# Homework 3

Computer Security

**Illa Nuka Saranya**

**50248926**

**04-OCT-18**

**Saranya@buffalo.edu**

**1)**For each encryption algorithm, the following table shows the measure of

(i) the time it take to generate a new key,

(ii) the time it takes to encrypt and

(iii) the time it takes to decrypt each of the two files. Compute encryption and decryption speeds per byte for both of the files.

(iv) Similarly, for the signature scheme, measure the key generation time and the time to produce and verify a signature for the two files (the total time and per-byte time).

(v) Finally, for the hash functions, measure the total time to compute the hash of both files and compute per-byte timings.

Table answering (i),(ii),(iii)

| Operation / Algorithm | Key Generation | One KB | | | | One MB | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Encryption (in ms) | Decryption (in ms) | Per Byte Speed (in ns) | | Encryption (in ms) | Decryption (in ms) | Per Byte Speed (in ns) | |
| | | | | Enc | Dec | | | Enc | Dec |
| AES_CBC_128 | 224.5 ms | 0.72 ms | 0.22 ms | 7998 | 2190 | 28.1 ms | 26.7 ms | 27 | 28 |
| AES_CTR_128 | 556.2 ms | 2.8 ms | 0.2 ms | 29567 | 2240 | 29.5 ms | 12.4 ms | 29 | 13 |
| AES_CTR_256 | 556.5 ms | 0.29 ms | 0.47 ms | 3457 | 4154 | 30.4 ms | 16.1 ms | 29 | 14 |

Table answering (iv)

| Operation on file / Algorithm | Key Generation (in ms) | One KB | | One MB | | One KB | | One MB | |
|---|---|---|---|---|---|---|---|---|---|
| | | Signing the File (in ms) | PBS (in ns) | Signing the File (in ms) | PBS (in ns) | Verifying the File (in ms) | PBS (in ns) | Verifying the File (in ms) | PBS (in ns) |
| RSA_2048 | 479.9 | 943.97 | 920940 | 222.2 | 212 | 8.43 | 8285 | 138.77 | 132 |
| RSA_3072 | 520.9 | 873.97 | 76985 | 189.2 | 190 | 9.43 | 8985 | 208.77 | 126 |
| DSA_2048 | 184.9 | 15.74 | 110832 | 48.4 | 50 | 9.8 | 61230 | 20.2 | 20 |
| DSA_3072 | 204.8 | 17 | 245805 | 46.37 | 42 | 34.7 | 141269 | 18.75 | 25 |

Table answering (v)

| Hashing / File Size | One KB | | One MB | |
|---|---|---|---|---|
| | Total Hashing time | Per Byte Speed (in ns) | Total Hashing time | Per Byte Speed (in ns) |
| Sha-256 | 12.5 ms | 147961 | 33.7 ms | 32 |
| Sha-512 | 0.46 ms | 4724 | 25.5 ms | 26 |
| Sha3-256 | 8.1 ms | 87573 | 22.5 ms | 21 |

**ii)** Providing the source code

**a)Creating a 128-bit AES key, encrypt and decrypt each of the two files using AES in the CBC mode:**

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.SecureRandom;
import java.util.concurrent.TimeUnit;

public class AES128CBC
 {
   public static void main(String[] args) throws Exception
         {
              long startTime = System.nanoTime();
              KeyGenerator keyGen = KeyGenerator.getInstance("AES");
              keyGen.init(128);
              SecretKey secretKey = keyGen.generateKey();
              long estimatedTime = System.nanoTime() - startTime;
              System.out.println(estimatedTime/100000);
              System.out.println("Time taken for key generation in ns: "+ estimatedTime );
              System.out.println("One MB");
              encrypt( secretKey,new File("oneMB.txt"),new File("oneMBcbc.encrypted"));
             decrypt(secretKey,new File("oneMBcbc.encrypted"),new File("decrypted-      oneMBcbc.txt"));
              System.out.println("One KB");
             encrypt( secretKey,new File("oneKB.txt"),new   File("oneKBcbc.encrypted"));
            decrypt(secretKey,new       File("oneKBcbc.encrypted"),new File("decrypted- oneKBcbc.txt"));
       }

public static void encrypt(SecretKey key,File inputFile,File outputFile)   throws      Exception
 {
      FileInputStream inputStream = new FileInputStream(inputFile);
     byte[] clean = new byte[(int) inputFile.length()];
     inputStream.read(clean);
     long startTime = System.nanoTime();
      byte[] iv = new byte[16];
      SecureRandom random = new SecureRandom();
      random.nextBytes(iv);
      IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
      cipher.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
      byte[] encrypted = cipher.doFinal(clean);
      byte[] encryptedIVAndText = new byte[16 + encrypted.length];
      System.arraycopy(iv, 0, encryptedIVAndText, 0, 16);
      System.arraycopy(encrypted, 0, encryptedIVAndText,16, encrypted.length);
```

```
        long estimatedTime= System.nanoTime()-startTime;
        System.out.println("Encryption time in ns: " + estimatedTime);
        System.out.println("Per Byte Speed: "+ estimatedTime/clean.length+);

        FileOutputStream outputStream = new FileOutputStream(outputFile);
            outputStream.write(encryptedIVAndText);
            inputStream.close();
            outputStream.close();

    }
    public static void decrypt( SecretKey key,File encryptedFile,File decryptedFile) throws Exception {

        byte[] iv = new byte[16];
        FileInputStream inputStream = new FileInputStream(encryptedFile);
        byte[] clean = new byte[(int) encryptedFile.length()];
        inputStream.read(clean);
        long startTime= System.nanoTime();
        System.arraycopy(clean, 0, iv, 0, iv.length);
        IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
        int encryptedSize = clean.length - 16;
        byte[] encryptedBytes = new byte[encryptedSize];
        System.arraycopy(clean, 16, encryptedBytes, 0, encryptedSize);
        Cipher cipherDecrypt = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipherDecrypt.init(Cipher.DECRYPT_MODE, key, ivParameterSpec);
        byte[] decrypted = cipherDecrypt.doFinal(encryptedBytes);
        long estimatedTime= System.nanoTime()-startTime;
        System.out.println("Decryption time in ns: " + estimatedTime);
        FileOutputStream outputStream = new FileOutputStream(decryptedFile);
            outputStream.write(decrypted);
            inputStream.close();
            outputStream.close();

    }
}
```

**Output**:

