Homework 03

Md Moniruzzaman Monir, # 50291708

October 04, 2018

Problem 01

Solution:

a) Key Generation time : 50576025 ns
Small File :
Encryption time: 967631 ns, Speed : 0.0010334518 byte/ns
Decryption time: 353622 ns, Speed : 0.00282787835 byte/ns

Large File :
Encryption time: 52784213 ns, Speed : 0.01894505844 byte/ns
Decryption time: 49815759 ns, Speed : 0.02007396896 byte/ns

b) Key Generation time : 50576025 ns
Small File :
Encryption time: 192661 ns, Speed : 0.00519046407 byte/ns
Decryption time: 147738 ns, Speed : 0.00676873925 byte/ns

Large File :
Encryption time: 28661383 ns, Speed : 0.03489015167 byte/ns
Decryption time: 18613272 ns, Speed : 0.05372510539 byte/ns

c) Key Generation time : 162847 ns
Small File :
Encryption time: 260733 ns, Speed : 0.00383534113 byte/ns
Decryption time: 164488 ns, Speed : 0.00607947084 byte/ns

Large File :
Encryption time: 20797468 ns, Speed : 0.04808277623 byte/ns
Decryption time: 23676585 ns, Speed : 0.04223582074 byte/ns

d) **Small File :**
SHA-256 Hashing: 113010.40 ns, Speed: 0.00884874312 bytes/ns
SHA-512 Hashing: 20970.83 ns, Speed: 0.04766255672 bytes/ns
SHA3-256 Hashing: 20029.16 ns, Speed: 0.04993219208 bytes/ns

Large File :
SHA-256 Hashing: 4993200.30 ns, Speed: 0.20027235839 bytes/ns
SHA-512 Hashing: 3090243.20 ns, Speed: 0.32359912643 bytes/ns
SHA3-256 Hashing: 6555080.41 ns, Speed: 0.15255342992 bytes/ns

e) RSA 2048-bit Key Generation time: 227632900.78 ns

Small File :
Encryption time : 757978.98 ns, Speed: 0.00131929779 bytes/ns
Decryption time : 4120038.98 ns, Speed: 0.00024271615 bytes/ns

Large File :
Encryption time : 4084877104.62 ns, Speed: 0.0002448054 bytes/ns
Decryption time : 11215083103.2 ns, Speed: 0.00008916563 bytes/ns

f) RSA 3072-bit Key Generation time: 269256942.368 ns

Small File :
Encryption time : 3570960.99 ns, Speed: 0.00028003666 bytes/ns
Decryption time : 6827043.533 ns, Speed: 0.00014647628 bytes/ns

Large File :
Encryption time : 5731897115.71 ns, Speed: 0.0001744623 bytes/ns
Decryption time : 28755152027.50 ns, Speed: 0.00003477637 bytes/ns

g) DSA 2048-bit Key Generation time: 1069301969.91 ns

Small File :
Signing time : 527948.2116 ns, Speed: 0.00189412517 bytes/ns
Verify time : 782026.518775 ns, Speed: 0.00127872901 bytes/ns

Large File :
Signing time : 4876987.838 ns, Speed: 0.2050445958 bytes/ns
Verify time : 5151887.075 ns, Speed:0.19410363337 bytes/ns

h) DSA 3072-bit Key Generation time: 4722790002.82 ns

Small File :
Signing time : 983013.38 ns, Speed: 0.00101728015 bytes/ns
Verify time : 1699073.41 ns, Speed: 0.00058855608 bytes/ns

Large File :
Signing time : 5384104.15 ns, Speed: 0.18573191976 bytes/ns
Verify time : 5887882.61 ns, Speed: 0.16984034265 bytes/ns

All Source Codes:

