Project 1 Report

Logan Connolly

**Java Code for parts 3, 4.c, and 4.d**

**Part 3:**

To read files of any given size, I read them in 8 byte increments, then encrypt and decrypt the 8 byte increment. Finally I read in the next 8 byte increment and repeat until the end of the file.

for (int i = 0; i < 8; i++) // reading the file 8 bytes at a time

{

if(next != -1)

{

text[i] = (byte)next;

next = infile.read();

}

else

{

i = 8;

}

}

To implement Cipher Block Chaining, the input had to be XORed with first the IV, and the other 8 byte increments were to be XORed with the previous blocks’ encrypted text.

byte[] xor = new byte[8];

for (int i = 0; i < 8; i++)

{

xor[i] = (byte)(text[i] ^ cbcchain[i]);

}

A variable called cbcchain was initialized with the IV, then the encrypted text was copied into the cbcchain.

To decrypt, first the encrypted text was run through the decrypting function, then XORed with the IV, which was loaded into the variable decryptchain. After, the original encrypted text was copied to the decryptchain.

byte[] textDecrypted = desCipher.doFinal(textEncrypted);

byte[] decryptxor = new byte[8];

for (int i = 0; i < 8; i++)

{

decryptxor[i] = (byte)(textDecrypted[i] ^ decryptchain[i]);

}

**Part 4.c:**

To read a file of an arbitrary size, I used the same method of reading in files as was used for the DES algorithm. A RandomAccessFile called infile pulled the file name that was listed as the third in the list of arguments. The program will read the file in 8 byte increments and encrypt/decrypt those increments as they are read.

RandomAccessFile infile = new RandomAccessFile(argv[2], "r");

byte[] input = new byte[8];

int next = infile.read();

while(next != -1)

{

for(int i = 0; i < 8; i++) // reading the file 8 bytes at a time

{

if (next != -1)

{

input[i] = (byte)next;

next = infile.read();

}

else

{

i = 8;

}

}

For timing, I called a variable called start, that would be incremented to indicate the start of the file reading. When start was <= 1, the start time would be record. When the file was out of bytes to read, the program would record the end time. Then the end time would be subtracted from the start time to get the time it took to encrypt the entire file and decrypt the entire file.

long millisEnStart = 0;

long millisEnEnd = 0;

long millisDeStart = 0;

long millisDeEnd = 0;

int start = 0;

if(start <= 1)

{

start++;

}

if(start <= 1)

{

millisEnStart = Calendar.getInstance().getTimeInMillis();

}

if(next != -1)

{

millisEnEnd = Calendar.getInstance().getTimeInMillis();

}

if(start <= 1)

{

millisDeStart = Calendar.getInstance().getTimeInMillis();

}

if(next != -1)

{

millisDeEnd = Calendar.getInstance().getTimeInMillis();

}

long totalEnTime = millisEnEnd - millisEnStart;

long totalDeTime = millisDeEnd - millisDeStart;

System.out.println("Time to Encrypt is: " + totalEnTime + " milliseconds.");

System.out.println("Time to Decrypt is: " + totalDeTime + " milliseconds.");

**Part 4.d**

As for RSA and DES, I made use of the RandomAccessFile to read in files of arbitrary length, but instead of reading 8 byte increments, the whole file is read in at once, then hashed as one byte array.

RandomAccessFile infile = new RandomAccessFile(argv[2], "r");

for(int i = 0; i < (int)infile.length(); i++)

{

if(next != -1)

{

input[i] = (byte)next;

next = infile.read();

}

else

{

i = (int)infile.length();

}

}

For timing, I took my method from RSA, but only the one start and end time are need as a hash cannot be undone, thus only the encrypting time is needed. The start and end times are recorded in a similar manner to what was done for RSA.

long millisStart;

long millisEnd;

millisStart = Calendar.getInstance().getTimeInMillis();

MessageDigest hash =

MessageDigest.getInstance("SHA-1", "BC");

hash.update(input);

byte[] digest = hash.digest();

millisEnd = Calendar.getInstance().getTimeInMillis();

long totalTime = millisEnd - millisStart;

System.out.println("Time to hash is: " + totalTime + " milliseconds.");

**Performance Measures**

**Comparison of DES and RSA Encryption**

Looking at the encryption timings for DES and RSA, the graphs both grow exponentially, but RSA takes longer on smaller files. The first significant increase in time is from 8 bytes to 16 bytes, while the first noticeable increase in time is between 4096 bytes and 32768 bytes.

**Comparison of DES and SHA1**

Looking at the timing differences between DES encryption and SHA1 hashing, SHA1 takes more time initially, but takes less on bigger file sizes. SHA1 takes around 20,000 microseconds to hash an 8 byte file while DES takes 12,000 microseconds to encrypt the same 8 bytes. When it comes to 262144 byte files, DES took 1,788,000 microseconds to encrypt while SHA1 took 28,000 microseconds to hash. So from these observations, DES is better encrypting smaller files while SHA1 is better at hashing bigger files, with respect to time.

**Comparison of RSA Encryption and Decryption**

Both RSA encryption and RSA decryption grow exponentially, but the encryption takes a small amount longer than the decryption. The difference in time is on average 4000 microseconds.

**Non-programing problems**

**a)** Since the function will output 0 no matter what, the only thing that a round of DES would do is swap the left and right sides of a 64 bit value. Thus 16 rounds of DES would result in the initial value and the 16th rounds value to be the same.

**b)** This is not a good message digest function as it will have many collisions and it will be easy to create a message with collisions.

Of the 6 requirements, the function only meets the first 3:

1. Can be applied to any size message

2. Produces a fixed length output h

3. Easy to compute h = H(m)

But it breaks the last 3:

4. Given h it is infeasible to find x H(y) = h

5. Given x is infeasible to find y H(y) = H(x)

6. It is infeasible to find any x, y H(y) = H(x)

**c)** Given the public key and the ciphered message, we must figure out the private to decrypt the message.

Since n = 35, p and q must be 5 and 7 respectively.

Then you can calculate φ(n) = (p-1)(q-1)=(5-1)(7-1)=4\*6=24

Next gcd(φ(n), e) = gcd(24, 5) =1, 1< e < φ(n)

Finally to find d, we do 5d mod 24 = 1, thus d = 5, so the private key is {5, 35}

Thus the deciphered message is 105 mod 35 = 5

To check we can do 55 mod 35 which equals 10, thus proving that the original plaintext message is 5.