END TO END ENCRYPTION IN ANDROID CHAT APPLICATION USING RSA ALGORITHM.

Authors: Deb Biswas (1941012412) Pragya (1941012998) Riya Dutta(1941012185)

BRANCH :CSE
SECTION: C

AD 1 END TERM PROJECT

Abstract:

Information technology is an important aspect of human life that has provided comfort and ease in communicating. The most used communication technologies today is the smartphone, which is a mobile phone with the ability of a portable computer. Currently, to communicate with each other, people feel more convenience using chat application in a smartphone than calling or short message services (SMS) features. Some of the advantages of chat application are that there is no message size limitations, highest number of consumers use it; especially younger demographics, and it also works on mobile web without application download. Due to the convenience of communication using chat apps increases, the security demands are also higher, i.e., a proper cryptography scheme is needed to protect the messages. In this project, we have implemented RSA algorithm to secure text message in messaging application of a smartphone. We will adopt this method to create a chat application in Android program that equipped end-to-end encryption. We will also give the experimental result of our chat apps performance such as the accuracy of the received text message, average encryption and decryption time.

keywords-: RSA algorithm , encryption, decryption, cryptography.

INTRODUCTION:

In this era, technology has become an important aspect of human needs. Almost all their activities have been facilitated by technology, especially communication technology. Communication technology has always evolved over time and the most commonly used communication technology today is smartphones. The smartphone is a telecommunication device with a touch screen function that offers more capabilities than simply calling and sending messages, which also works as a portable computer that can be carried anywhere. Due to the capabilities of the smartphone, people do not have to rely on call and short message services (SMS) features anymore. They can use chat applications to communicate with each other. As it is easier and more convenient to communicate with smartphones, there are also new threats to digital data that raise issues related to security in online data exchange. This security effort is carried out with the aim of preserving the integrity and confidentiality of any information that stored or transmitted by one party to another. To maintain the security of the current data transmission in the communication process, there is an algorithm that usually handles the situation, i.e., cryptography. Cryptography is a field of study that studies the schemes of data encryptions and decryptions. These schemes are called cryptographic systems or ciphers. Encryption or also known as enciphering is a process of transforming a raw data or the original message, i.e. plaintext, into a coded message called cipher text. Moreover, the decryption (deciphering) is the inverse process of encryption that converts the cipher text back to the plaintext. Cryptography itself consists of several algorithms that developed by researchers, such as Caesar cipher, Rivest Shamir Adleman (RSA), data encryption standard (DES), triple data encryption standard (3DES), elliptic curve cryptography (ECC), and advanced encryption standard (AES). These algorithms can improve the security of a system with the objective of providing a level of complexity that is required by a specific program to safeguard the integrity and confidentiality of the data. Nowadays, among all the cryptography algorithms, only ECC and RSA are the most common algorithms for information security. The RSA algorithm was first introduced by Ron Rivest, Adi Shamir, and Leonard Adleman in 19782. RSA is motivated by the

published works of Diffie and Hellman in 19763. RSA which stands for Rivest, Shamir and Adleman, is an algorithm for public-key cryptography. They present an encryption method with the property that publicly revealing an encryption key does not thereby reveal the corresponding decryption key. This has two important consequences; the first one is couriers or other secure means are not needed to transmit keys, since a message can be enciphered using an encryption key publicly revealed by the intended recipient. Only he can decipher the message, since only he knows the corresponding decryption key. The second one is the message can be signed using a privately held decryption key. Anyone can verify this signature using the corresponding publicly revealed encryption key.

In this project, we implement the RSA algorithm to encrypt and decrypt text messages in chat applications of a smartphone. Currently, there are two kinds of platforms that usually installed and popularly used in smartphones, i.e., Android and iOS. Based on the Statistics survey 8 that conducted in December 2017, Android has dominated the smartphone market which its percentage had reached 88.37% compared to the other platforms. Therefore, our chat application is built on an Android smartphone with the help of Android Studio, the real-time database firebase, and the local storage room persistence library. Two novelties given in this project are first we implement the algorithm uses in Smartphones. Secondly we validate the encryption and decryption algorithms.

GOAL / OBJECTIVE:

- To create a chat application in Android program that equipped end-to-end encryption to secure text messages in smartphones.

CONDITIONS AND CONSTRAINTS:

- There must be an internet connection while using this application.
- The ASCII value must be less than 256.

BRIEF DESCRIPTION / PROPOSED SYSTEM:

Our proposed system includes four steps: key generation, key distribution, encryption, and decryption.

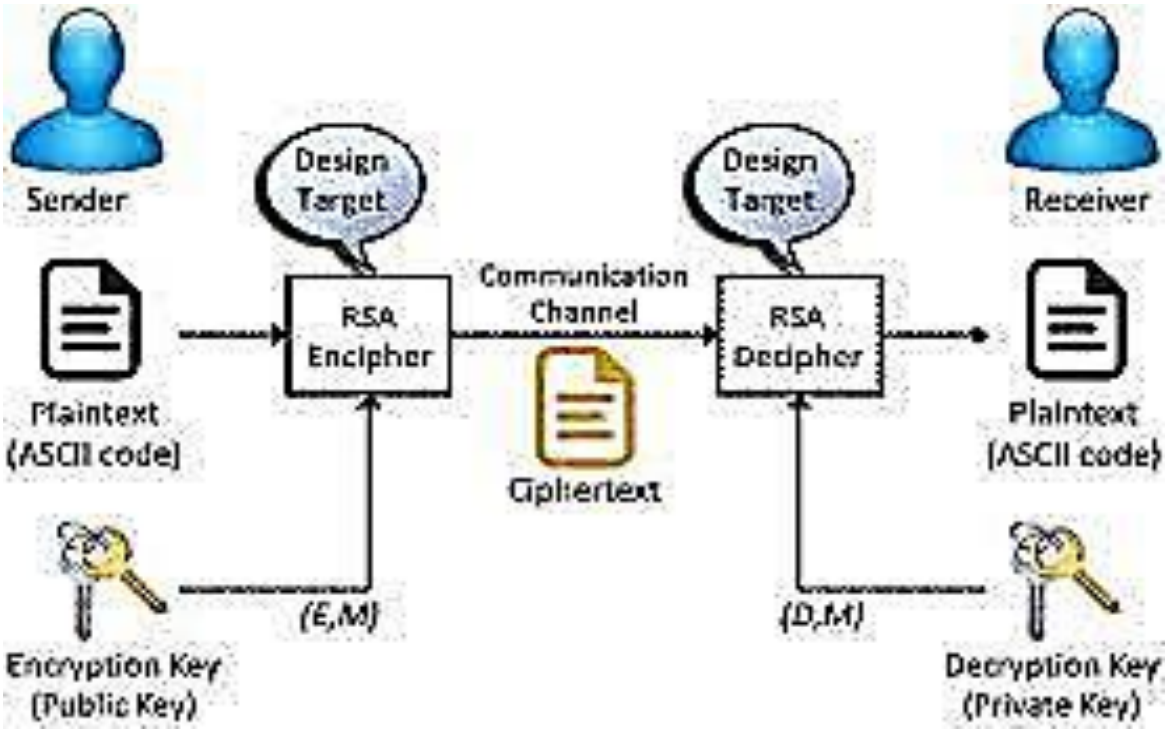
A basic principle behind RSA is the observation that it is practical to find three very large positive integers *e*, *d*, and *n*, such that with modular exponentiation for all integers *m* (with $0 \leq m < n$):

