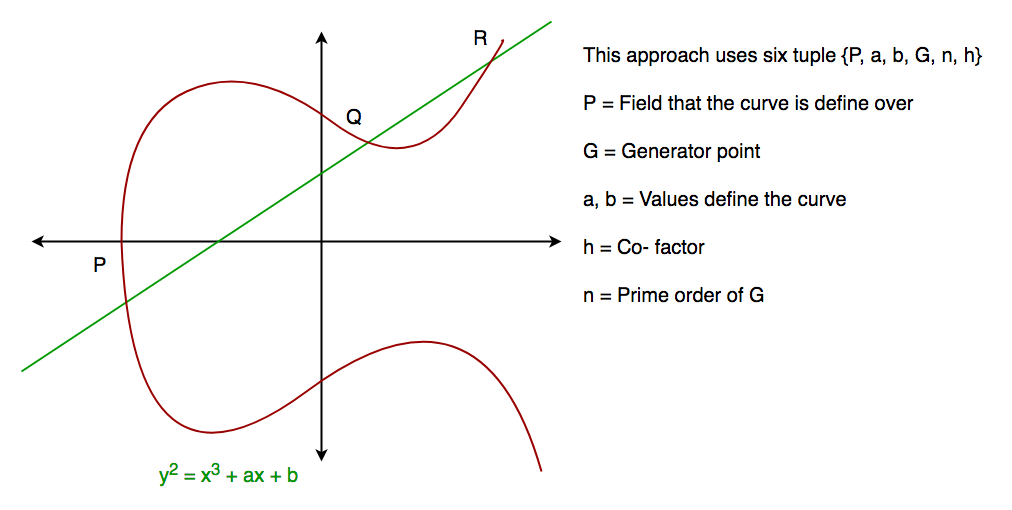
**Elliptic Curve Cryptography (ECC)** is an approach to public-key cryptography, based on the algebraic structure of elliptic curves over finite fields. ECC requires a smaller key as compared to non-ECC cryptography to provide equivalent security (a 256-bit ECC security has equivalent security attained by 3072-bit RSA cryptography).

For a better understanding of Elliptic Curve Cryptography, it is very important to understand the basics of the Elliptic Curve. An elliptic curve is a planar algebraic curve defined by an equation of the form

Where ‘a’ is the co-efficient of x and ‘b’ is the constant of the equation

The curve is non-singular; that is, its graph has no cusps or self-intersections (when the characteristic of the Co-efficient field is equal to 2 or 3).

In general, an elliptic curve looks like as shown below. Elliptic curves can intersect almost 3 points when a straight line is drawn intersecting the curve. As we can see, the elliptic curve is symmetric about the x-axis. This property plays a key role in the algorithm.



**Implementation code**

**Libraries and APIs**

Since version 7 Java supports various elliptic curves. If you have jshell installed on your machine, you can easily get a list of all supported curves by executing the following Java code in jshell.

jshell> import java.security.\*;  
jshell> Security.getProvider("SunEC").getService(  
...> "AlgorithmParameters", "EC").getAttribute(  
...> "SupportedCurves")

The list should contain at least the following curves

secp192r1 (NIST P-192)  
secp224r1 (NIST P-224)  
secp256r1 (NIST P-256)  
secp384r1 (NIST P-384)  
secp521r1 (NIST P-521)

These are curves that have been standardized by NIST in FIPS 186-4 The prefix “sec” stands for “Standards for Efficient Cryptography”, the letter “p” indicates that this curve is over a prime field, the number following the “p” denotes the key size and finally, the letter “r” indicates that the parameters of the curve were chosen verifiably at random.

**Create a Public and Private Key**

import java.security.\*;  
import java.security.spec.\*;

KeyPairGenerator g = KeyPairGenerator.getInstance("EC","SunEC");  
ECGenParameterSpec ecsp = new ECGenParameterSpec("secp224r1");  
g.initialize(ecsp);

KeyPair kp = g.genKeyPair();  
PrivateKey privKey = kp.getPrivate();  
PublicKey pubKey = kp.getPublic();

**Select the signature algorithm of your choice**

Signature s = Signature.getInstance("SHA256withECDH","SunEC");  
s.initSign(privKey);

Now we compute the signature of the message and apply other. we call the updatefunction of the Signatureinstance and provide the message as input. Then we call the sign method which computes and returns the signature.

byte[] msg = "Hello, World!".getBytes("UTF-8");  
byte[] sig;  
s.update(msg);  
sig = s.sign();

**Now, the signature is stored in the byte array “sig”.**

Signature sg = Signature.getInstance("SHA256withECDSA", "SunEC");  
sg.initVerify(pubKey);  
sg.update(msg);  
boolean validSignature = sg.verify(sig);

If the provided message is the same for which the signature was computed, the verify method will return true, otherwise false.

