operation which gives the remainder of division (e.g. 11 % 7 = 4).
- Totient: For distinct prime numbers p and q used in the RSA encryption algorithm, the totient is equivalent to \((p-1)*(q-1)\). The public key exponent is derived from the totient, insofar as the public key exponent is co-prime with the totient.
- Cipher text: Unreadable text which is representative of the original data. In RSA, the equation \(c = m^e \mod n\) is used to generate cipher text.
- GMP (GNU Multiple Precision Arithmetic Library): GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. It is utilized in our project to store and manipulate large numbers which require more bits and precision than the standard “long long” or “double” data types.
- Miller-Rabin Primality Test: Given an odd integer \(n\), let \(n = 2^r s + 1\) where \(s\) is an odd number. Choose a random integer \(a\) with \(1 <= a <= n-1\). If \(a^s \equiv (\text{mod } n)\) or \(a^{2js} \equiv -1 (\text{mod } n)\) for some \(0 <= j <= r-1\), then \(n\) passes the test. A prime number will pass the test for all \(a\) values.
2. Introduction

Our group developed an encrypted chat program. To encrypt the messages, we used the RSA encryption algorithm. Although the RSA algorithm is publicly available, it was non-trivial to implement and required the use of the GNU Multiple Precision Arithmetic Library (GMP) for the large numbers used to encrypt and decrypt messages. The program computes the necessary values for encryption and decryption of the chat and feeds text messages into the RSA class, which turns the text into cipher text. The messages are sent to the server program which directs the message to the appropriate client. Upon receiving the message, the client program decrypts the message back into plain text for reading.

3. Main Program

3.1 Background

Our goal in this project was to effectively implement the different methods we learned in CSE 30331: Data Structures. In the class, we focused on Object Oriented Programming (OOP) and fundamental techniques in the design and analysis of non-numerical algorithms and their data structures. We utilized these techniques to implement an encrypted chat using RSA encryption.

3.2 Result

To implement our RSA encryption chat, we used a custom built class to represent the RSA encryption algorithm. This class computes all the necessary values used in the RSA encryption algorithm, such as public key exponent, private key exponent, etc. In addition, the algorithm is fed the string, in sub-strings, of the text the user is attempting to send to chat with his/her friend. After transmitting the message, the other client decrypts the message using its own private key.
3.3 Functionality

Our program allows users to chat with each other in a secure manner. The messages sent between the users are encrypted and can only be read by the other person in the chat, who has the private key. Public keys are exchanged before communication begins, so all text sent by the users in encrypted.

3.3.1 Computing the Public Key Exponent

Two primes, p and q, are generated by the PrimeGenerator class upon running the program. This class generates a large number as a prime “candidate” and then passes this number through a series of prime probability test functions. These functions utilize a Miller-Rabin Primality Test (see Keywords section). Once these two large primes are generated, they are multiplied together to compute public key modulus n. A user’s public key is comprised of n and e.

3.3.2 Computing the Private Key Exponent

Using the primes p and q, the totient (see Keywords section) is computed as well. The private key exponent d is computed by solving the relation d*e \equiv 1 \pmod \text{totient}. This equation is solved in the RSA class which incorporates the extended Euclidean algorithm (see Keywords section). Once this step is complete, both the public key (n and e) and the private key (n and d) have been generated.

3.3.3 Establishing Network Connections

The client program connects to a user-specified server and port number. Each client connects to the server in order to communicate with a specific peer on the network. In order to communicate over the network, a TCP socket is created and managed by the clientConn class. The clientConn class handles all network traffic with a simple API so that all network related code (mostly written in C with some C++ STL string augmentation) is performed “under-the-hood” of the main client program. The main function also threads a function which submits periodic “hello” messages to the server using the clientConn API to keep the connection alive.

3.3.4 Exchanging Keys

Once each client has generated a key and joined the server, the server instructs the clients to exchange public keys. Each client stores the other’s public key and uses it to encrypt outbound messages.

3.3.5 Computing the Message into Numeric Value

Before generating the cipher text to send to the other user, the message must be converted to integer form so that it can be handled by the RSA encryption function c = m^e \pmod n. The “encrypt” function takes an STL string as input and first converts each character to its three digit ASCII representation. Each ASCII code is then concatenated and stored as the variable m.
3.3.6 Computing the Cipher Text (Encoding the Message)

With the new m value, the “encrypt” function generates cipher text by raising m to the power of e, and then taking the resulting value by the modulus of n. The cipher text is stored in the variable c.

3.3.7 Sending Messages

The cipher text is converted back to a string and inserted behind a custom-built message header which includes a command to be processed by the server, the sender’s username and the intended recipient’s username. This string is submitted to the socket using the clientConn API. The server can then appropriately route the message to the correct client which receives the message on a threaded function that utilizes the clientConn API.