[(m^e)]^d equivalent to m(mod n)

and that knowing *e* and *n*, or even *m*, it can be extremely difficult to find *d*. The triple bar (≡) here denotes modular congruence. In addition, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies:

[(m^d)]^e equivalent to m(mod n)

RSA involves a *public key* and a *private key*. The public key can be known by everyone, and it is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time by using the private key. The public key is represented by the integers *n* and *e*; and, the private key, by the integer *d* (although *n* is also used during the decryption process, so it might be considered to be a part of the private key, too). *m* represents the message (previously prepared with a certain technique explained below).



Key generation:

The keys for the RSA algorithm are generated in the following way:

1. Choose two distinct prime numbers p and q

- For security purposes, the integers p and q should be chosen at random, and should be similar in magnitude but differ in length by a few digits to make factoring harder .Prime integers can be efficiently found using a primality test

- p and q are kept secret.

2. Compute $n = pq$.

- n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
- n is released as part of the public key.

3. Compute $\lambda(n)$, where λ is Carmichael's totient function. Since $n = pq$, $\lambda(n) = \text{lcm}(\lambda(p),\lambda(q))$, and since p and q are prime, $\lambda(p) = \varphi(p) = p - 1$ and likewise $\lambda(q) = q - 1$. Hence $\lambda(n) = \text{lcm}(p - 1, q - 1)$.

- $\lambda(n)$ is kept secret.
- The lcm may be calculated through the Euclidean algorithm, since $\text{lcm}(a,b) = |ab|/\gcd(a,b)$.

4. Choose an integer e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$; that is, e and $\lambda(n)$ are co-prime.

- e having a short bit-length and small Hamming weight results in more efficient encryption – the most commonly chosen value for e is $2^{16} + 1 = 65{,}537$. The smallest (and fastest) possible value for e is 3, but such a small value for e has been shown to be less secure in some settings.
- e is released as part of the public key.

5. Determine d as $d \equiv e^{-1} \pmod{\lambda(n)}$; that is, d is the modular multiplicative inverse of e modulo $\lambda(n)$.

- This means: solve for d the equation $d \cdot e \equiv 1 \pmod{\lambda(n)}$; d can be computed efficiently by using the Extended Euclidean algorithm, since, thanks to e and $\lambda(n)$ being coprime, said equation is a form of Bézout's identity, where d is one of the coefficients.
- d is kept secret as the *private key exponent*.

The *public key* consists of the modulus n and the public (or encryption) exponent e. The *private key* consists of the private (or decryption) exponent d, which must be kept secret. p, q, and $\lambda(n)$ must also be kept secret because they can be used to calculate d. In fact, they can all be discarded after d has been computed.

In the original RSA paper, the Euler totient function $\varphi(n) = (p - 1)(q - 1)$ is used instead of $\lambda(n)$ for calculating the private exponent d. Since $\varphi(n)$ is always divisible by $\lambda(n)$ the algorithm works as well. That the Euler totient function can be used can also be seen as a consequence of Lagrange's theorem applied to the multiplicative group of integers modulo pq. Thus any d satisfying $d \cdot e \equiv 1 \pmod{\varphi(n)}$ also satisfies $d \cdot e \equiv 1 \pmod{\lambda(n)}$. However, computing d modulo $\varphi(n)$ will sometimes yield a result that is larger than

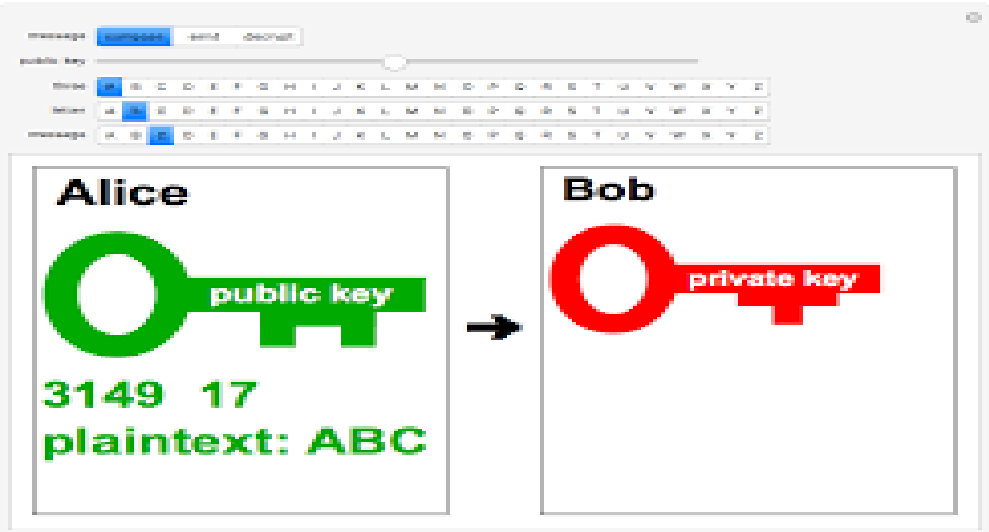
necessary (i.e. $d > \lambda(n)$). Most of the implementations of RSA will accept exponents generated using either method (if they use the private exponent d at all, rather than using the optimized decryption method based on the Chinese remainder theorem described below), but some standards such as FIPS 186-4 may require that $d < \lambda(n)$. Any "oversized" private exponents not meeting that criterion may always be reduced modulo $\lambda(n)$ to obtain a smaller equivalent exponent.