```
Time taken for key generation in ns: 227243385
One MB
Encryption time in ns: 28769744
Per Byte Speed of Encryption: 27
Decryption time in ns: 27633102
Per Byte Speed of Decryption: 26
One KB
Encryption time in ns: 897707
Per Byte Speed of Encryption: 9449
Decryption time in ns: 270507
Per Byte Speed of Decryption: 2415
```

**b)Creating a 128-bit AES key, encrypt and decrypt each of the two files using AES in the CTR mode**

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.SecureRandom;

public class AES128CTR {
   public static void main(String[] args) throws Exception
   {
     long startTime = System.nanoTime();
      KeyGenerator keyGen = KeyGenerator.getInstance("AES");
      keyGen.init(128);
      SecretKey secretKey = keyGen.generateKey();
      long estimatedTime = System.nanoTime() - startTime;
      System.out.println("Time taken for key generation:  "+ estimatedTime );
      System.out.println("One MB");
      encrypt( secretKey,new File("oneMB.txt"),new File("oneMBctr.encrypted"));
      decrypt(secretKey,new File("oneMBctr.encrypted"),new File("decrypted- oneMBctr.txt"));
      System.out.println("One KB");
      encrypt( secretKey,new File("oneKB.txt"),new File("oneKBcbc.encrypted"));
      decrypt(secretKey,new File("oneKBctr.encrypted"),new File("decrypted-oneKBctr.txt"));

   }

      public static void encrypt(SecretKey key,File inputFile,File outputFile) throws Exception
    {
                FileInputStream inputStream = new FileInputStream(inputFile);
                byte[] clean = new byte[(int) inputFile.length()];
                inputStream.read(clean);
                int ivSize = 16;
                byte[] iv = new byte[ivSize];
                long startTime = System.nanoTime();
                SecureRandom random = new SecureRandom();
                random.nextBytes(iv);
                IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
                Cipher cipher = Cipher.getInstance("AES/CTR/PKCS5Padding");
                cipher.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);
                byte[] encrypted = cipher.doFinal(clean);
                byte[] encryptedIVAndText = new byte[ivSize + encrypted.length];
                System.arraycopy(iv, 0, encryptedIVAndText, 0, ivSize);
                System.arraycopy(encrypted, 0, encryptedIVAndText, ivSize, encrypted.length);
                long estimatedTime = System.nanoTime()- startTime;
                System.out.println("Time taken for Encryption in ns: "+ estimatedTime);
```

```
            FileOutputStream outputStream = new FileOutputStream(outputFile);
                outputStream.write(encryptedIVAndText);
                inputStream.close();
                outputStream.close();
    }

     public static void decrypt( SecretKey key,File encryptedFile,File decryptedFile) throws Exception
    {
      int ivSize = 16;
      byte[] iv = new byte[ivSize];
      FileInputStream inputStream = new FileInputStream(encryptedFile);
      byte[] clean = new byte[(int) encryptedFile.length()];
      inputStream.read(clean);
      long startTime = System.nanoTime();
      System.arraycopy(clean, 0, iv, 0, iv.length);
      IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
      int encryptedSize = clean.length - ivSize;
      byte[] encryptedBytes = new byte[encryptedSize];
      System.arraycopy(clean, ivSize, encryptedBytes, 0, encryptedSize);
      Cipher cipherDecrypt = Cipher.getInstance("AES/CTR/PKCS5Padding");
      cipherDecrypt.init(Cipher.DECRYPT_MODE, key, ivParameterSpec);
      byte[] decrypted = cipherDecrypt.doFinal(encryptedBytes);
      long estimatedTime = System.nanoTime()-startTime;
      System.out.println("Time taken for Decryption in ns "+ estimatedTime);
      FileOutputStream outputStream = new FileOutputStream(decryptedFile);
      outputStream.write(decrypted);
      inputStream.close();
      outputStream.close();
     }
}
```

**Output**:

```
Time taken for key generation:  555642738
One MB
Time taken for Encryption in ns: 32784682
Per Byte Speed: 29
Time taken for Decryption in ns 13856018
Per Byte Speed: 13
One KB
Time taken for Encryption in ns: 2992644
Per Byte Speed: 31501
Time taken for Decryption in ns 240640
Per Byte Speed: 2167
```

**c)Create a 256-bit AES key, encrypt and decrypt each of the two files using AES in the CTR mode**

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.SecureRandom;

public class AES256CTR
 {
   public static void main(String[] args) throws Exception
   {
     long startTime = System.nanoTime();
      KeyGenerator keyGen = KeyGenerator.getInstance("AES");
      keyGen.init(256);
      SecretKey secretKey = keyGen.generateKey();
      long estimatedTime = System.nanoTime() - startTime;
      System.out.println("Time taken for key generation in ns: " + estimatedTime);
      System.out.println("One MB");
      encrypt( secretKey,new File("oneMB.txt"),new File("oneMBcbc.encrypted"));
      decrypt(secretKey,new File("oneMBcbc.encrypted"),new File("decrypted-oneMBcbc.txt"));
      System.out.println("One KB");
      encrypt( secretKey,new File("oneKB.txt"),new File("oneKBcbc.encrypted"));
      decrypt(secretKey,new File("oneKBcbc.encrypted"),new File("decrypted-oneKBcbc.txt"));

   }

   public static void encrypt(SecretKey key,File inputFile,File outputFile)
                   throws Exception
   {

      FileInputStream inputStream = new FileInputStream(inputFile);
      byte[] clean = new byte[(int) inputFile.length()];
      inputStream.read(clean);
      int ivSize = 16;
      long st= System.nanoTime();
      byte[] iv = new byte[ivSize];
      SecureRandom random = new SecureRandom();
      random.nextBytes(iv);
      IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
      byte[] keyBytes = new byte[16];
      System.arraycopy(key.getEncoded(), 0, keyBytes, 0, keyBytes.length);
      SecretKeySpec secretKeySpec = new SecretKeySpec(keyBytes, "AES");
      Cipher cipher = Cipher.getInstance("AES/CTR/PKCS5Padding");
```