Java codes for problem 1) a,b,c,d

```
package cryptography;

import javax.crypto.KeyGenerator; // For Key Generaion
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import javax.crypto.SecretKey;
import java.security.MessageDigest;
//import java.util.concurrent.TimeUnit;

public class app {

 public static void main(String[] args) {

  try {

   long startTime = System.nanoTime();
  KeyGenerator kgen = KeyGenerator.getInstance("AES");
  kgen.init(128); // 128 bit
  SecretKey key = kgen.generateKey();

  long endTime = System.nanoTime();

  long keyGenerationTime128 = (endTime - startTime);

  System.out.println("128 bit Key Generation Time in nanoseconds: " + keyGenerationTime128);

  AesEncryptionDecryptionCBC encrypter = new AesEncryptionDecryptionCBC(key);

  startTime = System.nanoTime();
  encrypter.encrypt(new FileInputStream("smallFile_1KB.txt"), new FileOutputStream("SmallFileEncrypted.txt"));
  endTime = System.nanoTime();

  long CBC_SmallFileEncryptionTime128 = (endTime - startTime);

  System.out.println(
   "Small file (1KB) Encryption time in nanoseconds - CBC Mode: " + CBC_SmallFileEncryptionTime128);
  startTime = System.nanoTime();
  encrypter.decrypt(new FileInputStream("SmallFileEncrypted.txt"),
   new FileOutputStream("SmallFileDecrypted.txt"));
  endTime = System.nanoTime();

  long CBC_SmallFileDecryptionTime128 = (endTime - startTime);

  System.out.println(
   "Small file (1KB) Decryption time in nanoseconds - CBC Mode: " + CBC_SmallFileDecryptionTime128);
  startTime = System.nanoTime();
  encrypter.encrypt(new FileInputStream("largeFile_1MB.txt"), new FileOutputStream("LargeFileEncrypted.txt"));
```

```
endTime = System.nanoTime();

long CBC_LargeFileEncryptionTime128 = (endTime - startTime);

System.out.println(
 "Large file (1MB) Encryption time in nanoseconds - CBC Mode: " + CBC_LargeFileEncryptionTime128);
startTime = System.nanoTime();
encrypter.decrypt(new FileInputStream("LargeFileEncrypted.txt"),
 new FileOutputStream("LargeFileDecrypted.txt"));
endTime = System.nanoTime();

long CBC_LargeFileDecryptionTime128 = (endTime - startTime);

System.out.println(
 "Large file (1MB) Decryption time in nanoseconds - CBC Mode: " + CBC_LargeFileDecryptionTime128);

AesEncryptionDecryptionCTR encrypter1 = new AesEncryptionDecryptionCTR(key);

startTime = System.nanoTime();
encrypter1.encrypt(new FileInputStream("smallFile_1KB.txt"),
 new FileOutputStream("SmallFileEncryptedCTR.txt"));
endTime = System.nanoTime();

long CTR_SmallFileEncryptionTime128 = (endTime - startTime);

System.out.println(
 "Small file (1KB) Encryption time in nanoseconds - CTR Mode: " + CTR_SmallFileEncryptionTime128);

startTime = System.nanoTime();
encrypter1.decrypt(new FileInputStream("SmallFileEncryptedCTR.txt"),
 new FileOutputStream("SmallFileDecryptedCTR.txt"));
endTime = System.nanoTime();

long CTR_SmallFileDecryptionTime128 = (endTime - startTime);
System.out.println(
 "Small file (1KB) Decryption time in nanoseconds - CTR Mode: " + CTR_SmallFileDecryptionTime128);

startTime = System.nanoTime();
encrypter1.encrypt(new FileInputStream("largeFile_1MB.txt"),
 new FileOutputStream("LargeFileEncryptedCTR.txt"));
endTime = System.nanoTime();

long CTR_LargeFileEncryptionTime128 = (endTime - startTime);

System.out.println(
 "Large file (1MB) Encryption time in nanoseconds - CTR Mode: " + CTR_LargeFileEncryptionTime128);

// Decrypt
startTime = System.nanoTime();
encrypter1.decrypt(new FileInputStream("LargeFileEncryptedCTR.txt"),
 new FileOutputStream("LargeFileDecryptedCTR.txt"));
endTime = System.nanoTime();
```

```
long CTR_LargeFileDecryptionTime128 = (endTime - startTime);

System.out.println(
 "Large file (1MB) Decryption time in nanoseconds - CTR Mode: " + CTR_LargeFileDecryptionTime128);
startTime = System.nanoTime();

KeyGenerator kgen1 = KeyGenerator.getInstance("AES");
kgen1.init(256); // 256 bit
SecretKey key1 = kgen1.generateKey();