Since any common factors of $(p - 1)$ and $(q - 1)$ are present in the factorisation of $n - 1 = pq - 1 = (p - 1)(q - 1) + (p - 1) + (q - 1)$, it is recommended that $(p - 1)$ and $(q - 1)$ have only very small common factors, if any besides the necessary 2.

Note: The authors of the original RSA paper carry out the key generation by choosing d and then computing e as the modular multiplicative inverse of d modulo $\varphi(n)$, whereas most current implementations of RSA do the reverse (choose e and compute d). Since the chosen key can be small whereas the computed key normally is not, the RSA paper's algorithm optimizes decryption compared to encryption, while the modern algorithm optimizes encryption instead.

Key distribution:

Suppose that Bob wants to send information to Alice. If they decide to use RSA, Bob must know Alice's public key to encrypt the message and Alice must use her private key to decrypt the message.



To enable Bob to send his encrypted messages, Alice transmits her public key (*n*, *e*) to Bob via a reliable, but not necessarily secret, route. Alice's private key (*d*) is never distributed.

Encryption:

Encryption is the process of encoding a message so that messages can only be accessed by the authorized one

After Bob obtains Alice's public key, he can send a message *M* to Alice.

To do it, he first turns *M* (strictly speaking, the un-padded plaintext) into an integer *m* (strictly speaking, the padded plaintext), such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext *c*, using Alice's public key *e*, corresponding to

m^e equivalent to c (mod n)

This can be done reasonably quickly, even for very large numbers, using modular exponentiation. Bob then transmits *c* to Alice.

Decryption:

Alice can recover *m* from *c* by using her private key exponent *d* by computing

c^d equivalent to [(m^e)]^d equivalent to m(mod n)

MATHEMATICAL FORMULA:

Proof using Fermat's little theorem

The proof of the correctness of RSA is based on Fermat's little theorem stating that $a^{p-1} \equiv 1 \pmod{p}$ for any integer a and prime p, not dividing a.

We want to show that

(m^e)^d≡ m (mod pq)

for every integer m when p and q are distinct prime numbers and e and d are positive integers satisfying $ed \equiv 1 \pmod{\lambda(pq)}$.

Since $\lambda(pq) = \text{lcm}(p-1, q-1)$ is, by construction, divisible by both $p-1$ and $q-1$, we can write

ed-1=h(p-1)=k(q-1) for some nonnegative integers h and k.

To check whether two numbers, such as m^{ed} and m, are congruent mod pq, it suffices (and in fact is equivalent) to check that they are congruent mod p and mod q separately.

To show $m^{ed} \equiv m \pmod{p}$, we consider two cases:

1. If $m \equiv 0 \pmod{p}$, m is a multiple of p. Thus m^{ed} is a multiple of p. So $m^{ed} \equiv 0 \equiv m \pmod{p}$.
2. If m is not equivalent to 0 (mod p),

   m^ed = m^(ed-1)m = m^h(p-1)m = m^p(h-1)m ≡ (1^h)m ≡ m (mod p)

where we used Fermat's little theorem to replace $m^{p-1} \bmod p$ with 1.

The verification that $m^{ed} \equiv m \pmod{q}$ proceeds in a completely analogous way:

1. If $m \equiv 0 \pmod{q}$, m^{ed} is a multiple of q. So $m^{ed} \equiv 0 \equiv m \pmod{q}$.
2. If m is not equivalent to 0 (mod q),

   m^ed = m^(ed-1)m = m^k(q-1)m = m^q(k-1)m ≡ (1^k)m ≡ m (mod q)

This completes the proof that, for any integer m, and integers e, d such that $ed \equiv 1 \pmod{\lambda(pq)}$,

(m^e)^d ≡ m (mod pq).

Proof using Euler's theorem

Although the original paper of Rivest, Shamir, and Adleman used Fermat's little theorem to explain why RSA works, it is common to find proofs that rely instead on Euler's theorem.

We want to show that $m^{ed} \equiv m \pmod{n}$, where $n = pq$ is a product of two different prime numbers and e and d are positive integers satisfying $ed \equiv 1 \pmod{\varphi(n)}$. Since e and d are positive, we can write $ed = 1 + h\varphi(n)$ for some non-negative integer h.

More generally, for any e and d satisfying $ed \equiv 1 \pmod{\lambda(n)}$, the same conclusion follows from Carmichael's generalization of Euler's theorem which states that $m^{\lambda(n)} \equiv 1 \pmod{n}$ for all m relatively prime to n.

When m is not relatively prime to n, the argument just given is invalid. This is highly improbable (only a proportion of $1/p + 1/q - 1/(pq)$ numbers have this property), but even in this case, the desired congruence is still true. Either $m \equiv 0 \pmod{p}$ or $m \equiv 0 \pmod{q}$, and these cases can be treated using the previous proof.

