

|  |
| --- |
| ***COMPUTR INFORMATION SYSTEM*** |
| ***CIS 5080 – Projects in Computer Information Systems*** |
|  |
| ***A Java Application for File Protection and Envelopes(Final Report)*** |
|  |
| ***Saleh Alsagoor PRESENTED TO DR. Steve Johnson*** |
| ***4/12/2009*** |
|  |

# Introduction

In this project, a Java application will be developed that performs encryption and decryption of files to ensure file security.

We have been encountering Internet Security and e-commerce in our day-to-day work. Unlike the time prior to the Internet and using Windows-NT on our computers (around the time DOS, Windows 95 and, Windows 97 ran our computers and Internet was less available), nowadays we log onto our University computers and are required to provide “passwords” for authenticating ourselves. We often carry out online travel reservations and bank transactions over the Internet. The security of these computers over a network has been always emphasized. We are often requested by our University’s Technical Support Group to keep changing our passwords. Password requirements are clearly specified on the University’s website [1]. Banks nowadays require us to provide “strong passwords” prior to allowing us to be able to transact online. In our homes, wireless Internet is used on our laptops and we tend to make them secure by enabling encryption and using MAC filters to restrict unauthorized users to misuse our wireless network.

## Motivation behind the Project

What has been under-discussed in the above examples is the need for the security of one’s own data. For example, we are always concerned about ensuring that no one eavesdrops on the wire when we transact online with a bank. But, we haven’t really thought about what would happen if someone were to get access to our files or our data. Very often we find our classmates and friends in our University having lost their USB memory drives. The contents of those drives could often be easily be seen by anyone who got hold of the same. If it contained Doctoral research material, that idea could be taken away by a competitor. If it contained documentation of a company’s trade-secret processes in a similar situation, a competitor could potentially ruin the company!

The problem of securing one’s own data using cryptographic techniques is addressed in the remainder of this section.

## An Overview of Cryptography

Cryptography is the science of concealing messages for the purposes of secure transmission to a recipient, or for secure storage on storage devices for later retrieval.

A typical model of an encryption scheme is generalized from [2]. Figure 1 gives the idea. A sender wishes to transmit a message to a recipient. The sender encrypts the message using an encryption key K1. This message is then transmitted over a transmission channel to the recipient. The recipient retrieves the message by decrypting the same with the help of a decryption key K2.

Encrypt

Encryption key, K1

**Sender**

Decrypt

Decryption key, K2

**Recipient**

Transmission

channel

Plaintext

Message

Plaintext

Message

**Figure 1. A Typical Encryption Model**

According to the figure, a cryptographic scheme consists of the following entities:

1. The message being secured, called *plaintext*.
2. The encrypted message, called *ciphertext*.
3. The *encryption key*.
4. The *decryption key*.
5. The *encryption algorithm*.
6. The *decryption algorithm*.

An assumption inherent in Figure 1 is that the encryption key K1 (decryption key K2) is available with the sender (recipient). How these keys are made available comes under the broad area of Key Management, which is outside the scope of this Project. In this Project, we always assume that keys to be used for transformation are already available.

One characteristic of cryptographic algorithms is its *key length*, measured in bits. The longer the key (more is its length), the stronger is the cryptographic algorithm. The strength of a cryptographic algorithm rests solely on its key length.

### Symmetric Key Cryptography

When the same key used for encryption is used for decryption, that is, when K1 = K2, the transformation process is called *Symmetric Key Cryptography*. In this case, it is assumed that the key is shared between the sender and recipient prior to initiation of an encryption / decryption transaction. Examples of symmetric key cryptographic algorithms are the Data Encryption Standard (DES) [3] and the newer Advanced Encryption Standard (AES) [4]. Symmetric key cryptographic algorithms involve use of logical and bit operations such as AND, OR, NOT, exclusive-OR, left-shift, right-shift and rotate operations. These operations execute very efficiently on computing machines, and also, are easily available as hardware devices. As such, these algorithms lend themselves more easily towards hardware implementation. The key sizes of these algorithms are typically around 64 to 256 bits. DES uses 64 bits while the AES uses 128 bits. Due to efficient executions on machines and short key lengths, they are suitable for bulk encryption such as in file and disk protection.