endTime = System.nanoTime();

long keyGenerationTime256 = (endTime - startTime);

System.out.println("256 bit Key Generation Time in nanoseconds: " + keyGenerationTime256);

AesEncryptionDecryptionCTR encrypter2 = new AesEncryptionDecryptionCTR(key1);
startTime = System.nanoTime();
encrypter2.encrypt(new FileInputStream("smallFile_1KB.txt"),
 new FileOutputStream("SmallFileEncryptedCTR256.txt"));
endTime = System.nanoTime();

long CTR_SmallFileEncryptionTime256 = (endTime - startTime);

System.out.println("Small file (1KB) Encryption time in nanoseconds - CTR Mode - 256 BIT KEY: " +
 CTR_SmallFileEncryptionTime256);
startTime = System.nanoTime();
encrypter2.decrypt(new FileInputStream("SmallFileEncryptedCTR256.txt"),
 new FileOutputStream("SmallFileDecryptedCTR256.txt"));
endTime = System.nanoTime();

long CTR_SmallFileDecryptionTime256 = (endTime - startTime);

System.out.println("Small file (1KB) Decryption time in nanoseconds - CTR Mode - 256 BIT KEY:" +
 CTR_SmallFileDecryptionTime256);

startTime = System.nanoTime();
encrypter2.encrypt(new FileInputStream("largeFile_1MB.txt"),
 new FileOutputStream("LargeFileEncryptedCTR256.txt"));
endTime = System.nanoTime();

long CTR_LargeFileEncryptionTime256 = (endTime - startTime);

System.out.println("Large file (1MB) Encryption time in nanoseconds - CTR Mode - 256 BIT KEY: " +
 CTR_LargeFileEncryptionTime256);

startTime = System.nanoTime();
encrypter2.decrypt(new FileInputStream("LargeFileEncryptedCTR256.txt"),
 new FileOutputStream("LargeFileDecryptedCTR256.txt"));
endTime = System.nanoTime();

long CTR_LargeFileDecryptionTime256 = (endTime - startTime);
```

```
System.out.println("Large file (1MB) Decryption time in nanoseconds - CTR Mode - 256 BIT KEY: " +
 CTR_LargeFileDecryptionTime256);

String[] HashArray = {
 "SHA-256",
 "SHA-512",
 "SHA3-256"
};
int m;
for (m = 0; m < 3; m++) {

 MessageDigest md = MessageDigest.getInstance(HashArray[m]);

 startTime = System.nanoTime();

 try {
  FileInputStream fis = new FileInputStream("smallFile_1KB.txt");
  byte[] dataBytes = new byte[1024];

  int nread = 0;
  while ((nread = fis.read(dataBytes)) != -1) {
   md.update(dataBytes, 0, nread);
  }
  fis.close();
 } catch (FileNotFoundException e) {
  System.out.println("File not found...");
 }

 byte[] mdbytes = md.digest();

 endTime = System.nanoTime();

 // convert the byte to hex format method
 StringBuffer sb = new StringBuffer();
 for (int i = 0; i < mdbytes.length; i++) {
  sb.append(Integer.toString((mdbytes[i] & 0xff) + 0x100, 16).substring(1));
 }

 // System.out.println("Hex format SHA 256 Hash Value of small
 // file is: "+ sb.toString());

 long hashTime = (endTime - startTime);
 System.out.println(HashArray[m] + "- Hashing time in nanoseconds - SMALL FILE " + hashTime);

 // LARGE FILE

 MessageDigest md2 = MessageDigest.getInstance(HashArray[m]);

 startTime = System.nanoTime();

 try {
  FileInputStream fis1 = new FileInputStream("largeFile_1MB.txt");
  byte[] dataBytes = new byte[1024];
```

```
    int nread = 0;
    while ((nread = fis1.read(dataBytes)) != -1) {
     md2.update(dataBytes, 0, nread);
    }
    fis1.close();
   } catch (FileNotFoundException e) {

   }

   byte[] mdbytes1 = md2.digest();

   endTime = System.nanoTime();
   // convert the byte to hex format method
   StringBuffer sb1 = new StringBuffer();
   for (int i = 0; i < mdbytes1.length; i++) {
    sb1.append(Integer.toString((mdbytes1[i] & 0xff) + 0x100, 16).substring(1));
   }

   hashTime = (endTime - startTime);
   System.out.println(HashArray[m] + "- Hashing time in nanoseconds- LARGE FILE " + hashTime);

  }

 } catch (Exception e) {
  e.printStackTrace();
 }
}
}

package cryptography;
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;

import java.io.InputStream;
import java.io.OutputStream;


import java.security.spec.AlgorithmParameterSpec;

public class AesEncryptionDecryptionCBC {

            Cipher ecipher;