EXAMPLES OF RSA :

Encryption Example:

In order to understand how encryption works when implemented we will practice an example using small prime factors. Remember the security in encryption relies not on the algorithm but on the difficulty of deciphering the key. Here is an example using the RSA encryption algorithm. Using RSA, choose p = 5 and q = 7, encode the phrase "hello". Apply the decryption algorithm to the encrypted version to recover the original plain text message. Using the RSA encryption method we get the following steps: 1) Choose two prime numbers p = 5 and q = 7 2) Compute n = pq and z = (p – 1)(q – 1) a. n = pq = (5)(7) = 35 b. z = (p – 1)(q – 1) = (5 – 1)(7 – 1) = (4)(6) = 24 3)

Choose a number e < n such that it has no common factors with z other than 1 a. Let e = 5 4) Find a number d such that ed divided z has a remainder of 1 a. Using the extended Euclidean algorithm to find the inverse modulo 35 find d = 29 5) The public key becomes K+ B or the number pair (n, e) and the private key becomes KB or the number pair (n, d) Thus, the encrypted value c of the plain text message m is: c = me mod n Now, using the alphabet such that the letters are numbered 1 through 26 we get :

Plain Text	**h**	**e**	**l**	**l**	**o**
Number	**8**	**5**	**12**	**12**	**15**
Encrypted Value	**8**	**10**	**17**	**17**	**15**

Where, 85 mod 35 = 8 55 mod 35 = 10 125 mod 35 = 17 125 mod 35 = 17 155 mod 35 = 15 This gives the encrypted value of the word “hello” as 8 10 17 17 15 where: h ! 8, e ! 10, l ! 17, l ! 17, 0 ! 15 In order to decrypt the message c, calculate: m = cd mod n Thus, we get:

Encrypted Value	**8**	**10**	**17**	**17**	**15**
Number	**8**	**5**	**12**	**12**	**15**
Plain Text	**h**	**e**	**l**	**l**	**o**

This gives the encrypted value of 8 10 17 17 15 as the word “hello” where: 8 -> h, 10 -> e, 17 -> l, 17 -> l, 15 -> o Which is the word “hello”. Of course if the message was encrypted using the ASCII values of the letters for the plain text, the lower and upper case letters could be differentiated.

TECHNOLOGY USED AND REQUIREMENTS OF THE APP:

Technology Used:

1. Android Studio for Java and XML Build

2. Firebase Authentication For Chat Login and Current User Reference

3. Firebase Real-Time Database for Users and Chat Storage

Requirements

1 .Android Operating System above Android 5.0 (Not Optimised for tablets)

2. Internet Connectivity

3. 1GB of RAM

4. 30MB of free storage

Limitations

1. Images cannot be send in the current stage of the app
2. Emojis or sticker cannot be encrypted In the current stage

CODE USING JAVA IN ANDROID STUDIO :

NOTE: THE FOLLOWING APP IS BUILD IN ANDROID STUDIO USING NATIVE JAVA

Encrypt and Decrypt Functions Used

P=17 q=19

DECRYPT:

Here d is the public key which is generated

```
public String decrpyt(String enmsg)
   {int p=17;
      int q=19;
      int n=p*q;
      ArrayList<Double>c=new ArrayList<Double>();
      int z=(p-1)*(q-1);
      ArrayList<BigInteger>msgback=new ArrayList<BigInteger>();
```

```
        int e=2;
        while(e<z)
        {
            if(gcd(e,z)==1)
            {
                break;

            }
            e++;

        }

        int d=0;
        System.out.println("value of e "+e);

        for (int i=0;i<=10;i++)
        {int f= 1+(i*z);


            if(f%e==0)
            {
                d=f/e;
                break;
            }




        }
        char[]encry_array=new char[enmsg.length()];

for(int i=0;i<enmsg.length();i++)
{encry_array[i]=enmsg.charAt( i );




}
        BigInteger N=BigInteger.valueOf(n);
```

```
ArrayList<BigInteger>C= new ArrayList<BigInteger>();
ArrayList<Double>ck=new ArrayList<Double>();
//ArrayList<Dr>original_msg=new ArrayList<>();
for (int i=0;i<encry_array.length;i++)
{char c1=encry_array[i];
    int number=c1;
  double douvalue=c1;
  ck.add( douvalue );




}
for(int i=0;i<ck.size();i++)
{C.add( BigDecimal.valueOf(ck.get(i)).toBigInteger());



}

for(int i=0;i<C.size();i++)
{
    msgback.add((C.get(i).pow(d)).mod(N));


}
System.out.println(msgback);

ArrayList<Integer>decrypted_msg=new ArrayList<Integer>();

for(int i=0;i<C.size();i++)
{
    decrypted_msg.add(msgback.get(i).intValue());


}

char []finaldec=new char[decrypted_msg.size()];

for(int i=0;i<C.size();i++)
{
    int k=decrypted_msg.get(i);
```

```
            char msg_char=(char)k;
            finaldec[i]=msg_char;

        }
        String returnmsg="";
        for(int i=0;i<finaldec.length;i++)
        {
            returnmsg+=finaldec[i];



        }

return returnmsg;
    }
```

ENCRYPT:

HERE e is the private key which is genrated

```
public String encrypt(String msg) {

    int p = 17;
    int q = 19;
    int n = p * q;
    ArrayList<Double> c = new ArrayList<Double>();
    int z = (p - 1) * (q - 1);
    ArrayList<BigInteger> msgback = new ArrayList<BigInteger>();

    char[] msgarray = new char[msg.length()];
    for (int i = 0; i < msgarray.length; i++) {
        msgarray[i] = msg.charAt( i );



    }

    ArrayList<Integer> original_msg = new ArrayList<>();
    for (int i = 0; i < msg.length(); i++) {
        char c1 = msgarray[i];
        int number = c1;
        original_msg.add( number );
```

```
}
System.out.println( original_msg );
System.out.println( "Value of z " + z );


int e = 2;
while (e < z) {
    if (gcd( e, z ) == 1) {
        break;

    }
    e++;

}

int d = 0;
System.out.println( "value of e " + e );

for (int i = 0; i <= 10; i++) {
    int f = 1 + (i * z);


    if (f % e == 0) {
        d = f / e;
        break;
    }


}

System.out.println( "Value of d " + d );
//encryption
for (int i = 0; i < original_msg.size(); i++) {
    c.add( (Math.pow( original_msg.get( i ), e )) % n );

}
System.out.println( c );
char[] encry_array = new char[original_msg.size()];

for (int i = 0; i < original_msg.size(); i++) {
    double k = c.get( i );
    int value = (int) k;
```

```
        char msg_encrchar = (char) value;
        encry_array[i] = msg_encrchar;

    }
    //System.out.println("Encrypted msg is "+String.valueOf(encry_array));

    String s = "";
    for (int i = 0; i < encry_array.length; i++) {
        s = s + encry_array[i];
    }
    return s;
}
```