```
        cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivParameterSpec);
        byte[] encrypted = cipher.doFinal(clean);
        byte[] encryptedIVAndText = new byte[ivSize + encrypted.length];
        System.arraycopy(iv, 0, encryptedIVAndText, 0, ivSize);
        System.arraycopy(encrypted, 0, encryptedIVAndText, ivSize, encrypted.length);
        long et= System.nanoTime()-st;
        System.out.println("Encryption time in ns: "+ et );
        FileOutputStream outputStream = new FileOutputStream(outputFile);
       outputStream.write(encryptedIVAndText);
        inputStream.close();
       outputStream.close();
}

   public static void decrypt( SecretKey key,File encryptedFile,File decryptedFile)
                        throws Exception {
       int ivSize = 16;
       int keySize = 16;
       long st= System.nanoTime();
       byte[] iv = new byte[ivSize];
       FileInputStream inputStream = new FileInputStream(encryptedFile);
       byte[] clean = new byte[(int) encryptedFile.length()];
       inputStream.read(clean);
       System.arraycopy(clean, 0, iv, 0, iv.length);
       IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
       int encryptedSize = clean.length - ivSize;
       byte[] encryptedBytes = new byte[encryptedSize];
       System.arraycopy(clean, ivSize, encryptedBytes, 0, encryptedSize);
       byte[] keyBytes = new byte[keySize];
       System.arraycopy(key.getEncoded(), 0, keyBytes, 0, keyBytes.length);
       SecretKeySpec secretKeySpec = new SecretKeySpec(keyBytes, "AES");
       Cipher cipherDecrypt = Cipher.getInstance("AES/CTR/PKCS5Padding");
       cipherDecrypt.init(Cipher.DECRYPT_MODE, secretKeySpec, ivParameterSpec);
       byte[] decrypted = cipherDecrypt.doFinal(encryptedBytes);
       long et= System.nanoTime()-st;
       System.out.println("Decryption time in ns: "+et);
       FileOutputStream outputStream = new FileOutputStream(decryptedFile);
       outputStream.write(decrypted);
       inputStream.close();
       outputStream.close();

   }
}
```

**Output:**

Time taken for key generation in ns: 593197985

One MB

Encryption time in ns: 35813593
Decryption time in ns: 16241087
Per Byte Speed: 15
One KB
Encryption time in ns: 299520
Decryption time in ns: 493654
Per Byte Speed: 4447

**d) Compute a hash of each of the files using hash functions SHA-256, SHA-512, and SHA3- 256**

**Computing hash of SHA-256 and SHA-512**

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.MessageDigest;

public class Hashing {
   public static void main(String[] args) throws Exception
 {
      System.out.println("One KB");
      hash_sha256(new File("oneKB.txt"),new File("OneKB_sha256.txt"));
      hash_sha512(new File("oneKB.txt"),new File("OneKB_sha512.txt"));
      System.out.println("One MB");
      hash_sha256(new File("oneMB.txt"),new File("OneMB_sha256.txt"));
      hash_sha512(new File("oneMB.txt"),new File("OneMB_sha512.txt"));
   }
   public static void hash_sha256(File inputFile,File outputFile) throws Exception
{

      FileInputStream inputStream = new FileInputStream(inputFile);
      byte[] clean = new byte[(int) inputFile.length()];
      inputStream.read(clean);
      long startTime = System.nanoTime();
      MessageDigest digest = MessageDigest.getInstance("SHA-256");
      digest.update(clean,0,clean.length);
      byte[] hash = digest.digest();
      long estimatedTime = System.nanoTime() - startTime;
      System.out.println("Time taken for SHA-256 Hashing in ns: "+ estimatedTime);
      FileOutputStream outputStream = new FileOutputStream(outputFile);
      outputStream.write(hash);
      inputStream.close();
     outputStream.close();
   }

   public static void hash_sha512(File inputFile,File outputFile) throws Exception
   {

      FileInputStream inputStream = new FileInputStream(inputFile);
      byte[] clean = new byte[(int) inputFile.length()];
```

```
      inputStream.read(clean);
      long startTime = System.nanoTime();
      MessageDigest digest = MessageDigest.getInstance("SHA-512");
       digest.update(clean,0,clean.length);
       byte[] hash = digest.digest();
       long estimatedTime = System.nanoTime() - startTime;
       System.out.println("Time taken for SHA-512 Hashing in ns: "+ estimatedTime);
       FileOutputStream outputStream = new FileOutputStream(outputFile);
      outputStream.write(hash);
       inputStream.close();
       outputStream.close();
            }
}
```

**Output**:

```
One KB
Time taken for SHA-256 Hashing in ns: 13355537
Per Byte Speed in ns: 140584
Time taken for SHA-512 Hashing in ns: 423254
Per Byte Speed in ns: 4455
One MB
Time taken for SHA-256 Hashing in ns: 41641866
Per Byte Speed in ns: 39
Time taken for SHA-512 Hashing in ns: 27709902
Per Byte Speed in ns: 26
```

**Computing hash of SHA3-256**

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.UnsupportedEncodingException;
import org.bouncycastle.jcajce.provider.digest.SHA3;
import org.bouncycastle.jcajce.provider.digest.SHA3.DigestSHA3;
import org.bouncycastle.util.encoders.Hex;

public class Hashing_Sha3_256 {
	public static void main(String[] args) throws UnsupportedEncodingException
	{
		try
		{
			System.out.println("One KB");
			hash_sha3_256(new File("oneKB.txt"),new File("OneKB_sha3_256.txt"));
			System.out.println("One MB");
			hash_sha3_256(new File("oneMB.txt"),new File("OneMB_sha3_256.txt"));
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}

	}
	public static void hash_sha3_256(File inputFile,File outputFile) throws Exception
	{
		FileInputStream inputStream = new FileInputStream(inputFile);
		byte[] clean = new byte[(int) inputFile.length()];
		inputStream.read(clean);
		long startTime = System.nanoTime();
		DigestSHA3 sha3256 = new SHA3.Digest256();
		 sha3256.update(clean);
		 byte[] hash= sha3256.digest();
		 long estimatedTime = System.nanoTime() - startTime;
		 System.out.println("Time taken for SHA3_256 in ns: "+estimatedTime);
		String hexString=Hex.toHexString(hash);
		FileOutputStream outputStream = new FileOutputStream(outputFile);
		 outputStream.write(("Hex String is:\n"+ hexString+" \n \nHash value

		 of the given file is:\n").getBytes());
		  outputStream.write(hash);
		  inputStream.close();
		   outputStream.close();
	 }