            Cipher dcipher;

            public AesEncryptionDecryptionCBC(SecretKey key) {
             // Create an 8-byte initialization vector
             byte[] iv = new byte[]
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
```

```
            AlgorithmParameterSpec paramSpec = new IvParameterSpec(iv);

            try {
             ecipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
             dcipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

             // CBC/CTR requires an initialization vector
             ecipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
             dcipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
            } catch (Exception e) {
             e.printStackTrace();
            }
           }

           // Buffer used to transport the bytes from one stream to another
           byte[] buf = new byte[1024];

           public void encrypt(InputStream in , OutputStream out) {
            try {
             // Bytes written to out will be encrypted
             out = new CipherOutputStream(out, ecipher);

             // Read in the clear text bytes and write to out to encrypt
             int numRead = 0;
             while ((numRead = in .read(buf)) >= 0) {
              out.write(buf, 0, numRead);
             }
             out.close();
            } catch (java.io.IOException e) {}
           }

           public void decrypt(InputStream in , OutputStream out) {
            try {
             // Bytes read from in will be decrypted
             in = new CipherInputStream( in , dcipher);

             // Read in the decrypted bytes and write the cleartext to out
             int numRead = 0;
             while ((numRead = in .read(buf)) >= 0) {
              out.write(buf, 0, numRead);
             }
             out.close();
            } catch (java.io.IOException e) {}
           }

}

package cryptography;
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
```

```
import java.io.InputStream;
import java.io.OutputStream;


import java.security.spec.AlgorithmParameterSpec;

public class AesEncryptionDecryptionCTR {
        Cipher ecipher;
        Cipher dcipher;

        public AesEncryptionDecryptionCTR(SecretKey key) {
         // Create an 8-byte initialization vector
         byte[] iv = new byte[]
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};

         AlgorithmParameterSpec paramSpec = new IvParameterSpec(iv);
         try {
          ecipher = Cipher.getInstance("AES/CTR/PKCS5Padding");
          dcipher = Cipher.getInstance("AES/CTR/PKCS5Padding");

          // CBC/CTR requires an initialization vector
          ecipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
          dcipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
         } catch (Exception e) {
          e.printStackTrace();
         }
        }

        // Buffer used to transport the bytes from one stream to another
        byte[] buf = new byte[1024];

        public void encrypt(InputStream in , OutputStream out) {
         try {
          // Bytes written to out will be encrypted
          out = new CipherOutputStream(out, ecipher);

          // Read in the clear text bytes and write to out to encrypt
          int numRead = 0;
          while ((numRead = in .read(buf)) >= 0) {
           out.write(buf, 0, numRead);
          }
          out.close();
         } catch (java.io.IOException e) {}
        }

        public void decrypt(InputStream in , OutputStream out) {
         try {
          // Bytes read from in will be decrypted
          in = new CipherInputStream( in , dcipher);

          // Read in the decrypted bytes and write the cleartext to out
          int numRead = 0;
```

```
            while ((numRead = in .read(buf)) >= 0) {
             out.write(buf, 0, numRead);
            }
            out.close();
           } catch (java.io.IOException e) {}
          }

}
```