3.3.5 Decoding the Cipher Text (Decoding the Message)

Using the cipher text c, the encoded message can be recovered by first raising the cipher text to the power d, and then taking the result by a modulus of n. The resulting number m is then broken into the three digit ASCII codes which represent each character of the original message. In our implementation the encoded message m is decoded into text within the same “decrypt” function. The decrypted plain text message is displayed to the user in clear, readable format. This decryption and display is performed in a separate thread from main so that we could use a “blocking” read call, one which does not finish until something is detected coming into the socket.

4. Conclusions

Overall the project was a success. Looking back, our group should have analyzed the GMP Application Programming Interface (API) more closely. We wrote and debugged whole functions which were already contained in the GMP API. In addition, we probably should have planned out the whole process of RSA encryption before coding. At first, the Sieve of Eratosthenes algorithm was used for finding prime numbers. The Sieve is inefficient to compute the large prime numbers necessary for the RSA encryption. In addition, we naively assumed at first, that the padding scheme for the message would result in characters of less than 15 digits, thus enabling us to use long long ints and other data types. Our padding scheme works off of the ASCII values of each character. Thus, the number of digits in the padded integer M for the message m, is \(10^{3\# \text{of characters}}\). We knew this would cause issues transferring over the network, and initially we were puzzled on how this was exactly going to take place. However, we soon saw the API for the Multiple Precision Integers had a function that would convert the Multiple Precision Integer into a string. However, we soon realized that the padding scheme resulted in very large numbers, and the RSA encrypted called for those large numbers to be raised to large powers. These operations are not available on fundamental C/C++ data types, and so that is why we needed to employ the GMP Library. As a result, many of our previously written functions had to be modified to work with the GMP API. We carefully planned out the process to make sure that the two entities could logistically interface with each other.
5. Future Work

With what has been created thus far, the main body of the program is in place for others to expand upon our project as they see fit. Some ideas we considered adding at a later date are improving upon the aesthetics and the security of the program. To improve the aesthetics of the terminal display, we could improve the way messages between the users are displayed. This would be possibly with a GUI written in Qt or similar software. To improve security of the program, we could standardize a time to decrypt a message, regardless of size. If the attacker knows a chat user’s hardware in sufficient detail and is able to measure the decryption times for several known ciphertexts, the attacker can deduce the user’s private key, \( d \), relatively quickly. However, standardizing a decryption time leads to an obvious slowdown in performance.

Another possible improvement would be to add a signing feature to our chat program. If user A uses user Y’s public key to send user Y an encrypted message, user A can claim to be user X but user Y has no way of verifying the message was actually from user X since anyone can use user Y’s public key to send encrypted messages to user Y. To prevent this, a signing method could be used to verify messages. The padding scheme could also be changed. Our current padding scheme of ASCII value representation of the string does not provide maximal security. A possible substitute would be an Optimal Asymmetric Encryption Padding (OAEP), which is often used together with RSA encryption. This padding scheme satisfies two important properties when used in tandem with RSA. Namely, it adds an element of randomness which can be used to convert a deterministic encryption scheme of RSA into a probabilistic scheme, and it prevents partial decryption of ciphertexts by ensuring that an attacker cannot recover any portion of the plaintext.

6. References

- Some functions sampled from [http://code.google.com/p/rsa/](http://code.google.com/p/rsa/) under the GNU General Public License
- [http://www.geometer.org/mathcircles/RSA.pdf](http://www.geometer.org/mathcircles/RSA.pdf)
7. Biographies

Christopher Durr

Chris is a junior Computer Science major at the University of Notre Dame du Lac. Having never programmed nor ever seen a line of source code before college, Chris can usually be found getting extremely frustrated with his homework. While not roaming the hallways of Fitzpatrick and Cushing, Chris is usually running, playing guitar, or eating Subway™.

Matthew Mooney

Matt is a junior Computer Science major at the University of Notre Dame du Lac. He is focusing his education on computer networks, but when asked what he plans on doing with his degree after graduation, he gives the completely unrelated response, “Fly jets.” When his life is not consumed by schoolwork or Air Force ROTC, he pretends to like running and Subway but really enjoys a good movie or round of Halo.

