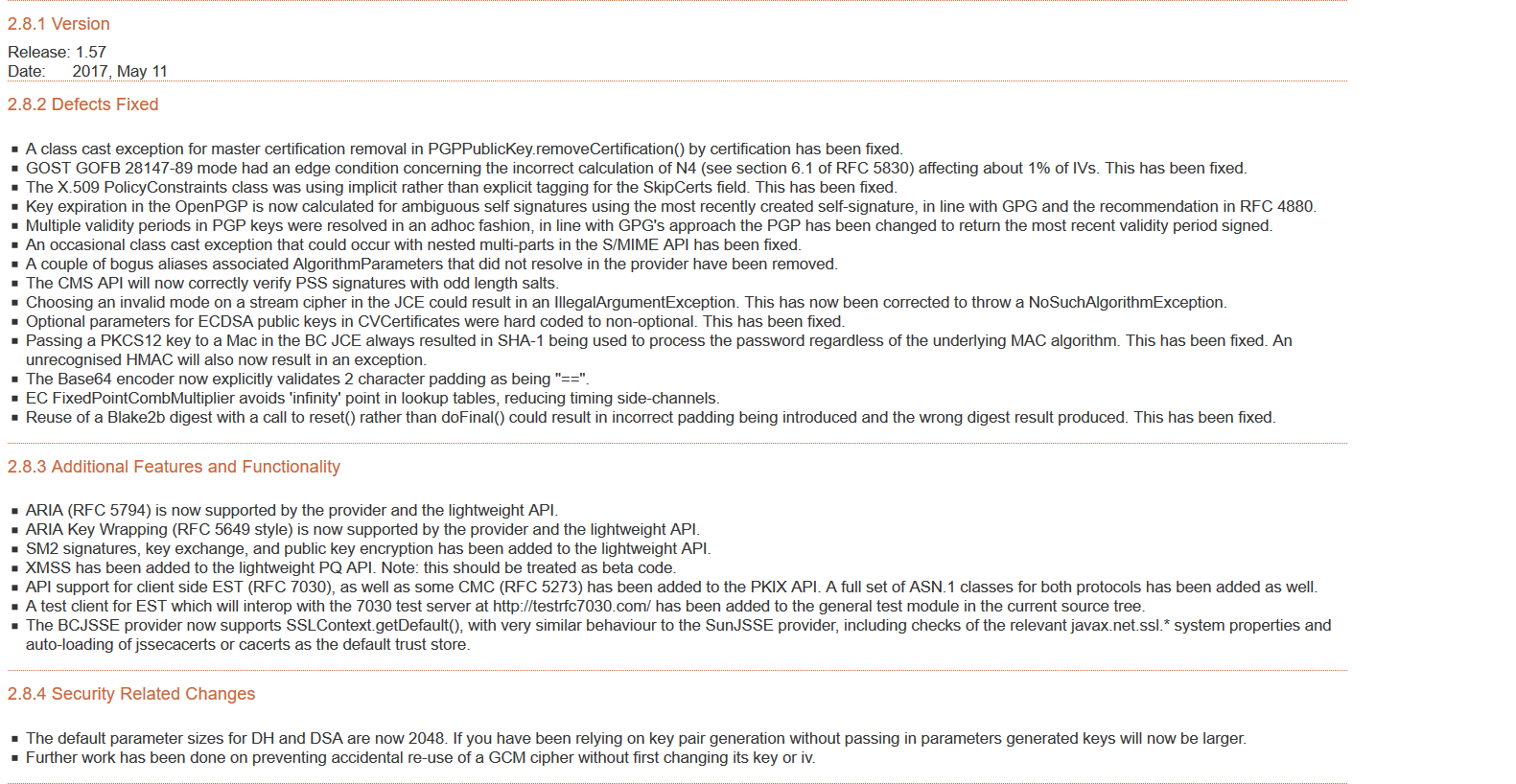
题目： java类库，使用Bouncy Castle的bcprov-jdkon15-164实现国密ecc算法的sm2加密签名规范。

1. 鄙人从16年中接触国密sm2算法以来，一路懵懵懂懂，跌跌撞撞，终于对非对称算法的领域应用略知一二。而由于java语言的本身性质，支持sm2算法同时又可以与其它平台互通的方式始终不得唯一。
2. 同时伴随sm2的应用在ECC算法中的地位逐渐攀升，2017年的1.57版bcprov库将SM2算法纳入到jce支持，虽然支持程度尚浅，但也是终于开始在世界范围内认可并推广使用。



1. 与sm2商用分治的另一领域，币圈，也使用了ECC的另一标准曲线“secp256r1”和“secp256k1”。由于在ECC曲线上找到一套适合应用于加密签名算法的标准参数费时费力，而其破解难度也不可估量，所以ecc的应用也逐渐成为非对称算法应用领域中的主流。
2. 本人在2017年使用bcprov库的Ecure及相关类，实现标准SM2的签名和加密，在1.53至1.59版都横行无阻。直到1.60版的出台，其大大删改了ECure的实现，尤其是取点的验证方面，使得鄙人不得不重新学习bcprov的sm2的实现。
3. 经过为期数天，终于在1.64版中固定了当前的sm2的实现方式，代码如下。

依赖库 mvn pom

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk15on --> <dependency>  <groupId>org.bouncycastle</groupId>  <artifactId>bcprov-jdk15on</artifactId>  <version>1.64</version> </dependency> |

实现代码：

|  |
| --- |
| package com.tes;  import org.bouncycastle.asn1.ASN1EncodableVector; import org.bouncycastle.asn1.ASN1Integer; import org.bouncycastle.asn1.ASN1Sequence; import org.bouncycastle.asn1.DERSequence; import org.bouncycastle.asn1.gm.GMNamedCurves; import org.bouncycastle.asn1.x9.X9ECParameters; import org.bouncycastle.crypto.InvalidCipherTextException; import org.bouncycastle.crypto.digests.SM3Digest; import org.bouncycastle.crypto.engines.SM2Engine; import org.bouncycastle.crypto.params.ECDomainParameters; import org.bouncycastle.crypto.params.ECPrivateKeyParameters; import org.bouncycastle.crypto.params.ECPublicKeyParameters; import org.bouncycastle.crypto.params.ParametersWithRandom; import org.bouncycastle.jcajce.provider.asymmetric.ec.BCECPrivateKey; import org.bouncycastle.jcajce.provider.asymmetric.ec.BCECPublicKey; import org.bouncycastle.jcajce.spec.SM2ParameterSpec; import org.bouncycastle.jce.provider.BouncyCastleProvider; import org.bouncycastle.jce.spec.ECParameterSpec; import org.bouncycastle.jce.spec.ECPrivateKeySpec; import org.bouncycastle.jce.spec.ECPublicKeySpec; import org.bouncycastle.util.encoders.Hex;  import javax.crypto.Cipher; import javax.crypto.spec.SecretKeySpec; import java.io.\*; import java.math.BigInteger; import java.security.\*; import java.security.cert.CertificateFactory; import java.security.cert.X509Certificate; import java.util.Arrays; import java.util.Base64;  */\*\*  \* 本类依赖包  \* bcpkix-jdk15on-164.jar  \* bcprov-jdk15on-164.jar  \*  \*  \* 用BC的注意点：  \* 这个版本的BC对SM3withSM2的结果为asn1格式的r和s，如果需要直接拼接的r||s需要自己转换。下面rsAsn1ToPlainByteArray、rsPlainByteArrayToAsn1就在干这事。  \* 这个版本的BC对SM2的结果为C1||C2||C3还是为C1||C3||C2，使用mode的枚举项进行初始化SM2Engine。  \*/* public class GmSm2Util {   private static X9ECParameters *x9ECParameters* = GMNamedCurves.*getByName*("sm2p256v1");  private static ECDomainParameters *ecDomainParameters* = new ECDomainParameters(*x9ECParameters*.getCurve(), *x9ECParameters*.getG(), *x9ECParameters*.getN());  private static ECParameterSpec *ecParameterSpec* = new ECParameterSpec(*x9ECParameters*.getCurve(), *x9ECParameters*.getG(), *x9ECParameters*.getN());   */\*\*GM SM2标准预处理ID\*/* private static byte[] *ID* = {0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38};   static {  if (Security.*getProvider*("BC") == null) {  Security.*addProvider*(new BouncyCastleProvider());  }  }   */\*\*  \*  \** ***@param*** *msg  \** ***@param*** *userId  \** ***@param*** *privateKey  \** ***@return*** *r||s，直接拼接byte数组的rs  \*/* public static byte[] signSm3WithSm2(byte[] msg, byte[] userId, PrivateKey privateKey){  return *rsAsn1ToPlainByteArray*(*signSm3WithSm2Asn1Rs*(msg, userId, privateKey));  }   */\*\*  \*  \** ***@param*** *msg  \** ***@param*** *userId  \** ***@param*** *privateKey  \** ***@return*** *rs in <b>asn1 format</b>  \*/* public static byte[] signSm3WithSm2Asn1Rs(byte[] msg, byte[] userId, PrivateKey privateKey){  if(null == userId) userId = *ID*;  try {  SM2ParameterSpec parameterSpec = new SM2ParameterSpec(userId);  Signature signer = Signature.*getInstance*("SM3withSM2", "BC");  signer.setParameter(parameterSpec);  signer.initSign(privateKey, new SecureRandom());  signer.update(msg, 0, msg.length);  byte[] sig = signer.sign();  return sig;  } catch (Exception e) {  throw new RuntimeException(e);  }  }   */\*\*  \*  \** ***@param*** *msg  \** ***@param*** *userId  \** ***@param*** *rs r||s，直接拼接byte数组的rs  \** ***@param*** *publicKey  \** ***@return*** *\*/* public static boolean verifySm3WithSm2(byte[] msg, byte[] userId, byte[] rs, PublicKey publicKey){  return *verifySm3WithSm2Asn1Rs*(msg, userId, *rsPlainByteArrayToAsn1*(rs), publicKey);  }   */\*\*  \*  \** ***@param*** *msg  \** ***@param*** *userId  \** ***@param*** *rs in <b>asn1 format</b>  \** ***@param*** *publicKey  \** ***@return*** *\*/* public static boolean verifySm3WithSm2Asn1Rs(byte[] msg, byte[] userId, byte[] rs, PublicKey publicKey){  if(null == userId) userId = *ID*;  try {  SM2ParameterSpec parameterSpec = new SM2ParameterSpec(userId);  Signature verifier = Signature.