Python codes for problem 1) e,f,g,h

```
from Crypto.PublicKey import RSA
from timeit import default_timer

start = default_timer ()
key = RSA.generate(2048)
end = default_timer ()
print('2048 bit RSA key generation time in microsecond '+ str(end-start))

public_key_2048 =  key.publickey()
private_key_2048 = key

from Crypto.Cipher import PKCS1_OAEP

cipher = PKCS1_OAEP.new(public_key_2048)

start = default_timer ()
with open("smallFile_1KB.txt", "rb") as in_file, open("smallFile_1KB_Enc.txt", "wb") as out_file:
   while True:
      piece = in_file.read(128)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.encrypt(piece))
end = default_timer ()
print('Small file encryption time in microsecond using 2048 bit RSA  '+ str(end-start))

start = default_timer ()
with open("largeFile_1MB.txt", "rb") as in_file, open("largeFile_1MB_Enc.txt", "wb") as out_file:
   while True:
      piece = in_file.read(128)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.encrypt(piece))
end = default_timer ()
print('Large file encryption time in microsecond using 2048 bit RSA  '+ str(end-start))

cipher = PKCS1_OAEP.new(private_key_2048)
```

```
start = default_timer ()
with open("smallFile_1KB_Enc.txt", "rb") as in_file, open("smallFile_1KB_Dec.txt", "wb") as out_file:
   while True:
      piece = in_file.read(256)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.decrypt(piece))
end = default_timer ()
print('Small file decryption time in microsecond using 2048 bit RSA  '+ str(end-start))

start = default_timer ()
with open("largeFile_1MB_Enc.txt", "rb") as in_file, open("largeFile_1MB_Dec.txt", "wb") as out_file:
   while True:
      piece = in_file.read(256)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.decrypt(piece))
end = default_timer ()
print('Large file decryption time in microsecond using 2048 bit RSA  '+ str(end-start))


from Crypto.PublicKey import RSA
from timeit import default_timer

start = default_timer ()
key = RSA.generate(3072)
end = default_timer ()
print('3072 bit key generation time in microsecond '+ str(end-start))

public_key_3072 =  key.publickey()
private_key_3072 = key

from Crypto.Cipher import PKCS1_OAEP

cipher = PKCS1_OAEP.new(public_key_3072)

start = default_timer ()
with open("smallFile_1KB.txt", "rb") as in_file, open("smallFile_1KB_Enc.txt", "wb") as out_file:
   while True:
      piece = in_file.read(128)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.encrypt(piece))
end = default_timer ()
print('Small file encryption time in microsecond using 3072 bit RSA  '+ str(end-start))

start = default_timer ()
with open("largeFile_1MB.txt", "rb") as in_file, open("largeFile_1MB_Enc.txt", "wb") as out_file:
```

```
   while True:
      piece = in_file.read(128)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.encrypt(piece))
end = default_timer ()
print('Large file encryption time in microsecond using 3072 bit RSA  '+ str(end-start))

cipher = PKCS1_OAEP.new(private_key_3072)

start = default_timer ()
with open("smallFile_1KB_Enc.txt", "rb") as in_file, open("smallFile_1KB_Dec.txt", "wb") as out_file:
   while True:
      piece = in_file.read(384)  # 3072/8
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.decrypt(piece))
end = default_timer ()
print('Small file decryption time in microsecond using 3072 bit RSA  '+ str(end-start))

start = default_timer ()
with open("largeFile_1MB_Enc.txt", "rb") as in_file, open("largeFile_1MB_Dec.txt", "wb") as out_file:
   while True:
      piece = in_file.read(384)
      if len(piece)==0:
         break # end of file
      out_file.write(cipher.decrypt(piece))
end = default_timer ()
print('Large file decryption time in microsecond using 3072 bit RSA  '+ str(end-start))


import hashlib
from Crypto.PublicKey import DSA
from Crypto.Signature import DSS
from timeit import default_timer

start = default_timer ()
key = DSA.generate(2048)
end = default_timer()
print('2048 bit DSA key generation time in microsecond '+ str(end-start))

sha256_hash = hashlib.sha256()

start = default_timer ()

with open("smallFile_1KB.txt","rb") as f:
   # Read and update hash string value in blocks of 4K
   for byte_block in iter(lambda: f.read(4096),b""):
```

```
        sha256_hash.update(byte_block)
    hash_obj = sha256_hash.hexdigest()

signer = DSS.new(key, 'fips-186-3')
signature = key.sign(hash_obj)

pub_key = key.publickey()

if pub_key.verify(hash_obj, signature):
    print "OK"
else:
    print "Incorrect signature"

end = default_timer ()

print('Using 2048 bit DSA key signing and verification time in microsecond '+ str(end-start))

sha256_hash = hashlib.sha256()

start = default_timer()

with open("largeFile_1KB.txt","rb") as f:
    # Read and update hash string value in blocks of 4K
    for byte_block in iter(lambda: f.read(4096),b""):
        sha256_hash.update(byte_block)
    hash_obj = sha256_hash.hexdigest()

signer = DSS.new(key, 'fips-186-3')
signature = key.sign(hash_obj)

pub_key = key.publickey()

if pub_key.verify(hash_obj, signature):
    print "OK"
else:
    print "Incorrect signature"

end = default_timer ()

print('Using 2048 bit DSA key signing and verification time in microsecond '+ str(end-start))

import hashlib
from Crypto.PublicKey import DSA
from Crypto.Signature import DSS
from timeit import default_timer

start = default_timer ()
key = DSA.generate(3072)
```

```
end = default_timer()
print('2048 bit DSA key generation time in microsecond '+ str(end-start))

sha256_hash = hashlib.sha256()

start = default_timer ()

with open("smallFile_1KB.txt","rb") as f:
   # Read and update hash string value in blocks of 4K
   for byte_block in iter(lambda: f.read(4096),b""):
      sha256_hash.update(byte_block)
   hash_obj = sha256_hash.hexdigest()

signer = DSS.new(key, 'fips-186-3')
signature = key.sign(hash_obj)

pub_key = key.publickey()

if pub_key.verify(hash_obj, signature):
   print "OK"
else:
   print "Incorrect signature"

end = default_timer ()

print('Using 2048 bit DSA key signing and verification time in microsecond '+ str(end-start))

sha256_hash = hashlib.sha256()

start = default_timer()

with open("largeFile_1KB.txt","rb") as f:
   # Read and update hash string value in blocks of 4K
   for byte_block in iter(lambda: f.read(4096),b""):
      sha256_hash.update(byte_block)
   hash_obj = sha256_hash.hexdigest()

signer = DSS.new(key, 'fips-186-3')
signature = key.sign(hash_obj)

pub_key = key.publickey()

if pub_key.verify(hash_obj, signature):
   print "OK"
else:
   print "Incorrect signature"

end = default_timer ()
```

```
print('Using 2048 bit DSA key signing and verification time in microsecond '+ str(end-start))
```