}
```

**Output**:

One KB
Time taken for SHA3_256 in ns: 8939958
Per Byte speed of hashing in ns: 94104
One MB
Time taken for SHA3_256 in ns: 19076292
Per Byte speed of hashing in ns: 18

**e)**

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Base64;
import java.util.Arrays;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchProviderException;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import org.bouncycastle.jcajce.provider.*;
public class sample1
{
   static private Base64.Encoder encoder = Base64.getEncoder();
   static SecureRandom srandom = new SecureRandom();
   static private void processFile(Cipher ci,InputStream in,OutputStream out)
           throws javax.crypto.IllegalBlockSizeException,
                javax.crypto.BadPaddingException,
                java.io.IOException
   {
           byte[] ibuf = new byte[86];
           int len;
           while ((len = in.read(ibuf)) != -1) {
              byte[] obuf = ci.update(ibuf, 0, len);
              if ( obuf != null ) out.write(obuf);
           }
           byte[] obuf = ci.doFinal();
           if ( obuf != null ) out.write(obuf);
   }

   static private void processFile(Cipher ci,String inFile,String outFile)
```

```
        throws javax.crypto.IllegalBlockSizeException,
            javax.crypto.BadPaddingException,
            java.io.IOException
{
        try (FileInputStream in = new FileInputStream(inFile);
            FileOutputStream out = new FileOutputStream(outFile)) {
                processFile(ci, in, out);
            }
}

static private void doGenkey(String args)
        throws java.security.NoSuchAlgorithmException,
            java.io.IOException
{
        String fileBase = args;
        long startTime=System.nanoTime();
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(2048);
        KeyPair kp = kpg.generateKeyPair();
        byte[] s= kp.getPrivate().getEncoded();
        byte[] p= kp.getPublic().getEncoded();
        long estimatedTime=System.nanoTime()-startTime;
        System.out.println("Key generation: " + estimatedTime);
        try (FileOutputStream out = new FileOutputStream(fileBase + ".key")) {
                out.write(s);
            }
        try (FileOutputStream out = new FileOutputStream(fileBase + ".pub")) {
                out.write(p);
            }
}

static private void doEncrypt(String pvtKeyFile,String inputFile)
        throws java.security.NoSuchAlgorithmException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            java.security.InvalidKeyException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pvtKeyFile));
        long startTime= System.nanoTime();
        PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PrivateKey pvt = kf.generatePrivate(ks);
        Cipher cipher;
        try {
```

```
                Security.addProvider(new   org.bouncycastle.jce.provider.BouncyCastleProvider());
                cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding", "BC");
                cipher.init(Cipher.ENCRYPT_MODE, pvt);
                long et= System.nanoTime()-startTime;
                System.out.println("Time taken for encryption: "+et);
                String file="encrypted_"+inputFile.substring(0,inputFile.indexOf('.'));
                processFile(cipher, inputFile, file + ".enc");
            }
         catch (NoSuchProviderException e) {
                e.printStackTrace();
        }

}

 static private void doDecrypt(String pubKeyFile,String encryptedFile)
        throws java.security.NoSuchAlgorithmException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            java.security.InvalidKeyException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pubKeyFile));
        long startTime= System.nanoTime();
        X509EncodedKeySpec ks = new X509EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PublicKey pub = kf.generatePublic(ks);
        Cipher cipher;
        try {
                Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding", "BC");
                cipher.init(Cipher.DECRYPT_MODE, pub);
                long estimatedTime= System.nanoTime()-startTime;
                System.out.println("Time taken for decryption: "+ estimatedTime);
    String file="decrypted_"+encryptedFile.substring(0,encryptedFile.indexOf('.'));
                processFile(cipher, encryptedFile, file + ".txt");
        } catch (NoSuchProviderException e)
        {
                                e.printStackTrace();
        }

}
 static private void doEncryptRSAWithAES(String pvtKeyFile, String inputFile)
        throws java.security.NoSuchAlgorithmException,
            java.security.InvalidAlgorithmParameterException,
            java.security.InvalidKeyException,
```

```
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pvtKeyFile));
         long startTime = System.nanoTime();
        PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PrivateKey pvt = kf.generatePrivate(ks);

        KeyGenerator kgen = KeyGenerator.getInstance("AES");
        kgen.init(128);
        SecretKey skey = kgen.generateKey();

        byte[] iv = new byte[128/8];
        srandom.nextBytes(iv);
        IvParameterSpec ivspec = new IvParameterSpec(iv);
        String file="encrypted_"+inputFile.substring(0,inputFile.indexOf('.'));
        try (FileOutputStream out = new FileOutputStream(file + ".enc")) {
                {
                        Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                        Cipher cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding",
                        "BC");
                        cipher.init(Cipher.ENCRYPT_MODE, pvt);
                        byte[] b = cipher.doFinal(skey.getEncoded());
                        out.write(b);
                 }

                out.write(iv);
                Cipher ci = Cipher.getInstance("AES/CBC/PKCS5Padding");
                ci.init(Cipher.ENCRYPT_MODE, skey, ivspec);
             long et = System.nanoTime()-startTime;
       System.out.println("Encryption using AES RSA: "+et);
                try (FileInputStream in = new FileInputStream(inputFile))
                 {
                        processFile(ci, in, out);
                   }
           } catch (NoSuchProviderException e) {

                        e.printStackTrace();
                }
}

   static private void doDecryptRSAWithAES(String pubKeyFile,String encryptedFile) throws
java.security.NoSuchAlgorithmException,java.security.InvalidAlgorithmParameterException,
```

```
            java.security.InvalidKeyException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
  {

        byte[] bytes = Files.readAllBytes(Paths.get(pubKeyFile));
         long startTime = System.nanoTime();
        X509EncodedKeySpec ks = new X509EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PublicKey pub = kf.generatePublic(ks);

        try (FileInputStream in = new FileInputStream(encryptedFile)) {
                SecretKeySpec skey = null;
                {
                        Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                        Cipher cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding",
"BC");
                   cipher.init(Cipher.DECRYPT_MODE, pub);
                   byte[] b = new byte[256];
                   in.read(b);
                   byte[] keyb = cipher.doFinal(b);
                   skey = new SecretKeySpec(keyb, "AES");
                }

                byte[] iv = new byte[128/8];
                in.read(iv);
                IvParameterSpec ivspec = new IvParameterSpec(iv);

          Cipher ci = Cipher.getInstance("AES/CBC/PKCS5Padding");
          ci.init(Cipher.DECRYPT_MODE, skey, ivspec);
          long et = System.nanoTime()-startTime;
     System.out.println("decryption using AES RSA: "+et);

        String file="decrypted_"+encryptedFile.substring(0,encryptedFile.indexOf('.'));
                try (FileOutputStream out = new FileOutputStream(file+".txt")){
                        processFile(ci, in, out);
                   }
            } catch (NoSuchProviderException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
  }

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