*getInstance*("SM3withSM2", "BC");  verifier.setParameter(parameterSpec);  verifier.initVerify(publicKey);  verifier.update(msg, 0, msg.length);  return verifier.verify(rs);  } catch (Exception e) {  throw new RuntimeException(e);  }  }      */\*\*  \* c1||c3||c2  \** ***@param*** *data  \** ***@param*** *key  \** ***@return*** *\*/* public static byte[] sm2Encrypt(byte[] data, PublicKey key){  BCECPublicKey localECPublicKey = (BCECPublicKey) key;  ECPublicKeyParameters ecPublicKeyParameters = new ECPublicKeyParameters(localECPublicKey.getQ(), *ecDomainParameters*);  SM2Engine sm2Engine = new SM2Engine(SM2Engine.Mode.*C1C3C2*);  sm2Engine.init(true, new ParametersWithRandom(ecPublicKeyParameters, new SecureRandom()));  try {  return sm2Engine.processBlock(data, 0, data.length);  } catch (InvalidCipherTextException e) {  throw new RuntimeException(e);  }  }   */\*\*  \* c1||c3||c2  \** ***@param*** *data  \** ***@param*** *key  \** ***@return*** *\*/* public static byte[] sm2Decrypt(byte[] data, PrivateKey key){  BCECPrivateKey localECPrivateKey = (BCECPrivateKey) key;  ECPrivateKeyParameters ecPrivateKeyParameters = new ECPrivateKeyParameters(localECPrivateKey.getD(), *ecDomainParameters*);  SM2Engine sm2Engine = new SM2Engine(SM2Engine.Mode.*C1C3C2*);  sm2Engine.init(false, ecPrivateKeyParameters);  try {  return sm2Engine.processBlock(data, 0, data.length);  } catch (InvalidCipherTextException e) {  throw new RuntimeException(e);  }  }     // 附赠sm4ECB算法支持   public static byte[] sm4Encrypt(byte[] keyBytes, byte[] plain){  if(keyBytes.length != 16) throw new RuntimeException("err key length");  if(plain.length % 16 != 0) throw new RuntimeException("err data length");   try {  Key key = new SecretKeySpec(keyBytes, "SM4");  Cipher out = Cipher.*getInstance*("SM4/ECB/NoPadding", "BC");  out.init(Cipher.*ENCRYPT\_MODE*, key);  return out.doFinal(plain);  } catch (Exception e) {  throw new RuntimeException(e);  }  }   public static byte[] sm4Decrypt(byte[] keyBytes, byte[] cipher){  if(keyBytes.length != 16) throw new RuntimeException("err key length");  if(cipher.length % 16 != 0) throw new RuntimeException("err data length");   try {  Key key = new SecretKeySpec(keyBytes, "SM4");  Cipher in = Cipher.*getInstance*("SM4/ECB/NoPadding", "BC");  in.init(Cipher.*DECRYPT\_MODE*, key);  return in.doFinal(cipher);   } catch (Exception e) {  throw new RuntimeException(e);  }   }   */\*\*  \** ***@param*** *bytes  \** ***@return*** *\*/* public static byte[] sm3(byte[] bytes) {  SM3Digest sm3 = new SM3Digest();  sm3.update(bytes, 0, bytes.length);  byte[] result = new byte[sm3.getDigestSize()];  sm3.doFinal(result, 0);  return result;  }   private final static int *RS\_LEN* = 32;   private static byte[] bigIntToFixexLengthBytes(BigInteger rOrS){  // for sm2p256v1, n is 00fffffffeffffffffffffffffffffffff7203df6b21c6052b53bbf40939d54123,  // r and s are the result of mod n, so they should be less than n and have length<=32  byte[] rs = rOrS.toByteArray();  if(rs.length == *RS\_LEN*) return rs;  else if(rs.length == *RS\_LEN* + 1 && rs[0] == 0) return Arrays.