Paul Zakas

Paul is a junior Computer Science major at the University of Notre Dame du Lac. He enjoys playing ping-pong and eating cookies at North Dining Hall. He is not funny and has no personality whatsoever.
Appendix A: Selected Source Code

The following snippet of code creates a sequential Multiple Precision Integer based upon the consecutive ASCII values of a string. For example, string “abc”, ASCII codes of 97, 98, 99, respectively, gets translated into the Multiple Precision Integer 979899. The value is then raised to the power e, and modded by n, according to RSA conventions. The value then is translated into a char* using a function from the GMP API, which converts the Multiple Precision Integer into a 16 bit char* representation:

// various cases based upon the size of the message
// createM parses 3 characters at a time, so need
// different cases based upon % 3
if(message.size() % 3 == 0) {
    power = message.size() / 3;
    // each character takes up 3 decimal digits
    --power; // subtract so the last power is 0
    for(i=0; i<message.size();)
    {
        string sbstr = message.substr(i, 3);
        // create 3 letter substring
        sbstrVal = createM(sbstr);
        // get the ASCII value of the substring
        mpz_pow_ui(temp, ten, power * 9);
        // each 3 letter substring ASCII takes up 9
digits put the values into the right nth digit positions by multiplying by power
        mpz_mul_ui(temp, temp, sbstrVal);
        // now multiply the nth digits by the ASCII value
        mpz_add(messageVal, messageVal, temp);
        // add the temp to the running total
        i = i + 3; // go to the next 3 letters
        power--;
        // subtract 1 from power, put the next 3
        letters, in the 9 lower decimal places from
        the previous position
    }
}

// formatting if you have one extra character than % 3
else if(message.size() % 3 == 1) {


string sbstr;
power = message.size() / 3;
--power;
for(i=0;i<message.size();)
{
    if(i+3 < message.size())
        // if you havent reached last character
        sbstr = message.substr(i,3);
    else
    {
        // if you have reached the last character get the ASCII at the last
        // character
        unsigned long int k = (unsigned long int) message[message.size()-1];
        mpz_add_ui(messageVal, messageVal, k); // add it to the running value
        break; // you are done
    }
    sbstrVal = createM(sbstr);
    mpz_pow_ui(temp,ten,power * 9 +3);
    // extra character, adds 3 digits
    mpz_mul_ui(temp,temp,sbstrVal);
    mpz_add(messageVal, messageVal, temp);
    i=i+3;
    power--;
}
else if(message.size() % 3 == 2)
{
    string sbstr;
    power = message.size() / 3;
    --power;
    for(i=0;i<message.size();)
    {
        if(i+3 < message.size())
            // havent reached last two characters
            sbstr = message.substr(i,3);
        else
        {
            // get the second to last character value multiply by 1000
            unsigned long int k = (unsigned long int) message[message.size()-2] * 1000;
            k += (unsigned long int) message[message.size()-1];
            // add the last character to the value
            mpz_add_ui(messageVal, messageVal, k);
            // add to the running total
        }
    }
}
break;  // you are done
}
sbstrVal = createM(sbstr);
mpz_pow_ui(temp, ten, power * 9 +6;
   // have extra 2 digits, adds 6 digits
mpz_mul_ui(temp, temp, sbstrVal);
mpz_add(messageVal, messageVal, temp);
i = i + 3;
power--;
}
mpz_powm(messageVal, messageVal, e, n);
// do mod operation required in RSA
size_t numdigits = mpz_sizeinbase(messageVal, 16);
// get number of digits
char *convert = new char[numdigits];
// create a char* based upon messageVal
mpz_get_str(convert, 16, messageVal);
// messageVal converted into base 16 value and stored in char* convert string
ctxt(convert);
// create a string based upon messageVal
return ctxt;

The following snippet is the main functionality of the createM function called by the encode function:

unsigned long int j = message.size()-1;
// CREATE AN INTEGER BASED UPON ASCII VALUES OF STRING
for(i=0;i<message.size();i++)
{
   mpz_pow_ui(temp, ten, (unsigned long int)j*3);
      // push first ASCII value to the higher digits of number
mpz_mul_ui(temp, temp, (unsigned long int)(message[i]));
      // get the message into the higher bits by multiplying ASCII value
mpz_add(messageVal, messageVal, temp);
      // increment the running sum
j--;
}
unsigned long int rtn = mpz_get_ui(messageVal);
return rtn;
The following snippet of code is used in decrypt. At this point, the ASCII representation of the
message the other user sent, is in the form of a Multiple Precision Integer. The following code
translates the Multiple Precision Integer into a char* equivalent:

```c
int numdigits = mpz_sizeinbase(c,10);
i=numdigits;
mpz_ui_pow_ui(gtemp,10,i);
mpz_mod(gresult,c,gtemp);
mpz_t result;
mpz_init(result);
mpz_ui_pow_ui(temp,10,i-1);
mpz_mod(result,c,temp);
mpz_sub(temp2,gresult,result);
unsigned long int val = mpz_get_ui(temp2);
//if you get zero, you subtracted the messageVal from itself, thus numdigits is off by 1
if(val == 0)
{
    numdigits -= 1;
    //if numdigits is 1 more than it should be
    i = numdigits;
    //subtract 1 from it, and start the loop at the
    //correct val
}
//numdigits is sometimes one more than the numdigits in MPZ_T. This is an error in the GMP
Library and is documented in their API. The above code checks for that particular error.
```