```
            String inputFileName1="oneKB.txt";
             String inputFileName2="oneMB.txt";
             String keyName="rsaalgorithm";
String file1="encrypted_"+inputFileName1.substring(0,inputFileName1.indexOf('.'));
String file2="encrypted_"+inputFileName2.substring(0,inputFileName2.indexOf('.'));
             sample1.doGenkey(keyName);
             sample1.doEncrypt("rsaalgorithm.key",inputFileName1);
             sample1.doDecrypt("rsaalgorithm.pub",file1+".enc");
             sample1.doEncryptRSAWithAES("rsaalgorithm.key",inputFileName2);
             sample1.doDecryptRSAWithAES("rsaalgorithm.pub",file2+".enc");

   }
}
```

**Output**:

```
Key generation: 608380939
Time taken for encryption: 963634300
Per Byte Speed:941049
Time taken for decryption: 7849397
Per Byte Speed:7665
Time taken for encryption: 205275569
Per Byte Speed:195
Time taken for decryption: 132766890
Per Byte Speed:126
```

**f) Create a 3072-bit RSA key, encrypt and decrypt the files above with PKCS #1 v2 padding (at least v2.0, but v2.2 is preferred; it may also be called OAEP).**

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Base64;
import java.util.Arrays;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchProviderException;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Security;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import org.bouncycastle.jcajce.provider.*;
public class sample1
{
   static private Base64.Encoder encoder = Base64.getEncoder();
   static SecureRandom srandom = new SecureRandom();
   static private void processFile(Cipher ci,InputStream in,OutputStream out)
         throws javax.crypto.IllegalBlockSizeException,
              javax.crypto.BadPaddingException,
              java.io.IOException
   {
         byte[] ibuf = new byte[1024];
         int len;
         while ((len = in.read(ibuf)) != -1) {
            byte[] obuf = ci.update(ibuf, 0, len);
            if ( obuf != null ) out.write(obuf);
         }
         byte[] obuf = ci.doFinal();
         if ( obuf != null ) out.write(obuf);
```

```
}

static private void processFile(Cipher ci,String inFile,String outFile)
        throws javax.crypto.IllegalBlockSizeException,
            javax.crypto.BadPaddingException,
            java.io.IOException
{
        try (FileInputStream in = new FileInputStream(inFile);
           FileOutputStream out = new FileOutputStream(outFile)) {
                processFile(ci, in, out);
          }
}

static private void doGenkey(String args)
        throws java.security.NoSuchAlgorithmException,
            java.io.IOException
{
        String fileBase = args;
        long startTime=System.nanoTime();
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
        kpg.initialize(3072);
        KeyPair kp = kpg.generateKeyPair();
        byte[] s= kp.getPrivate().getEncoded();
        byte[] p= kp.getPublic().getEncoded();
        long estimatedTime=System.nanoTime()-startTime;
        System.out.println("Key generation: " + estimatedTime);
        try (FileOutputStream out = new FileOutputStream(fileBase + ".key")) {
                out.write(s);
          }
        try (FileOutputStream out = new FileOutputStream(fileBase + ".pub")) {
                out.write(p);
          }
}

static private void doEncrypt(String pvtKeyFile,String inputFile)
        throws java.security.NoSuchAlgorithmException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            java.security.InvalidKeyException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pvtKeyFile));
        long startTime= System.nanoTime();
        PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
```

```
        PrivateKey pvt = kf.generatePrivate(ks);
        Cipher cipher;
        try {
                Security.addProvider(new   org.bouncycastle.jce.provider.BouncyCastleProvider());
                cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding", "BC");
                cipher.init(Cipher.ENCRYPT_MODE, pvt);
                long et= System.nanoTime()-startTime;
                System.out.println("Time taken for encryption: "+et);
                String file="encrypted_"+inputFile.substring(0,inputFile.indexOf('.'));
                processFile(cipher, inputFile, file + ".enc");
           }
         catch (NoSuchProviderException e) {
                e.printStackTrace();
        }

}

 static private void doDecrypt(String pubKeyFile,String encryptedFile)
        throws java.security.NoSuchAlgorithmException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            java.security.InvalidKeyException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pubKeyFile));
        long startTime= System.nanoTime();
        X509EncodedKeySpec ks = new X509EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PublicKey pub = kf.generatePublic(ks);
        Cipher cipher;
        try {
                Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding", "BC");
                cipher.init(Cipher.DECRYPT_MODE, pub);
                long estimatedTime= System.nanoTime()-startTime;
                System.out.println("Time taken for decryption: "+ estimatedTime);
    String file="decrypted_"+encryptedFile.substring(0,encryptedFile.indexOf('.'));
                processFile(cipher, encryptedFile, file + ".txt");
        } catch (NoSuchProviderException e)
        {
                                e.printStackTrace();
        }

 }
 static private void doEncryptRSAWithAES(String pvtKeyFile, String inputFile)
```

```
        throws java.security.NoSuchAlgorithmException,
            java.security.InvalidAlgorithmParameterException,
            java.security.InvalidKeyException,
            java.security.spec.InvalidKeySpecException,
            javax.crypto.NoSuchPaddingException,
            javax.crypto.BadPaddingException,
            javax.crypto.IllegalBlockSizeException,
            java.io.IOException
{

        byte[] bytes = Files.readAllBytes(Paths.get(pvtKeyFile));
         long startTime = System.nanoTime();
        PKCS8EncodedKeySpec ks = new PKCS8EncodedKeySpec(bytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PrivateKey pvt = kf.generatePrivate(ks);

        KeyGenerator kgen = KeyGenerator.getInstance("AES");
        kgen.init(128);
        SecretKey skey = kgen.generateKey();

        byte[] iv = new byte[128/8];
        srandom.nextBytes(iv);
        IvParameterSpec ivspec = new IvParameterSpec(iv);
        String file="encrypted_"+inputFile.substring(0,inputFile.indexOf('.'));
        try (FileOutputStream out = new FileOutputStream(file + ".enc")) {
                {
                        Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                        Cipher cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding",
                        "BC");
                        cipher.init(Cipher.ENCRYPT_MODE, pvt);
                        byte[] b = cipher.doFinal(skey.getEncoded());
                        out.write(b);
                 }

                out.write(iv);
                Cipher ci = Cipher.getInstance("AES/CBC/PKCS5Padding");
                ci.init(Cipher.ENCRYPT_MODE, skey, ivspec);
             long et = System.nanoTime()-startTime;
       System.out.println("Encryption using AES RSA: "+et);
                try (FileInputStream in = new FileInputStream(inputFile))
                 {
                        processFile(ci, in, out);
                   }
           } catch (NoSuchProviderException e) {

                        e.printStackTrace();
                }
}
```

```
   static private void doDecryptRSAWithAES(String pubKeyFile,String encryptedFile) throws
java.security.NoSuchAlgorithmException,java.security.InvalidAlgorithmParameterException,
                java.security.InvalidKeyException,
                java.security.spec.InvalidKeySpecException,
                javax.crypto.NoSuchPaddingException,
                javax.crypto.BadPaddingException,
                javax.crypto.IllegalBlockSizeException,
                java.io.IOException
  {

          byte[] bytes = Files.readAllBytes(Paths.get(pubKeyFile));
           long startTime = System.nanoTime();
          X509EncodedKeySpec ks = new X509EncodedKeySpec(bytes);
          KeyFactory kf = KeyFactory.getInstance("RSA");
          PublicKey pub = kf.generatePublic(ks);

          try (FileInputStream in = new FileInputStream(encryptedFile)) {
                    SecretKeySpec skey = null;
                    {
                              Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
                              Cipher cipher = Cipher.getInstance("RSA/None/OAEPWithSHA1AndMGF1Padding",
"BC");
                      cipher.init(Cipher.DECRYPT_MODE, pub);
                      byte[] b = new byte[256];
                      in.read(b);
                      byte[] keyb = cipher.doFinal(b);
                      skey = new SecretKeySpec(keyb, "AES");
                    }

                    byte[] iv = new byte[128/8];
                    in.read(iv);
                    IvParameterSpec ivspec = new IvParameterSpec(iv);

           Cipher ci = Cipher.getInstance("AES/CBC/PKCS5Padding");
           ci.init(Cipher.DECRYPT_MODE, skey, ivspec);
           long et = System.nanoTime()-startTime;
      System.out.println("decryption using AES RSA: "+et);

          String file="decrypted_"+encryptedFile.substring(0,encryptedFile.indexOf('.'));
                    try (FileOutputStream out = new FileOutputStream(file+".txt")){
                              processFile(ci, in, out);
                      }
            } catch (NoSuchProviderException e) {
                              // TODO Auto-generated catch block
                              e.printStackTrace();
                    }
  }
```