*copyOfRange*(rs, 1, *RS\_LEN* + 1);  else if(rs.length < *RS\_LEN*) {  byte[] result = new byte[*RS\_LEN*];  Arrays.*fill*(result, (byte)0);  System.*arraycopy*(rs, 0, result, *RS\_LEN* - rs.length, rs.length);  return result;  } else {  throw new RuntimeException("err rs: " + Hex.*toHexString*(rs));  }  }   */\*\*  \* BC的SM3withSM2签名得到的结果的rs是asn1格式的，这个方法转化成直接拼接r||s  \** ***@param*** *rsDer rs in asn1 format  \** ***@return*** *sign result in plain byte array  \*/* private static byte[] rsAsn1ToPlainByteArray(byte[] rsDer){  ASN1Sequence seq = ASN1Sequence.*getInstance*(rsDer);  byte[] r = *bigIntToFixexLengthBytes*(ASN1Integer.*getInstance*(seq.getObjectAt(0)).getValue());  byte[] s = *bigIntToFixexLengthBytes*(ASN1Integer.*getInstance*(seq.getObjectAt(1)).getValue());  byte[] result = new byte[*RS\_LEN* \* 2];  System.*arraycopy*(r, 0, result, 0, r.length);  System.*arraycopy*(s, 0, result, *RS\_LEN*, s.length);  return result;  }   */\*\*  \* BC的SM3withSM2验签需要的rs是asn1格式的，这个方法将直接拼接r||s的字节数组转化成asn1格式  \** ***@param*** *sign in plain byte array  \** ***@return*** *rs result in asn1 format  \*/* private static byte[] rsPlainByteArrayToAsn1(byte[] sign){  if(sign.length != *RS\_LEN* \* 2) throw new RuntimeException("err rs. ");  BigInteger r = new BigInteger(1, Arrays.*copyOfRange*(sign, 0, *RS\_LEN*));  BigInteger s = new BigInteger(1, Arrays.*copyOfRange*(sign, *RS\_LEN*, *RS\_LEN* \* 2));  ASN1EncodableVector v = new ASN1EncodableVector();  v.add(new ASN1Integer(r));  v.add(new ASN1Integer(s));  try {  return new DERSequence(v).getEncoded("DER");  } catch (IOException e) {  throw new RuntimeException(e);  }  }   public static KeyPair generateKeyPair(){  try {  KeyPairGenerator kpGen = KeyPairGenerator.*getInstance*("EC", "BC");  kpGen.initialize(*ecParameterSpec*, new SecureRandom());  KeyPair kp = kpGen.generateKeyPair();  return kp;  } catch (Exception e) {  throw new RuntimeException(e);  }  }   public static BCECPrivateKey getPrivatekeyFromD(BigInteger d){  ECPrivateKeySpec ecPrivateKeySpec = new ECPrivateKeySpec(d, *ecParameterSpec*);  return new BCECPrivateKey("EC", ecPrivateKeySpec, BouncyCastleProvider.