A symmetric key cryptographic algorithm also has a *block-length* associated with it. This is the basic unit of encryption. For example, DES encrypts a 64-bit block of data to yield a 64-bit output ciphertext at a time. The AES encrypts either 128, 192 or 256 bits of data at a time depending upon the user’s requirement and yields the corresponding size for the output ciphertext block. The question that will arise naturally is how such a scheme is used for bulk encryption of files having sizes of the order of Megabytes or Gigabytes. This is discussed in Section 1.2.3 below.

### Asymmetric (Public) Key Cryptography

When the encryption and decryption keys are not the same that is, when K1 ≠ K2, the transmission is called *Asymmetric Key Cryptography* (also called *Public* Key Cryptography).

One may imagine that for an asymmetric key cryptosystem, Figure 1 would assume the sender to possess the key K1 alone, and the recipient to possess the key K2 alone, but this is not how the system is typically used. Usually, a public key cryptosystem is employed instead in the following way.

Each user involved in the transaction possesses two keys. One of the two keys K1 and K2 is made public to the world and is known as the *public* key. The other key is secret, known only to the user who possesses the matching public key. This second key is known to no one else and is called the *private* key. A sender wishing to encrypt data brings up the recipient’s public key (known to the world, and to the sender), encrypts the data with this public key, and transmits to the recipient. Only the recipient will be able to decrypt the data using the private key that corresponds to this public key.

One may wonder that even when K1 ≠ K2, the keys K1 and K2 may be unrelated but that is not the case, and that K1 and K2 are indeed, related. It is this difficulty in determining their relationship that the security of asymmetric key cryptosystems rests upon.

Asymmetric key cryptographic algorithms involve the use of number-theoretic techniques such as modular arithmetic and in particular, modular exponentiation. They also use keys having lengths ranging from 1,024 bits and above. For military-grade security, a 4,096-bit key-length is encouraged. For commercial purposes, 1,024-bit keys are typically used. This suggests that number-theoretic computations cannot be realized using built-in operations available in programming languages. *Multiprecision* (also called large-integer) algorithms are used instead [5] and the challenge is to have highly efficient implementation of these algorithms available. Examples of asymmetric key algorithms are the Rivest-Shamir Adleman (RSA) algorithm [2], the El Gamal algorithm [6] and the Paillier algorithm [7]. Due to efficiency considerations and large-size keys involved, asymmetric key algorithms are not suitable for bulk data encryption. However, they are used for encrypting keys that are themselves used in symmetric-key algorithms. Keys used to encrypt other keys are often termed as *key encryption keys* (to differentiate from *message encryption keys*), to be discussed in Section 1.2.4. They are also used in Key Management to distribute and agree upon the symmetric key to be used by two parties in a transaction involving symmetric key cryptography.

Asymmetric key based schemes are also used in Digital Signatures and Certificates. These are outside the scope of the Project.

### Block Encryption of Bulk Data, and Last Block Padding

Usually bulk data to be encrypted using a symmetric-key algorithm is divided into a number of blocks. For example, a 1 MB file to be encrypted with the AES is divided into 128-bit blocks (16-byte blocks). The number of blocks is then equal to 1 x 1,024 x 1,024 bytes / 16 bytes = 65,536. Each of these blocks is encrypted to yield a 128-bit ciphertext which is concatenated to the output file.

What happens if the file-size is not a multiple of the block-size? In that case, the last block is incomplete. The last block is *padded* to complete. There is a standard available to do this padding, called the *Cryptographic Message Syntax* (CMS) [8]. When the file-size is a multiple of the block-size, as provided by the example in the previous paragraph, CMS padding is still followed to add an extra block. An outline of the CMS padding technique is provided in Section 2.1.3. In short, CMS padding adds information needed to examine the last block and stop at the correct point during decryption. At this time (during decryption) the last block will be *un-padded*.

### Message Envelopes

A way to solve to an extent, the problem in Figure 1, of ensuring that the shared key is already available with the sender and the recipient, is the following. The sender encrypts the plaintext with a *message encryption key* (MEK) and packages the ciphertext into a *message envelope*. The sender further adds the MEK into the envelope. This envelope is now sent to the recipient, possibly as an e-mail attachment or in a USB memory module.

However this presents a problem if the envelope gets into the hands of an unauthorized party. The intruder would retrieve the plaintext from the ciphertext using the MEK within the envelope.