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


            String inputFileName1="oneKB.txt";
             String inputFileName2="oneMB.txt";
             String keyName="rsaalgorithm";
String file1="encrypted_"+inputFileName1.substring(0,inputFileName1.indexOf('.'));
String file2="encrypted_"+inputFileName2.substring(0,inputFileName2.indexOf('.'));
             sample1.doGenkey(keyName);
             sample1.doEncrypt("rsaalgorithm.key",inputFileName1);
             sample1.doDecrypt("rsaalgorithm.pub",file1+".enc");
             sample1.doEncryptRSAWithAES("rsaalgorithm.key",inputFileName2);
             sample1.doDecryptRSAWithAES("rsaalgorithm.pub",file2+".enc");

   }
}
```

**Output**:

```
Key generation: 608380939
Time taken for encryption: 963634300
Per Byte Speed:941049
Time taken for decryption: 7849397
Per Byte Speed:7665
Time taken for encryption: 205275569
Per Byte Speed:195
Time taken for decryption: 132766890
Per Byte Speed:126
```

**g) Create a 2048-bit DSA key, sign the two files and verify the corresponding signatures.**

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.DSAPrivateKey;
import java.security.interfaces.DSAPublicKey;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.security.SecureRandom;
import java.security.Signature;


public class DSA2048
{
public static void main(String[] args)
{
  try
  {
            generateDSA2048KeyPair();
            File inputFile1=new File("oneKB.txt");
            File inputFile2=new File("oneMB.txt");
            System.out.println("1 KB File");
            sign(inputFile1);
            verify(new

           File(inputFile1.getName().substring(0,inputFile1.getName().indexOf('.'))+"_signedData"));
            System.out.println("1 MB File");
            sign(inputFile2);
            verify(new

            File(inputFile2.getName().substring(0,inputFile2.getName().indexOf('.'))+"_signedData"));
   }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

```
public static void generateDSA2048KeyPair() {
        try {
                long startTime= System.nanoTime();
                KeyPairGenerator pairgen = KeyPairGenerator.getInstance("dsa");
                SecureRandom random = new SecureRandom();
                pairgen.initialize(2048, random);
                KeyPair keyPair = pairgen.generateKeyPair();
                DSAPrivateKey privateKey = (DSAPrivateKey) keyPair.getPrivate();
                DSAPublicKey publicKey = (DSAPublicKey) keyPair.getPublic();
                long estimatedTime= System.nanoTime()-startTime;
                System.out.println("Time taken for DSA -2048 Key Generation: "+     estimatedTime);
                ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("public.pub"));
                out.writeObject(publicKey);
                out.close();
                out = new ObjectOutputStream(new FileOutputStream("private.key"));
                out.writeObject(privateKey);
                out.close();
          }
    catch (Exception e)
         {
                e.printStackTrace();
        }
}

public static long sign(File infile)
{
try {

             ObjectInputStream keyIn = new ObjectInputStream(new   FileInputStream("private.key"));
             PrivateKey privkey = (PrivateKey) keyIn.readObject();
             keyIn.close();
             InputStream in = new FileInputStream(infile);
             int length = (int) infile.length();
             byte[] message = new byte[length];
             in.read(message, 0, length);
             in.close();
             long startTime=System.nanoTime();
             Signature signalg = Signature.getInstance("SHA256withDSA");
             signalg.initSign(privkey);
             signalg.update(message);
             byte[] signature = signalg.sign();
             long estimatedTime=System.nanoTime()-startTime;
             System.out.println("Time taken for Signing the file: "+ estimatedTime );
             DataOutputStream out = new DataOutputStream(new
        FileOutputStream(infile.getName().substring(0,infile.getName().indexOf('.'))+"_signedData"));
             int signlength = signature.length;
             out.writeInt(signlength);
             out.write(signature, 0, signlength);
```

```
                out.write(message, 0, length);
                out.close();
               return estimatedTime;
}
catch(Exception e)
{
                        e.printStackTrace();
}
return 0;