*CONFIGURATION*);  }   public static BCECPublicKey getPublickeyFromXY(BigInteger x, BigInteger y){  ECPublicKeySpec ecPublicKeySpec = new ECPublicKeySpec(*x9ECParameters*.getCurve().createPoint(x, y), *ecParameterSpec*);  return new BCECPublicKey("EC", ecPublicKeySpec, BouncyCastleProvider.*CONFIGURATION*);  }   public static PublicKey getPublickeyFromX509File(File file){  try {  CertificateFactory cf = CertificateFactory.*getInstance*("X.509", "BC");  FileInputStream in = new FileInputStream(file);  X509Certificate x509 = (X509Certificate) cf.generateCertificate(in);  return x509.getPublicKey();  } catch (Exception e) {  throw new RuntimeException(e);  }  }   static void write(byte[] b, String f) throws Exception {  File file = null;  if (!(file = new File(f)).isFile()) {  file.createNewFile();  }  OutputStream out = new FileOutputStream(file);  out.write(b, 0, b.length);  out.flush();  out.close();  }     public static void main(String[] args) throws Exception {  // // 查看EC (elliptic cure)曲线的标准参数 --------------------- // System.out.println("GMNamedCurves: "); // for(Enumeration e = GMNamedCurves.getNames(); e.hasMoreElements();) { // System.out.println(e.nextElement()); // } // System.out.println("sm2p256v1 n:"+x9ECParameters.getN()); // System.out.println("sm2p256v1 nHex:"+Hex.toHexString(x9ECParameters.getN().toByteArray()));    // 生成公私钥对 ---------------------  KeyPair kp = *generateKeyPair*();   System.*out*.println(Hex.*toHexString*(kp.getPrivate().getEncoded()));  System.*out*.println(Hex.*toHexString*(kp.getPublic().getEncoded()));   System.*out*.println(kp.getPrivate().getAlgorithm());  System.*out*.println(kp.getPublic().getAlgorithm());   System.*out*.println(kp.getPrivate().getFormat());  System.*out*.println(kp.getPublic().getFormat());   System.*out*.println("private key d: " + ((BCECPrivateKey)kp.getPrivate()).getD());  System.*out*.println("public key q:" + ((BCECPublicKey)kp.getPublic()).getQ()); //{x, y, zs...}   System.*out*.println("==================");  System.*out*.printf("%s\n%s\n",  Base64.*getEncoder*().encodeToString(kp.getPrivate().getEncoded()),  Base64.*getEncoder*().encodeToString(kp.getPublic().getEncoded()));   *write*(((BCECPrivateKey) kp.getPrivate()).getD().toByteArray(), "C:\\Users\\49762\\Desktop\\sk.bin");  *write*( kp.getPublic().getEncoded(), "C:\\Users\\49762\\Desktop\\pk.bin");  System.*out*.println("------------------");   byte[] msg = "message digest".getBytes();  byte[] userId = "1234567812345678".getBytes();  byte[] sig = *signSm3WithSm2*(msg, userId, kp.getPrivate());  System.*out*.println(Hex.*toHexString*(sig));  System.