To solve this problem, the sender will have to encrypt the MEK using the recipient’s public key, also called the *key encryption key* (KEK). The resulting ciphertext is called the *encrypted message encryption key* (EMEK). The EMEK is now bundled into the envelope along with the ciphertext message and sent to the recipient. Figure 2 suggests the layout of the message envelope. This structure conforms, in a sense, to the format provided in the Cryptographic Message Syntax (CMS) [9].

Encrypted Text

Encrypted Message Encryption Key

(EMEK)

Message Encryption Key (MEK)

Recipient’s Public Key (KEK)

**Figure 2. Layout of a Message Envelope. The MEK and KEK are only indicators for the locks in the figure and are not actually stored in the envelope**

In this format, the recipient, upon receipt of the envelope, would retrieve the MEK from the EMEK using the recipient’s private key corresponding to the public key (KEK). With the help of the (symmetric) MEK, the recipient would finally retrieve the plaintext message.

To thwart an intruder from directly using the recipient’s private key to retrieve the plaintext from the envelope, the recipient’s private key is usually stored on the storage device encrypted with a passphrase known only to the recipient and to no one else.

The structure of Figure 2 can be generalized to include multiple recipients. The MEK is encrypted using the public key of each recipient. Each of these resulting EMEK’s is stored in the envelope along with information for the recipient to identify the EMEK from among the several ones. Upon receipt of the envelope, the recipient uses the identifying information to select the particular EMEK to decrypt using the private key, and the process continues. The layout of such an envelope and its implementation is beyond the scope of the current Project.

# A Proposed Solution to the Problem of Securing Files

The problem of file protection is envisaged to be of two kinds.

# Securing a file for later retrieval

# Securing a file for transmission to a recipient – possibly as an e-mail attachment – and retrieval of the file by the recipient

Each of the above is discussed now. Thereafter, the popular algorithms to be used for our solution will be outlined in the rest of this section.

## Securing a file for later retrieval

# To secure a file, one encrypts the file with the help of an encryption key – typically a passphrase – that only the person knows. No one else will be able to retrieve the contents of the file without knowing the passphrase. The following are the steps performed to secure a file.

### Algorithm: File Encryption

Inputs:

* The file to be secured
* A 16-character passphrase to be used to secure the file

Outputs:

* The encrypted (secured) file

Steps:

1. Divide the input file into a number of blocks. Each block will be of size 128 bits (that is, 16 bytes) long.
2. Encrypt each block using the Advanced Encryption Standard (AES) algorithm using the 128-bit (that is, a 16-byte) passphrase as the encryption key. This block will also be 128 bits long. If the file-size is not a multiple of the block-size, then the last block should be padded. Padding is done using the Cryptographic Message Syntax (CMS). (See Section 2.1.3.)
3. Append each encrypted block to the output file.

### Algorithm: File Decryption

Inputs:

* The encrypted file
* A 16-character passphrase to be used to decrypt the file

Outputs:

* The decrypted (plaintext) file

Steps:

1. Divide the input (encrypted) file into a number of blocks. Each block will be of size 128 bits (that is, 16 bytes) long.
2. Decrypt each block using the Advanced Encryption Standard (AES) algorithm using the 128-bit (that is, a 16-byte) passphrase as the encryption key. This block will also be 128 bits long. If the file-size is not a multiple of the block-size, then the last block should be unpadded. Un-padding is done using the Cryptographic Message Syntax (CMS). (See Section 2.1.3.)
3. Append each encrypted block to the output file to form the plaintext.

### Last-block padding during Encryption – An Overview of CMS Padding

Suppose the input file is of size *F* bytes and the block-size is *k* bytes. The input file is divided into blocks of size *k* bytes each. Each block will be encrypted to yield a ciphertext block of size *k* bytes which is added to the output file. If *F* is not a multiple of *k*, the last block size will be (*F* – *k* × ⎣*F*/*k*⎦ ) bytes, clearly ranging between 0 and (k-1) bytes, that is, the last block will be partially filled. The remaining (k – F + k × ⎣*F*/*k*⎦) bytes will be padded, each byte containing a value equal to (k – F + k × ⎣*F*/*k*⎦).