SENDING THE MESSAGE IN ENCRYPTED FORMAT TO FIREBASE DATABASE

```
  send.setOnClickListener( new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            String msg = msg_edit.getText().toString();
            if (!msg.equals( "" )) {
                String msg2 = encrypt( msg );//encrypted function is called
                sendMessage( fuser.getUid(), userid, msg2 );


            } else {
                Toast.makeText( MessageActivity.this, "Please send non empty",
Toast.LENGTH_SHORT ).show();
            }
            msg_edit.setText( "" );
        }
    } );
}
```

```
private void sendMessage(String sender, String receiver, String message) {
        DatabaseReference reference =
FirebaseDatabase.getInstance().getReference();

        HashMap<String, Object> hashmap = new HashMap<>();
        hashmap.put( "sender", sender );
```

```
        hashmap.put( "receiver", receiver );
        hashmap.put( "message", message );
        reference.child( "Chats" ).push().setValue( hashmap );

//
        final DatabaseReference chatRef =
FirebaseDatabase.getInstance().getReference( "ChatList" ).child( fuser.getUid()
).child( userid );


        chatRef.addListenerForSingleValueEvent( new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (!snapshot.exists()) {
                    chatRef.child( "id" ).setValue( userid );
                }
            }

            @Override
            public void onCancelled(@NonNull DatabaseError error) {

            }
        } );


    }
```

Here when the send button is clicked the string gets converted to encrypted string by encrypt function which is then send to sendMessage Function from which the encrypted message is send to database.

GETTING THE DATA FROM DATABASE AND DISPLAYING IT TO USER

```
public void onDataChange(@NonNull DataSnapshot snapshot) {
    Users user = snapshot.getValue( Users.class );
    username.setText( user.getUsername() );
    readMessage( fuser.getUid(), userid );
}
```

```
private void readMessage(final String myid,
                         final String userid) {


    mchat = new ArrayList<>();
    reference = FirebaseDatabase.getInstance().getReference( "Chats" );
    reference.addValueEventListener( new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot) {
            mchat.clear();
            String f;

            for (DataSnapshot snapshot1 : snapshot.getChildren()) {
                Chat chat = snapshot1.getValue( Chat.class );
                if (chat.getReceiver() != null && chat.getSender() != null) {
                    if ((chat.getReceiver().equals( myid ) && chat.getSender().equals(
userid )) || (chat.getReceiver().equals( userid ) && chat.getSender().equals(
myid ))) {
                        String fk=chat.getMessage();//MESSAGE TAKEN FROM
DATABASE
                        String msgdecrypted=decrpyt( fk );//decrypted function is called
                        Chat c2=new Chat( chat.getSender(),
chat.getReceiver(),msgdecrypted);

                        mchat.add( c2 );

                    }
                }
                messageAdapter = new MessageAdapter( MessageActivity.this,
mchat );
                recyclerView.setAdapter( messageAdapter );
            }
        }
```

When a new message is stored at database it is been decrypted and displayed to the user