*out*.println(*verifySm3WithSm2*(msg, userId, sig, kp.getPublic()));  *write*( sig, "C:\\Users\\49762\\Desktop\\sv.bin");  System.*out*.println("sig.len = " + sig.length);   // // 由d生成私钥 --------------------- // BigInteger d = new BigInteger("097b5230ef27c7df0fa768289d13ad4e8a96266f0fcb8de40d5942af4293a54a", 16); // BCECPrivateKey bcecPrivateKey = getPrivatekeyFromD(d); //// System.out.println(bcecPrivateKey.getParameters()); //// System.out.println(Hex.toHexString(bcecPrivateKey.getEncoded())); //// System.out.println(bcecPrivateKey.getAlgorithm()); //// System.out.println(bcecPrivateKey.getFormat()); //// System.out.println(bcecPrivateKey.getD()); //// System.out.println(bcecPrivateKey instanceof java.security.interfaces.ECPrivateKey); //// System.out.println(bcecPrivateKey instanceof ECPrivateKey); //// System.out.println(bcecPrivateKey.getParameters()); ////  //// 公钥X坐标PublicKeyXHex: 59cf9940ea0809a97b1cbffbb3e9d96d0fe842c1335418280bfc51dd4e08a5d4 //// 公钥Y坐标PublicKeyYHex: 9a7f77c578644050e09a9adc4245d1e6eba97554bc8ffd4fe15a78f37f891ff8 // PublicKey publicKey = getPublickeyFromX509File(new File("/Users/xxx/Downloads/xxxxx.cer")); // System.out.println(publicKey); //// PublicKey publicKey1 = getPublickeyFromXY(new BigInteger("59cf9940ea0809a97b1cbffbb3e9d96d0fe842c1335418280bfc51dd4e08a5d4", 16), new BigInteger("9a7f77c578644050e09a9adc4245d1e6eba97554bc8ffd4fe15a78f37f891ff8", 16)); //// System.out.println(publicKey1); //// System.out.println(publicKey.equals(publicKey1)); //// System.out.println(publicKey.getEncoded().equals(publicKey1.getEncoded())); //   // // sm2 encrypt and decrypt test --------------------- // KeyPair kp = generateKeyPair(); // PublicKey publicKey2 = kp.getPublic(); // PrivateKey privateKey2 = kp.getPrivate(); // byte[]bs = sm2Encrypt("s".getBytes(), publicKey2); // System.out.println(Hex.toHexString(bs)); // bs = sm2Decrypt(bs, privateKey2); // System.out.println(new String(bs));    // // sm4 encrypt and decrypt test --------------------- // //0123456789abcdeffedcba9876543210 + 0123456789abcdeffedcba9876543210 -> 681edf34d206965e86b3e94f536e4246 // byte[] plain = Hex.decode("0123456789abcdeffedcba98765432100123456789abcdeffedcba98765432100123456789abcdeffedcba9876543210"); // byte[] key = Hex.decode("0123456789abcdeffedcba9876543210"); // byte[] cipher = Hex.decode("595298c7c6fd271f0402f804c33d3f66"); // byte[] bs = sm4Encrypt(key, plain); // System.out.println(Hex.toHexString(bs));; // bs = sm4Decrypt(key, bs); // System.out.println(Hex.toHexString(bs));  } } |