**CHAPTER FIVE**

**CONCLUSION AND RECOMMENDATIONS**

**5.1 Summary**

Currently in Nigeria, many organizations such as commercial banks (Oceanic Bank, Zenith Bank, UBA etc...), Nigerian Football Federation and some other organizations use SMS to transmit sensitive information over the GSM network. Also some hospitals, especially the private ones, use SMS to transfer health related information to patients as plain text in Nigeria. Moreover, many people use SMS to send sensitive data without knowing the security vulnerability of the GSM network. Transmitting plain SMS is not secure because the service provider keeps the record of all transmitted messages in their database without any security measure. Therefore a secure SMS messaging solution is needed.

As we know mobile phones and PDAs (personal digital assistants) have constrained environments and are highly limited battery life. Therefore, finding the appropriate algorithms that will not increase the need for processing resources (memory, processor and power) of the device is the main issue in mobile application development. As mentioned in Chapter 2, ECC asymmetric key encryption is more suitable encryption algorithm for mobile environment, because it does not heighten the need for much processing resources of the device, hence, ECC was used to transmit one time password. Blowfish symmetric key encryption algorithm used to encrypt the SMS message on this system.

The testing conducted shows that the basic mobile app requirements for speed and compactness were met. The program size is less than 15MB and it can be installed into a mobile phone working on Android platform. The user experiences no delays while using the program, which is a clear indication that the speed requirement is met. It was made sure that the user interface is simple and straight forward to use. In mobile applications, access control is vital, the developed application can be used to authenticate the sender of a message. Also the encryption implemented makes it possible to protect the SMS, from being corrupted or tampered with during transmission. Most importantly, the messages containing delicate information are stored securely and remain undisclosed even when the device is accessed by a person unauthorized or even an adversary. The most unique and vital point that was considered is the security of the encrypted message against various attacks such as Brute Force attack, pattern attack etc. This application guarantees secure end to end transfer of SMS without any corrupt data segments.

**5.2 Conclusion**

The study is motivated by the fact of ensuring privacy and security in SMS systems, but solution cost was also taken into account vigorously. Several relevant works were analyzed for gathering knowledge to accomplish this research. This is an application layer protocol and low cost approach in context of computational and implementation scenario. For any SMS service, this approach can be used to ensure privacy and security. However, the operators and handset /mobile phones providers need to be aligned with this approach and cumulative understanding can ensure secured messaging with high privacy.

In the proposed solution, Hybrid encryption was used, which is the combination of symmetric and asymmetric key encryption. Asymmetric key encryption (ECC) used to encrypt the one time password which is used as secret key to symmetric cryptography. Symmetric cryptography (Blowfish) was implemented for encrypt and decrypt the original SMS.

The designed system provides a secure communication platform between two mobile phones over GSM network. This system was implemented using android studio which runs with java programming language and therefore the mobile application software can be installed on any android operating system based mobile phone. Based on the results of the discussion in the previous chapters in this thesis report, it can be concluded as follows:

1. Blowfish algorithm can be applied to applications based on Android SMS Cryptography.

2. SMS Cryptography Applications can be run on Android versions as low as version 4.2.2 (Kitkat), the Android version 4.1.2 (Jelly Bean), Android Version 3.2 (Icecream Sandwich), the Android version 3.0 (Honey Comb), Android Version 2.3 (Ginger Bread). Besides emulators, the application was also tested on real mobile phone with newer android OS versions 6, 7 and 8 such as Samsung, HTC, Tecno etc.

**5.3 Recommendation and Future Work**

The proposed system can be easily adapted to support secure SMS messaging solutions for many applications such as SMS based (USSD) banking, SMS based health and SMS based voting. This solution is highly recommend to any commercial company and government organizations, which need confidential information transmitted over the GSM network. The work has been done by sending the shared or secret key, which is used to encrypt and to decrypt the SMS message, together with the message. To send the secrete key along with the message encrypted by using ECC asymmetric encryption mechanism

As we know one of the drawback of this asymmetric encryption is it produce larger size cipher or encrypted text than the original text. Thus, one of the future work suggested is to research on how the size of SMS content to be sent can be decreased as a single SMS message by using some compression technique on key encryption/ decryption part of the proposed system.

For further algorithm implementation, the study is also looking forward for possibilities that the ECC and Blowfish algorithms implemented can also be applied and implemented in different environments not only in SMS communication lines but also on email, documents, social media and other transactional data for privacy and security purposes. For further enhancement, the security framework may also be compared to other encryption algorithms.

**APPENDICES**

**ECC Encryption Code**

public static byte[] encrypt(byte[] data, byte[] publicKey) {

try { boolean failure = true;

IESEngine engine = new IESEngine(new ECDHBasicAgreement(), new KDF2BytesGenerator(new SHA1Digest()), new HMac(new SHA1Digest()));

ECPoint publicPoint = ecparms.getCurve().decodePoint(publicKey);

ECPublicKeyParameters pubKey = new ECPublicKeyParameters(publicPoint, ecparms);

ECPrivateKeyParameters k = null;

ECPublicKeyParameters R = null; ECPoint Z = null; initPRNG(); while (failure) {

BigInteger privNum = new BigInteger(PRNG.generateSeed(NONCE\_SIZE)); // select k k = new ECPrivateKeyParameters(privNum, ecparms); // this is just a container

R = generatePublicKey(k); // R = kP

// calculate Z=hkQ

BigInteger z = new BigInteger(ecparms.getH().toByteArray()); // z=h z = z.multiply(privNum); // z=hk Z = pubKey.getQ(); // Z=Q if (Z instanceof ECPoint.F2m) {

ECPoint.F2m Z2 = new ECPoint.F2m(Z.getCurve(), Z.getX(), Z.getY()); // clone Z = Z2.multiply(z); // Z=zQ <=> Z=hkQ

}

if (Z instanceof ECPoint.Fp) {

ECPoint.Fp Z2 = new ECPoint.Fp(Z.getCurve(), Z.getX(), Z.getY()); // clone

Z = Z2.multiply(z); // Z=zQ <=> Z=hkQ

}

if (!Z.isInfinity()) { failure = false; // see step 2

} }

// step 3 and 4:

// setup KDF(xz,R) - see new IESParameters(...)