FINALLY THE FULL CHAT ACTIVITY OF THE APP COMBINED USING ABOVE FUNCTIONS AND OTHER NECESSARY FUNCTIONS

```
package com.example.dchat;
```

```
import android.annotation.SuppressLint;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;
import android.widget.Toolbar;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.example.dchat.Adapter.MessageAdapter;
import com.example.dchat.Model.Chat;
import com.example.dchat.Model.Users;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class MessageActivity extends AppCompatActivity {
    TextView username;
    FirebaseUser fuser;
    RecyclerView recyclerView;
    DatabaseReference reference;
    EditText msg_edit;
    Button send;
    MessageAdapter messageAdapter;
    List<Chat> mchat;
    Intent intent;
    String userid;
```

```
    //RecyclerView recyclerView;
    @SuppressLint("RestrictedApi")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate( savedInstanceState );
        setContentView( R.layout.activity_message );
        if (getSupportActionBar() != null) {
            getSupportActionBar().hide();
        }
        username = findViewById( R.id.us1 );
        send = findViewById( R.id.sendbtn );
        msg_edit = findViewById( R.id.send_text );
        recyclerView = findViewById( R.id.rc2 );
        recyclerView.setHasFixedSize( true );

        LinearLayoutManager linearLayoutManager = new LinearLayoutManager(
getApplicationContext() );
        linearLayoutManager.setStackFromEnd( true );
        recyclerView.setLayoutManager( linearLayoutManager );



        intent = getIntent();
        userid = intent.getStringExtra( "userid" );
        fuser = FirebaseAuth.getInstance().getCurrentUser();
        reference = FirebaseDatabase.getInstance().getReference( "MyUsers"
).child( userid );
        reference.addValueEventListener( new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                Users user = snapshot.getValue( Users.class );
                username.setText( user.getUsername() );
                readMessage( fuser.getUid(), userid );
            }

            @Override
            public void onCancelled(@NonNull DatabaseError error) {

            }
        } );
        send.setOnClickListener( new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
            String msg = msg_edit.getText().toString();
            if (!msg.equals( "" )) {
                String msg2 = encrypt( msg );//encrypted function is called
                sendMessage( fuser.getUid(), userid, msg2 );


            } else {
                Toast.makeText( MessageActivity.this, "Please send non empty",
Toast.LENGTH_SHORT ).show();
            }
            msg_edit.setText( "" );
        }
    } );
  }

  private void setSupportActionBar(Toolbar toolbar) {
  }
  public String decrpyt(String enmsg)
  {int p=17;
    int q=19;
    int n=p*q;
    ArrayList<Double>c=new ArrayList<Double>();
    int z=(p-1)*(q-1);
    ArrayList<BigInteger>msgback=new ArrayList<BigInteger>();




    int e=2;
    while(e<z)
    {
      if(gcd(e,z)==1)
      {
        break;

      }
      e++;

    }

    int d=0;
    System.out.println("value of e "+e);
```

```
for (int i=0;i<=10;i++)
{int f= 1+(i*z);


   if(f%e==0)
   {
      d=f/e;
      break;
   }




}
char[]encry_array=new char[enmsg.length()];

for(int i=0;i<enmsg.length();i++)
{encry_array[i]=enmsg.charAt( i );




}
BigInteger N=BigInteger.valueOf(n);

ArrayList<BigInteger>C= new ArrayList<BigInteger>();
ArrayList<Double>ck=new ArrayList<Double>();
//ArrayList<Dr>original_msg=new ArrayList<>();
for (int i=0;i<encry_array.length;i++)
{char c1=encry_array[i];
   int number=c1;
   double douvalue=c1;
   ck.add( douvalue );




}
for(int i=0;i<ck.size();i++)
{C.add( BigDecimal.valueOf(ck.get(i)).toBigInteger());
```

```
    }

    for(int i=0;i<C.size();i++)
    {
        msgback.add((C.get(i).pow(d)).mod(N));


    }
    System.out.println(msgback);

    ArrayList<Integer>decrypted_msg=new ArrayList<Integer>();

    for(int i=0;i<C.size();i++)
    {
        decrypted_msg.add(msgback.get(i).intValue());


    }

    char []finaldec=new char[decrypted_msg.size()];

    for(int i=0;i<C.size();i++)
    {
        int k=decrypted_msg.get(i);
        char msg_char=(char)k;
        finaldec[i]=msg_char;

    }
    String returnmsg="";
    for(int i=0;i<finaldec.length;i++)
    {
        returnmsg+=finaldec[i];



    }

    return returnmsg;
}
public String encrypt(String msg) {
```

```
int p = 17;
int q = 19;
int n = p * q;
ArrayList<Double> c = new ArrayList<Double>();
int z = (p - 1) * (q - 1);
ArrayList<BigInteger> msgback = new ArrayList<BigInteger>();

char[] msgarray = new char[msg.length()];
for (int i = 0; i < msgarray.length; i++) {
    msgarray[i] = msg.charAt( i );


}

ArrayList<Integer> original_msg = new ArrayList<>();
for (int i = 0; i < msg.length(); i++) {
    char c1 = msgarray[i];
    int number = c1;
    original_msg.add( number );


}
System.out.println( original_msg );
System.out.println( "Value of z " + z );


int e = 2;
while (e < z) {
    if (gcd( e, z ) == 1) {
        break;

    }
    e++;

}

int d = 0;
System.out.println( "value of e " + e );

for (int i = 0; i <= 10; i++) {
    int f = 1 + (i * z);
```

```
            if (f % e == 0) {
                d = f / e;
                break;
            }


        }

        System.out.println( "Value of d " + d );
        //encryption
        for (int i = 0; i < original_msg.size(); i++) {
            c.add( (Math.pow( original_msg.get( i ), e )) % n );

        }
        System.out.println( c );
        char[] encry_array = new char[original_msg.size()];

        for (int i = 0; i < original_msg.size(); i++) {
            double k = c.get( i );
            int value = (int) k;
            char msg_encrchar = (char) value;
            encry_array[i] = msg_encrchar;

        }
        //System.out.println("Encrypted msg is "+String.valueOf(encry_array));

        String s = "";
        for (int i = 0; i < encry_array.length; i++) {
            s = s + encry_array[i];
        }
        return s;
    }
    private void sendMessage(String sender, String receiver, String message) {
        DatabaseReference reference =
FirebaseDatabase.getInstance().getReference();

        HashMap<String, Object> hashmap = new HashMap<>();
        hashmap.put( "sender", sender );
        hashmap.put( "receiver", receiver );
        hashmap.put( "message", message );
        reference.child( "Chats" ).push().setValue( hashmap );

//
```

```
        final DatabaseReference chatRef =
FirebaseDatabase.getInstance().getReference( "ChatList" ).child( fuser.getUid()
).child( userid );


        chatRef.addListenerForSingleValueEvent( new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                if (!snapshot.exists()) {
                    chatRef.child( "id" ).setValue( userid );
                }
            }

            @Override
            public void onCancelled(@NonNull DatabaseError error) {

            }
        } );


    }

    static int gcd(int e, int z) {
        if (e == 0)
            return z;
        else
            return gcd( z % e, e );
    }



    private void readMessage(final String myid,
                             final String userid) {


        mchat = new ArrayList<>();
        reference = FirebaseDatabase.getInstance().getReference( "Chats" );
        reference.addValueEventListener( new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot snapshot) {
                mchat.clear();
                String f;
```

```
            for (DataSnapshot snapshot1 : snapshot.getChildren()) {
                Chat chat = snapshot1.getValue( Chat.class );
                if (chat.getReceiver() != null && chat.getSender() != null) {
                    if ((chat.getReceiver().equals( myid ) &&
chat.getSender().equals( userid )) || (chat.getReceiver().equals( userid ) &&
chat.getSender().equals( myid ))) {
                        String fk=chat.getMessage();//MESSAGE TAKEN  FROM
DATABASE
                        String msgdecrypted=decrpyt( fk );//decrypt function is called
                        Chat c2=new Chat( chat.getSender(),
chat.getReceiver(),msgdecrypted);

                        mchat.add( c2 );

                    }
                }
                messageAdapter = new MessageAdapter( MessageActivity.this,
mchat );
                recyclerView.setAdapter( messageAdapter );
            }
        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    } );


  }

}
```

RESULT ANALYSIS:

USER 1

DATA SEND AND RECIEVED BY APP

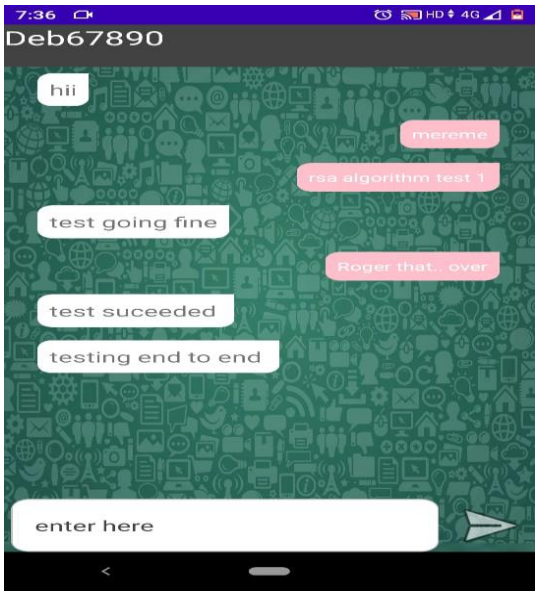


IMAGE 1 : DATA SEND AND RECEIVED BY USER 1

USER 2

DATA SEND AND RECEIVED

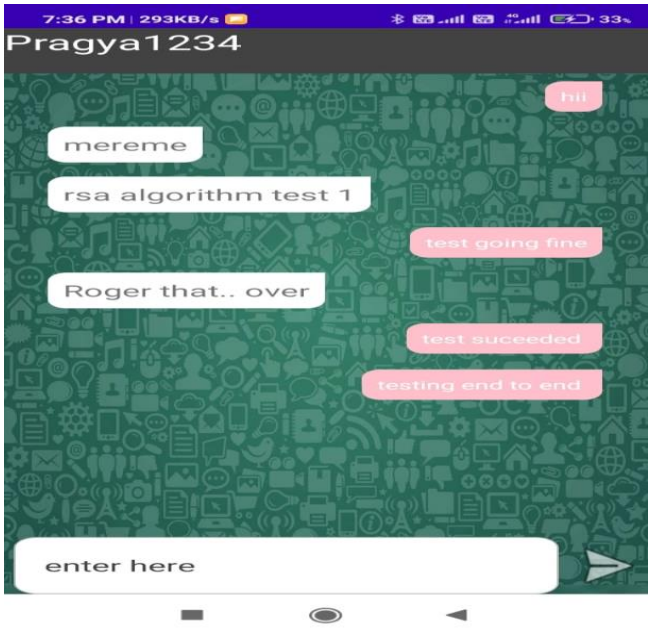
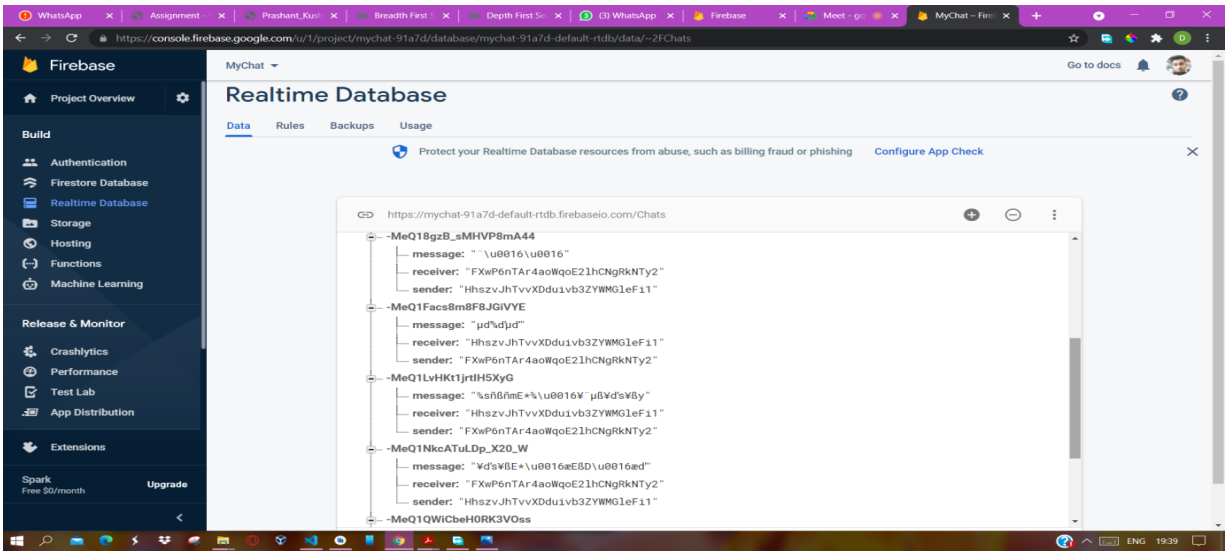


IMAGE 2:DATA SEND AND RECEIVED BY USER 2

HERE THE DATA RECEIVED AND SEND BY THE BOTH THE USER HAD BEEN ENCRYPTED AND STORED IN THE DATABASE AND DATA GETS DECRYTED AT THE USER END .

AS YOU CAN SEE BELOW THE DATA IS STORED IN AN ENCRYTED MANNER BY RSA ALGORITHM WITH TWO PRIMES AS 17 AND 19



REAL- LIFE APPLICATIONS:

When transmitting electronic data, the most common use of cryptography is to encrypt and decrypt email and other plain-text messages. The simplest method uses the symmetric or “secret key” system. Here, data is encrypted using a secret key, and then both the encoded message and secret key are sent to the recipient for decryption.

The problem? If the message is intercepted, a third party has everything they need to decrypt and read the message. To address this issue, cryptologists devised the asymmetric or “public key” system. In this case, every user has

two keys: one public and one private. Senders request the public key of their intended recipient, encrypt the message and send it along. When the message arrives, only the recipient's private key will decode it — meaning theft is of no use without the corresponding private key.