I run all of my codes in this VM image from https://cse.buffalo.edu/~eblanton/misc/vm/.

My comments regarding the performance aspects and observed results from my codes:

- For AES algorithm the speed is low for small file while the speed is higher for the large file. When I encrypt small file with 128-bit AES key in CBC mode the speed was **0.0010334518 byte/ns**, but when for the large file the speed was **0.01894505844 byte/ns**. So, large file is taking less time to encrypt a single byte. I observed a huge performance boost for large file. AES is good for large file encryption according to my results. In RSA algorithm the scenario is completely opposite. The speed reduces for large files. Using 2048 bit RSA key the encryption speed was **0.00131929779 bytes/ns** for small file but for large file the speed was **0.0002448054 bytes/ns**. So large file takes more time to encrypt a single byte. Thus it is not good for large file encryption from speed perspective.

- In **AES** algorithm decryption take less time than encryption for both small and large file. The difference is notable for **CBC** mode. In cipher-block chaining encryption must be done sequentially, while decryption can be *parallelized* as the XOR step (with the previous block of ciphertext) is done after the block cipher is applied. With 128-bit AES key in CBC mode the encryption time for small file is **967631 ns** while the decryption time is **353622 ns**. But in CTR mode encryption time for small file is **192661 ns** and decryption time is **147738 ns** which is not a notable difference like CBC mode. For **RSA** , decryption is slower than encryption; that's because both RSA encryption and decryption involve modular exponentiation, but whereas the public encryption exponent is normally small and fixed, the secret decryption exponent is usually almost as long as the modulus. Thus, doubling the modulus size makes encryption take twice as long, but makes decryption take *four times* as long. With RSA 2048-bit Key the encryption time for small file is **4084877104.62 ns** while the decryption time is **11215083103.2 ns**.

- For AES algorithm the 128 bit key generation time is **50576025 ns** but 256-bit key generation time is **162847 ns**. So the key generation time is reduced when the key length increases. I think there is some kind of setup cost for generating a key. That's why the time is larger for 128-bit key. For small file the encryption and decryption time is increases with the increase of key length. But for large file the encryption time is reduced with increase in key length.

  For RSA, the key generation time increases with the increase in key length. RSA 2048-bit Key Generation time is **227632900.78 ns** while RSA 3072-bit Key Generation time is **269256942.368 ns.** Also both encryption and decryption time increases if the key size is increased.

Problem 02

Solution:

1. **delete-all-rights** (s_1, s_2, o) :
   if (modify $\in A_i[s_1, o]$ and $own \notin A_i[s_2, o]$):
   $A_{i+1}[s_2, o] = \varnothing$

2. **copy-all-rights** (s_1, s_2, o) :
   $\forall x \in \{r, w, e, a, l, m\}$:
   if ($x^* \in A_i[s_1, o]$) :
   $\mathbf{A}_{i+1}[s_2, o] = A_i[s_2, o] \cup \{x\}$