for(i;i>0;i--)
{
    //take the mod 10th power
    mpz_ui_pow_ui(gtemp,10,i);
    mpz_mod(gresult,c,gtemp);
    //take the mod one 10th power less
    mpz_ui_pow_ui(temp,10,i-1);
    mpz_mod(result,c,temp);
    mpz_sub(temp2,gresult,result);
    //subtract those values, giving you a lone digit
    followed by 0s
    mpz_cdiv_q(temp2,temp2,temp);
    //divide by the lesser 10th thus getting you the lone
digit by itself
    val = mpz_get_ui(temp2);
    //get the value of the digit
std::stringstream out;
out << val;
tempascii = out.str();
    // tempascii string gets that value using stringstream
totascii+=tempascii;
    // add that character to the running totascii
}

The following snippet of code, also used in decrypt, takes the char* ASCII representation of
the message, and translates the ASCII values into their character value and pushes them onto
a string, which contains the original message:

int getCeil = numdigits/3;
if(numdigits % 3 != 0) // need ceil(numdigits /3)
    getCeil++;
    // formatting is ASCII >99
if(totascii[0] == '1')
{
    for(i=0;i<getCeil;i++)
    {
        int l=i;
        cval = (totascii[l+k] - '0')* 100;

        cval += (totascii[k+(l+1)]-'0')* 10;

        cval += (totascii[k+(l+2)] -'0');
        fmess[i] = (char)cval;
        k+=2;
        // increase by 2, to get 3 chars (i increases by 1)
    }
}

// formatting if first character has ASCII <100
else
{
    for(i=0;i<getCeil;i++)
    {
        int l=i;
        if(i==0)
            cval = (totascii[l+k] - '0')* 10;
        else
            cval = (totascii[l+k] - '0') *100;

        if(i==0)
This snippet of code is from the clientConn class. This is the function responsible for attaching the custom header and submitting the messages (already encrypted) to the server:

```cpp
void clientConn::send_to_serv(string mess, string destname) {
    mess = "S " + username + " " + destname + " " + mess;
    char* buffer = new char [mess.size() + 1];
    strcpy(buffer, mess.c_str());
    int n;
    n = write(sock_id, buffer, strlen(buffer));
    if(n < 0)
        {
            try{throw sock_ex();
            }
        }
    catch(sock_ex ex)
    {
        cerr<<"Bad Socket in send_to_serv()"<<endl;throw(ex);
    }
    delete[] buffer;
}
```

This snippet is controls the hello messages. The first line is the call for a threaded process in main() and the following function uses the custom structure (my_thread) to access the instantiated clientConn (my_conn):

```cpp
int t1 = pthread_create(&threads[1],NULL,sendHello, (void *)&my_thread);
void *sendHello(void *thread_arg)
```
Appendix B: Average Prime Computation Evaluation

![Average Prime Computation Graph](image-url)
Appendix C: How Public Key Encryption and Decryption Works

Encryption:
1. If X wishes to send a message M to Y, Y transmits their public key (n, e) to X.
2. First, X turns M into a number m < n, by using a reversible protocol called a padding scheme.
3. After that the cipher text c is computed with the equation c = m^e mod n. X then transmits
   cipher text c to user Y.

Decryption:
1. Y can recover m from c by using Y’s private key exponent d with the equation m = c^d mod n.
2. Given m, Y can recover the original message M by reversing the original padding scheme.

Mathematical proof of decryption procedure:
c^d ≡ (m^n)^d ≡ m^{ed} mod n. Since, e and d are coprime, ed ≡ (mod p-1) and ed ≡ (mod q-1).

Using the definition of modulus, ed = k(p-1) +1 and ed = h (q-1) + 1 for satisfying values of k and h.

If m is not a multiple of p then m and p are coprime because p is prime; so by Fermat’s little
Theorem: m^{p-1} ≡ 1 mod p.

Substituting in for ed, m^{ed} = m^{k(p-1)+1} = m^{(p-1)k} m ≡ 1^k m = m mod p

If m is a multiple of p, m^{ed} ≡ 0 mod p.

Thus we can conclude: m^{ed} ≡ m mod q.

Since p and q are distinct prime numbers, they are coprime to each other, so since both primes
divide m^{ed} − m implies the product p*q divides m^{ed} − m, which means:
m^{ed} ≡ m (mod pq), and thus, c^d ≡ m mod n.