}

public static long verify(File infile)
{
try {
   ObjectInputStream keyIn = new ObjectInputStream(new  FileInputStream("public.pub"));
   PublicKey pubkey = (PublicKey) keyIn.readObject();
   keyIn.close();
   DataInputStream in = new DataInputStream(new FileInputStream(infile));
   int signlength = in.readInt();
   byte[] signature = new byte[signlength];
   in.read(signature, 0, signlength);
   int length = (int) infile.length() - signlength - 4;
   byte[] message = new byte[length];
   in.read(message, 0, length);
   in.close();
   long startTime=System.nanoTime();
   Signature verifyalg = Signature.getInstance("sha256withDSA");
   verifyalg.initVerify(pubkey);
   verifyalg.update(message);
   boolean verify=verifyalg.verify(signature);
   long estimatedTime=System.nanoTime()-startTime;
   System.out.println("Time taken for verifying the signed file: "+ estimatedTime );
   if (!verify) System.out.print("not ");
   System.out.println("Signature of the given file is verified!");
   return estimatedTime;
}
catch(Exception e)
{
          e.printStackTrace();
}
return 0;
          }

}
```

**Output:**

Time taken for DSA -2048 Key Generation: 188169200
1 KB File
Time taken for Signing the file: 15363433
per byte speed of signing: 161720
Time taken for verifying the signed file: 10584760
per byte speed of verification: 65743
Signature of the given file is verified!
1 MB File
Time taken for Signing the file: 56701939
per byte speed of signing: 54
Time taken for verifying the signed file: 21678534
per byte speed of verification: 20
Signature of the given file is verified!

**h) Create a 3072-bit DSA key, sign the two files and verify the corresponding signatures.**

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.DSAPrivateKey;
import java.security.interfaces.DSAPublicKey;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.security.SecureRandom;
import java.security.Signature;

public class DSA3072
{
public static void main(String[] args)
{
  try
  {
            generateDSA2048KeyPair();
             File inputFile1=new File("oneKB.txt");
             File inputFile2=new File("oneMB.txt");
             System.out.println("1 KB File");
             sign(inputFile1);
             verify(new
             File(inputFile1.getName().substring(0,inputFile1.getName().indexOf('.'))+"_signedData"));
             System.out.println("1 MB File");
             sign(inputFile2);
             verify(new
              File(inputFile2.getName().substring(0,inputFile2.getName().indexOf('.'))+"_signedData"));

  }
  catch (Exception e)
  {
              e.printStackTrace();
  }
}
public static void generateDSA2048KeyPair() {
          try
        {
                    long startTime= System.nanoTime();
                    KeyPairGenerator pairgen = KeyPairGenerator.getInstance("dsa");
                    SecureRandom random = new SecureRandom();
```

```
                pairgen.initialize(3072, random);
                KeyPair keyPair = pairgen.generateKeyPair();
                DSAPrivateKey privateKey = (DSAPrivateKey) keyPair.getPrivate();
                 DSAPublicKey publicKey = (DSAPublicKey) keyPair.getPublic();
                long estimatedTime= System.nanoTime()-startTime;
                System.out.println("Time taken for DSA -2048 Key Generation: "+ estimatedTime);
                ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("public.pub"));
                out.writeObject(publicKey);
                out.close();
                out = new ObjectOutputStream(new FileOutputStream("private.key"));
                out.writeObject(privateKey);
                out.close();
        }
        catch (Exception e)
        {
                e.printStackTrace();
        }
}

public static void sign(File infile)
{
 try
 {
    ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream("private.key"));
    PrivateKey privkey = (PrivateKey) keyIn.readObject();
    keyIn.close();
    InputStream in = new FileInputStream(infile);
    int length = (int) infile.length();
    byte[] message = new byte[length];
    in.read(message, 0, length);
    in.close();
    long startTime=System.nanoTime();
    Signature signalg = Signature.getInstance("SHA256withDSA");
    signalg.initSign(privkey);
    signalg.update(message);
    byte[] signature = signalg.sign();
    long estimatedTime=System.nanoTime()-startTime;
    System.out.println("Time taken for Signing the file: "+ estimatedTime );
    DataOutputStream out = new DataOutputStream(new

    FileOutputStream(infile.getName().substring(0,infile.getName().indexOf('.'))+"_signedData"));
    int signlength = signature.length;
    out.writeInt(signlength);
    out.write(signature, 0, signlength);
    out.write(message, 0, length);
    out.close();
}
catch(Exception e)
```

```
{
        e.printStackTrace();
}

}

public static void verify(File infile) {
try {
            ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream("public.pub"));
            PublicKey pubkey = (PublicKey) keyIn.readObject();
            keyIn.close();
            DataInputStream in = new DataInputStream(new FileInputStream(infile));
            int signlength = in.readInt();
            byte[] signature = new byte[signlength];
            in.read(signature, 0, signlength);
            int length = (int) infile.length() - signlength - 4;
            byte[] message = new byte[length];
            in.read(message, 0, length);
            in.close();
            long startTime=System.nanoTime();
            Signature verifyalg = Signature.getInstance("sha256withDSA");
            verifyalg.initVerify(pubkey);
            verifyalg.update(message);
            boolean verify=verifyalg.verify(signature);
            long estimatedTime=System.nanoTime()-startTime;
            System.out.println("Time taken for verifying the signed file: "+ estimatedTime );
            if (!verify) System.out.print("not ");
             System.out.println("The signature of the given file is verified.");
}
        catch(Exception e)
         {
                e.printStackTrace();
         }
   }









}
```

**Output:**
Time taken for DSA -2048 Key Generation in ns: 202105859
1 KB File
Time taken for Signing the file in ns: 16862315
Per byte speed of signature in ns: 177498
Time taken for verifying the signed file in ns: 33516416
Per byte speed of verification in ns: 198321
The signature of the given file is verified.
1 MB File
Time taken for Signing the file: 40405385
Per byte speed of signature: 38
Time taken for verifying the signed file: 30245159
Per byte speed of verification: 28
The signature of the given file is verified.