Encryption in Whatsapp :

WhatsApp uses the 'signal' protocol for encryption, which uses a combination of asymmetric and symmetric key cryptographic algorithms. The symmetric key algorithms ensure confidentiality and integrity whereas the asymmetric key cryptographic algorithms help in achieving the other security goals namely authentication and non-repudiation. In symmetric key cryptography a single key is used for encryption of the data as well as decryption. In asymmetric key cryptography there would be two separate keys. The data which is encrypted using the public key of a user can only be decrypted using the private key of that user and vice versa.

Whatsapp uses the Curve25519 based algorithm. The history of Curve25519 is worth noting as it was introduced after the concerns over allegations that certain parameters of the previously prevalent P-256 NIST standards have been manipulated by NSA for easier snooping. Elliptic Curve Diffie Hellman algorithm is a mathematical algorithm which helps two communicating entities

to agree up on a shared secret without actually sending the actual keys to each other.

Encryption in Instagram:

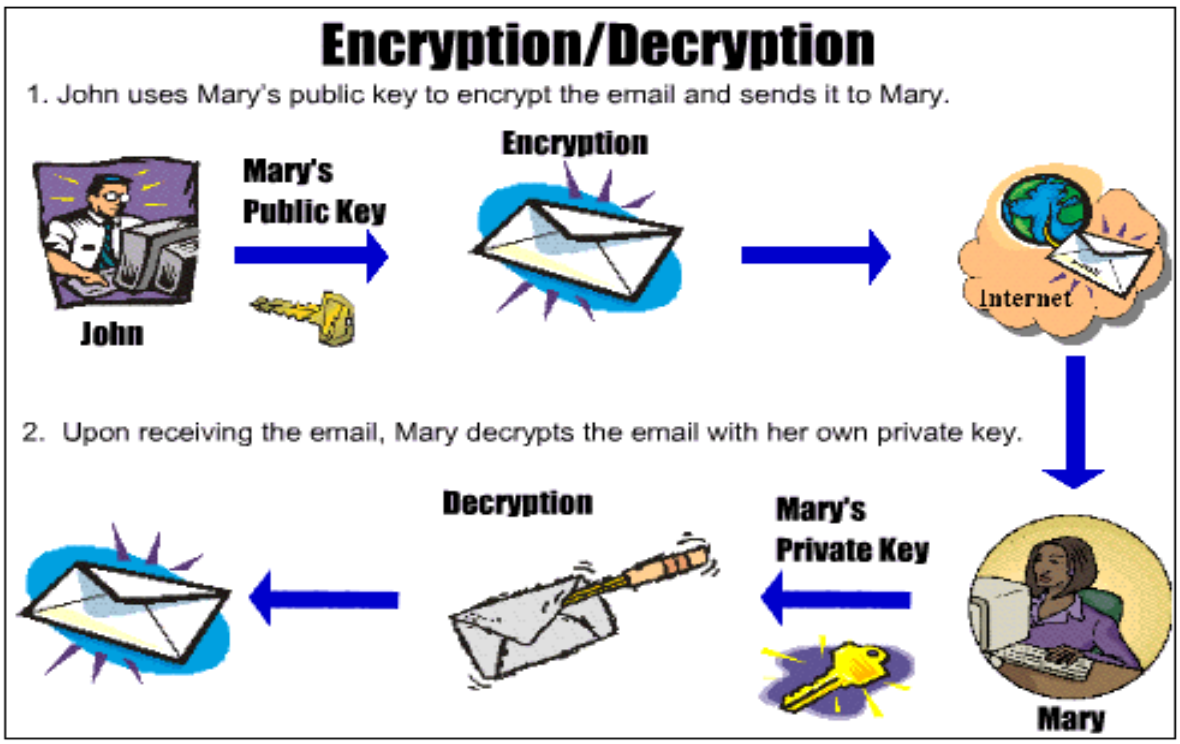
Your interaction with Instagram is likely an encrypted communication. When your phone requests data with instagram it will use SSL/TLS over port 443 to encrypt requests from Instagram servers and will send you data over the same encrypted data stream. This prevents malicious parties from eavesdropping on the conversation between you and instagram.

Encryption / Decryption in email :

Email encryption is a method of securing the content of emails from anyone outside of the email conversation looking to obtain a participant's information. In its encrypted form, an email is no longer readable by a human. Only with your private email key can your emails be unlocked and decrypted back into the original message.

Email encryption works by employing something called public key cryptography. Each person with an email address has a pair of keys

associated with that email address, and these keys are required in order to encrypt or decrypt an email. One of the keys is known as a “public key”, and is stored on a keyserver where it is tied to your name and email address and can be accessed by anyone. The other key is your private key, which is not shared publicly with anyone.



When an email is sent, it is encrypted by a computer using the public key and the contents of the email are turned into a complex, indecipherable scramble that is very difficult to crack. This public key cannot be used to decrypt the sent message, only to encrypt it. Only the person with the proper corresponding private key has the ability to decrypt the email and read its contents.

There are various types of email encryption, but some of the most common encryption protocols are:

- **OpenPGP** — a type of PGP encryption that utilizes a decentralized, distributed trust model and integrates well with modern web email clients
- **S/MIME** — a type of encryption that is built into most Apple devices and utilizes a centralized authority to pick the encryption algorithm and key size

Email encryption services can be used to provide encryption in a few separate but related areas:

- The connection between email providers can be encrypted, preventing outside attackers from finding a way to intercept any incoming or outgoing emails as they travel between servers
- The content of the email can be encrypted, ensuring that even if an email is intercepted by an attacker, the contents of the email will still be entirely unreadable
- Old or archived emails that are already stored within your email client should also be encrypted to prevent attackers from potentially gaining access to emails that aren't currently in transit between servers

STATEMENT OF DECLARATION / CONCLUSION:

We present our proposed system to create an android based chat application that comes with end-to-end encryption. The form of chat application using the RSA algorithm for text encryption and decryption . In this research It proves that the RSA algorithm can be implemented and has competitive performance in both time and accuracy. However, the application of RSA in the Android app still needs to be optimized for better results. For the future, therefore, the optimization of ECC methods can be implemented and also performed in the encryption of images and videos.