Now calculated Elliptic curve is input of the Diffie hellman.

import java.math.BigInteger;

import java.security.KeyFactory;

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.SecureRandom;

import javax.crypto.spec.DHParameterSpec;

import javax.crypto.spec.DHPublicKeySpec;

public class Main {

public final static int pValue = 47;

public final static int gValue = 71;

public final static int XaValue = 9;

public final static int XbValue = 14;

public static void main(String[] args) throws Exception {

BigInteger p = new BigInteger(Integer.toString(pValue));

BigInteger g = new BigInteger(Integer.toString(gValue));

BigInteger Xa = new BigInteger(Integer.toString(XaValue));

BigInteger Xb = new BigInteger(Integer.toString(XbValue));

int bitLength = 512; // 512 bits

SecureRandom rnd = new SecureRandom();

p = BigInteger.probablePrime(bitLength, rnd);

g = BigInteger.probablePrime(bitLength, rnd);

createSpecificKey(p, g);

}

public static void createSpecificKey(BigInteger p, BigInteger g) throws Exception {

KeyPairGenerator kpg = KeyPairGenerator.getInstance("DiffieHellman");

DHParameterSpec param = new DHParameterSpec(p, g);

kpg.initialize(param);

KeyPair kp = kpg.generateKeyPair();

KeyFactory kfactory = KeyFactory.getInstance("DiffieHellman");

DHPublicKeySpec kspec = (DHPublicKeySpec) kfactory.getKeySpec(kp.getPublic(),

DHPublicKeySpec.class);

}

}

**This code is also provide ECDH security you can change bit level according to your requirement.**

import sun.misc.BASE64Decoder;

import sun.misc.BASE64Encoder;

import javax.crypto.\*;

import javax.crypto.spec.SecretKeySpec;

import java.io.IOException;

import java.security.\*;

public class AESSecurityCap {

private PublicKey publickey;

KeyAgreement keyAgreement;

byte[] sharedsecret;

String ALGO = "AES";

AESSecurityCap() {

makeKeyExchangeParams();

}

private void makeKeyExchangeParams() {

KeyPairGenerator kpg = null;

try {

kpg = KeyPairGenerator.getInstance("EC");

kpg.initialize(128);

KeyPair kp = kpg.generateKeyPair();

publickey = kp.getPublic();

keyAgreement = KeyAgreement.getInstance("ECDH");

keyAgreement.init(kp.getPrivate());

} catch (NoSuchAlgorithmException | InvalidKeyException e) {

e.printStackTrace();

}

}

public void setReceiverPublicKey(PublicKey publickey) {

try {

keyAgreement.doPhase(publickey, true);

sharedsecret = keyAgreement.generateSecret();

} catch (InvalidKeyException e) {

e.printStackTrace();

}

}

public String encrypt(String msg) {

try {

Key key = generateKey();

Cipher c = Cipher.getInstance(ALGO);

c.init(Cipher.ENCRYPT\_MODE, key);

byte[] encVal = c.doFinal(msg.getBytes());

return new BASE64Encoder().encode(encVal);

} catch (BadPaddingException | InvalidKeyException | NoSuchPaddingException | IllegalBlockSizeException | NoSuchAlgorithmException e) {

e.printStackTrace();

}

return msg;

}

public String decrypt(String encryptedData) {

try {

Key key = generateKey();

Cipher c = Cipher.getInstance(ALGO);

c.init(Cipher.DECRYPT\_MODE, key);

byte[] decordedValue = new BASE64Decoder().decodeBuffer(encryptedData);

byte[] decValue = c.doFinal(decordedValue);

return new String(decValue);

} catch (BadPaddingException | InvalidKeyException | NoSuchPaddingException | IllegalBlockSizeException | NoSuchAlgorithmException | IOException e) {

e.printStackTrace();

}

return encryptedData;

}

public PublicKey getPublickey() {

return publickey;

}

protected Key generateKey() {

return new SecretKeySpec(sharedsecret, ALGO);

}

}