---

**iii)** The following code runs on the server: **timberlake.cse.buffalo.edu** that has the support of jdk 1.8

- **Bouncy Castle Crypto APIs** are used in one or more of the programs by configuring the build path of the programs to add external jar files.
- The above codes are run and tested for correctness on local machine using java eclipse IDE that uses jdk 1.8
- The local machine configuration is as follows
- Procesor: Intel core i3 $7^{th}$ gen
- System Type: 64 bit operating system,X86 processor
- AES-NI: No

---

**iv)**

**How per byte speed changes for different algorithms between small and large files**

Encryption Algorithms:

- It is observed that the per byte speed of encryption and hashing increases with the increase in the file size which means that smaller files take more time for encrypting a single byte of file than the larger file.
- The per byte speed of encryption is more than the per byte speed of decryption for both small and large files in CBC and CTR modes of AES with key size of 128 bits.
- The per byte speed of encryption is less than the per byte speed of decryption for smaller files in CTR mode of AES with key size of 256 bits.
- Per byte speed comparison
    - AES_CTR_128 > AES_CBC_128 > AES_CTR_256

Hashing :

- The per byte speed of Hashing algorithms increases with the increase in the file size.
- The per byte speed of hashing using sha-512 is more when compared to the other two for small files where as sha-256 has the least per byte speed of hashing.

- For larger files, the per byte speed of hashing for sha-256,sha-512 and sha3-256 are more or less the same but it is observed that sha-256 has lower speed of hashing per byte relatively.

Signature:

- It is observed that the DSA 3072 takes relatively less time than DSA 2048 for signing the file where as DSA 2048 takes relatively less time than DSA 3072 during verification of the signature.
- The per byte speed of signing the file is more for DSA 3072 when compared to DSA 2048 where as the per byte speed of verifying the file is more for DSA 2048 when compared to DSA 3072.

**How encryption and decryption times differ for a given encryption algorithm**

Small Files:

- The encryption time in CBC mode of AES is less than the CTR mode with key size of 128 bits but is more than the CTR mode of AES with key size of 256 bits.
- The encryption time of small files using RSA algorithms is more than the encryption time using AES algorithms .
- The decryption time of small files using RSA Algorithms is less than or equal to the decryption time using AES algorithms.
- The decryption time is more for AES in CTR mode with the key bit size of 256 bits than its encryption.
- The decryption time for AES in CBC and CTR with key size of 128 bits is less than the encryption time.
- DSA 2048 has better performance for files with smaller sizes both in terms of signing and verifying the data when compared to DSA 3072

Large files:

- The encryption time for RSA algorithms is more than the encryption time for AES algorithms is more or less the same for large files.
- The decryption time for AES in CTR mode is less when compared to CBC mode of AES. CTR gives better performance for files with large sizes than CBC mode of AES.
- The decryption time of large files using RSA Algorithms is less than the decryption time using AES algorithms
- The signing and verification of larger files with DSA 3072 has better performance in terms of speed when compare to DSA 2048;

**how key generation, encryption, and decryption times differ with the increase in the key size**.

**Key Generation time**

- The time taken for key generation remains more or less the same for both CTR and CBC modes of AES irrespective of the key size.
- The key generation time for RSA-3072 is far more than the key generation time for RSA-2048.

**AES in CTR mode**

- Encryption time for AES in CTR mode with key size 256 is more than the one with key size 128.
- Decryption time for AES in CTR mode with key size 256 is more than its encryption time. It is also more than the decryption time of AES algorithm in CTR mode with 128 bit key.

**RSA**

- Encryption time for RSA Algorithm with key size 2048 is more than the one with key size 3072.
- Decryption time of RSA Algorithms is far less than its encryption time for any key size.

**How hashing time differs between the algorithms and with increase of the key size**

- Using secure hash algorithm we implement repeated routines of encryption and decryption on fixed block size.As a result, the hashing time increases with the increase in the size of the text.
- It is observed that sha-512 is faster than sha-256 for the files of size 1KB and 1 MB but sometimes Sha-256 is faster than sha-512 for shorter strings.
- Sha-512 works on large data chunks (1024 bit blocks) where as sha-256 relatively smaller data chunks(512 bit blocks).

**Problem 2:**

Given set of rights {read, write, execute, append, list, modify, own}.

If delete_all_rights(s1, s2, o) is a command that causes subject $S_1$ to delete all rights the subject $S_2$ has over object o only if $S_1$ has modify rights over o and $S_2$ does not have own rights over o.

The following is the syntactic notation of it.

**Pre-condition**: m* ∈ $A_i$ [$S_1$, o] or own ∈ $A_i$ [$S_1$, o] and own ∉ $A_i$ [$S_2$, o] where m is modify

**Command** : **delete_all_rights(s1, s2, o)**

Before deleting all the rights, we have to check if $S_2$ has no own right but there is no command in primitive list that can check for the absence of right. Hence we create a temporary subject and destroy it again after we are done with the task of checking for the absence of a specific right in a subject.

Let the temporary subject be named as s.

If there $S_2$ has own right, read right in A[s,o] will be deleted which will make the further commands to be not executed as it returns false when checked for read in A[s,o].

**delete_all_rights($S_1$,$S_2$,o)**

create subject s
enter read in A[s, o]
if own in A [q, o] then
delete read from a[s, o]

if modify in A [$S_1$, o] and read in A [s, o] then
delete read in A [$S_2$,o]
delete write in A a[$S_2$, o]
delete execute in A [$S_2$, o]
delete append in A [$S_2$, o]
delete list in A [$S_2$, o]
delete modify in A [$S_2$, o]
delete own in A [$S_2$, o]
destroy subject s
end

**Post-condition** :

$A_{i+1}$[$S_2$, o] = $A_i$ [$S_2$, o] - { r, w, e, a, l, m}

⇨ $A_{i+1}$[$S_2$, o] = Ø

b)

**copy all rights(s1, s2, o)**

A command that copies all rights that s1 has over o to s2 in such a way that only those rights with an associated copy flag are copied and the new copy doesn't have the copy flag.

Here we first check if the $S_1$ subject has particular copy flag associated with a right. We then add the right with no copy flag to the subject $S_2$

if read* in A[$S_1$,o]
 then enter read into A[$S_2$,o]

if write* in A[$S_1$,o]
 then enter write into A[$S_2$,o]

if execute* in A[$S_1$,o]
 then enter execute into A[$S_2$,o]

if append* in A[$S_1$,o]
 then enter append into A[$S_2$,o]

if list* in A[$S_1$,o]
 then enter list into A[$S_2$,o]

if modify* in A[$S_1$,o]
 then enter modify into A[$S_2$,o]

end

**References**


1. Access Control Matrix Model Retrieved Sept 29,2018, from http://nob.cs.ucdavis.edu/classes/ecs235b-2014-01/slides/2014-01-14.pdf
2. A Performance Comparison of Data Encryption Algorithms, Retrieved Sept 29,2018, from https://ieeexplore.ieee.org/abstract/document/1598556