// pass the parameters for step4 engine.init(true, (CipherParameters) k, (CipherParameters) pubKey, new IESParameters(Z.getX().toBigInteger().toByteArray(), R.getQ().getEncoded(), 64));

// step 4 generate C and t byte[] cAndT = engine.processBlock(data, 0, data.length);

// collect the result in format

// byte[0] = length of R (encoded)

// R in encoded form

// C and t as coming from the engine byte[] publicBytes = R.getQ().getEncoded(); byte[] out = new byte[1 + publicBytes.length + cAndT.length]; out[0] = (byte) publicBytes.length; // WARN this will crash if

// key.length > 255

System.arraycopy(publicBytes, 0, out, 1, publicBytes.length);

System.arraycopy(cAndT, 0, out, 1 + publicBytes.length, cAndT.length); out1 = new byte[1 + publicBytes.length + cAndT.length];

out11 = out; pr = publicKey; return out;

} catch (Exception excep) { return null;

} catch (Throwable t) { return null; } }

**ECC Decryption Code**

public static byte[] decrypt(byte[] data, AsymmetricCipherKeyPair pair) throws InvalidCipherTextException {

IESEngine engine = new IESEngine(new ECDHBasicAgreement(), new KDF2BytesGenerator(new SHA1Digest()), new HMac( new SHA1Digest()));

ECPrivateKeyParameters p = (ECPrivateKeyParameters) pair.getPrivate();

// deserialize data into (R,C,t) int lengthOfR = data[0]; if (lengthOfR > data.length) { throw new InvalidCipherTextException("Lengthfield invalid");

}

byte[] RasBytes = new byte[lengthOfR];

System.arraycopy(data, 1, RasBytes, 0, lengthOfR);

ECPoint RPoint = ecparms.getCurve().decodePoint(RasBytes); // decode into R

ECPublicKeyParameters R = new ECPublicKeyParameters(RPoint, ecparms); // this is just a container byte[] cAndT = new byte[data.length - lengthOfR - 1];

System.arraycopy(data, 1 + lengthOfR, cAndT, 0, data.length - lengthOfR- 1);

// compute Z=hdR (step 2)

BigInteger z = new BigInteger(ecparms.getH().toByteArray()); // z=h z = z.multiply(p.getD()); // z=hd ECPoint Z = R.getQ(); // Z=Q if (Z instanceof ECPoint.F2m) {

ECPoint.F2m Z2 = new ECPoint.F2m(Z.getCurve(), Z.getX(), Z.getY()); // clone Z = Z2.multiply(z); // Z=zQ <=> Z=hdQ

}

if (Z instanceof ECPoint.Fp) {

ECPoint.Fp Z2 = new ECPoint.Fp(Z.getCurve(), Z.getX(), Z.getY()); // clone Z = Z2.multiply(z); // Z=zQ <=> Z=hdQ

}

if (Z.isInfinity()) { throw new InvalidCipherTextException("Z is infinite"); // reject if Z is infinite (see step 2)

}

// step 3

// setup KDF(xz,R) - see new IESParameters(...)

// pass the parameters for step4 engine.init(false, (CipherParameters) p, (CipherParameters) R, new IESParameters(Z.getX().toBigInteger().toByteArray(), R.getQ().getEncoded(), 64));

// step 4 and 5 byte[] clear = engine.processBlock(cAndT, 0, cAndT.length);

return clear;

}

**Blowfish Encryption and Decryption Code**

public Encryptor( byte[] key ){ cipher = new PaddedBlockCipher( new CBCBlockCipher( new BlowfishEngine() ) ); this.key = new KeyParameter( key );

}

// Initialize the cryptographic engine. // The string should be at least 8 chars long. public Encryptor( String key ){ this( key.getBytes() );

}

// Private routine that does the gritty work. private byte[] callCipher( byte[] data ) throws CryptoException { int size = cipher.getOutputSize( data.length ); byte[] result = new byte[ size ]; int olen = cipher.processBytes( data, 0,data.length, result, 0 ); olen += cipher.doFinal( result, olen ); if( olen < size ){ byte[] tmp = new byte[ olen ]; System.arraycopy( result, 0, tmp, 0, olen ); result = tmp;

}

return result;

}

// Encrypt arbitrary byte array, returning the // encrypted data in a different byte array.

public synchronized byte[] encrypt( byte[] data ) throws CryptoException { if( data == null || data.length == 0 ){ return new byte[0];

}

cipher.init( true, key );

System.out.print("encrypted byte:"+callCipher( data ));

System.out.print("encrypted String:"+new String(callCipher( data ))); return callCipher( data );

}

// Encrypts a string.

public byte[] encryptString( String data ) throws CryptoException { if( data == null || data.length() == 0 ){ return new byte[0];

}

return encrypt( data.getBytes() );

}

// Decrypts arbitrary data.

public synchronized byte[] decrypt( byte[] data ) throws CryptoException { if( data == null || data.length == 0 ){ return new byte[0];

}

cipher.init( false, key ); return callCipher( data );

}

// Decrypts a string that was previously encoded

// using encryptString. public String decryptString( byte[] data ) throws CryptoException { if( data == null || data.length == 0 ){ return "";

}

return new String( decrypt( data ) );

}