***Example 1*:**

Suppose an input file is 130 bytes long, to be encrypted with AES having a block-size of 128 bits (16 bytes). Since 130 is not a multiple of 16, the last block will contain only 2 bytes filled, and the remaining (16 – 2) = 14 bytes empty. According to CMS, these 14 empty slots will be filled with the number 14. Note that the size of the output ciphertext file will be (130+14) = 144 bytes.

If the input file size *F* is a multiple of the block-size *k*, then, the last block will be completely filled and the number (*F* – *k* × ⎣*F*/*k*⎦ ) will equal zero. In this case, an extra block of block-size *k* will be added, and will be filled with the number *k*.

***Example 2*:**

Suppose an input file is 128 bytes long, to be encrypted with AES having a block-size of 128 bits (16 bytes). Since 128 is a multiple of 16, the last block will contain all 16 bytes filled, and the remaining (16 – 16) = 0 bytes empty. According to CMS, an extra block will now be added and filled with the number 16 (equal to the block-size). Note that the size of the output ciphertext file will be (128+16) = 144 bytes.

In general, the size of the encrypted file

= k × ⎡F/k⎤, if F is a not multiple of k

= k × (1 + F/k), if F is a multiple of k

The reason for this kind of padding being done is that during decryption, upon encountering the last block, one can determine how many bytes to un-pad (indicating that the underlying plaintext file size is either a multiple of the block-size or not). Accordingly one could also determine the size of the underlying plaintext file but is not necessary.

## Securing a file for transmission to a recipient

A sender wishing to send a file securely to a recipient will create a message envelope like the one in Figure 2. The envelope will contain the file encrypted with the MEK, and the MEK encrypted with the recipient’s public key (the KEK) forming the EMEK. The recipient, upon receiving the envelope, will supply the passphrase to retrieve the private key and decrypt the EMEK with the now-retrieved private key to obtain the MEK. Finally, the recipient will decrypt the encrypted message with the now-retrieved MEK to obtain the plaintext message.

### Algorithm: Securing File in Message Envelope

Inputs:

* The file to be secured into a message envelope
* A 16-character passphrase to be used to encrypt the file
* The recipient’s public key, stored in a public key file

Outputs:

* The message envelope, as an output file

Steps:

1. Select file to be secured into message envelope
2. Input a 16-character passphrase.
3. Input recipient’s public key stored in a public key file. (This file is generated by Algorithm 2.3.2.)
4. With the help of the algorithm in Section 2.1.1, encrypt the file using the passphrase to yield an encrypted file. Append the encrypted file into an empty output file.
5. Using the RSA algorithm, encrypt the passphrase to yield the EMEK. Append the EMEK into the output file. This output file is now the message envelope.

### Algorithm: File Retrieval from Envelope

Inputs:

* The message envelope from which the file is to be retrieved
* The recipient’s private key file, encrypted using a 16-character passphrase
* A16-character passphrase to be used to retrieve the recipient’s private key

Outputs:

* The retrieved plaintext file, as an output file

Steps:

1. Select message envelope to be processed.
2. Input the recipient’s private key corresponding to the recipient’s public key, stored in a private key file. (This private key is stored encrypted, protected by a 16-character passphrase, and is generated by the algorithm of Section 2.3.4).
3. Using the AES algorithm and the recipient’s 16-byte passphrase as the decryption key, decrypt the protected private key to yield the private key.
4. Retrieve the EMEK from the message envelope.
5. Using the RSA algorithm, decrypt the EMEK with the private key to yield the MEK.
6. Retrieve the encrypted file from the message envelope.
7. With the help of the algorithm of Section 2.1.2, decrypt the encrypted file using the MEK to yield the decrypted file.

## Algorithms

Contemporary well-known algorithms available in the literature, namely, the Advanced Encryption Standard (AES) algorithm [4] and the Rivest-Shamir-Adleman (RSA) algorithm [2] will be used in our implementation. The AES algorithm being used in the Project will have a key-length of 128 bits (16 bytes) and a block-size of 128 bits (16 bytes). The RSA algorithm being used in the Project will have a key-length of 1,024 bits.

### Locations of Cryptographic Algorithms in our Project

The different points at which these algorithms are directly used in all algorithms of Sections 2.1 and 2.2, are the following:

1. Step 2 of the algorithm of Section 2.1.1 uses the AES block-encryption algorithm with a 128-bit key.
2. Step 2 of the algorithm of Section 2.1.2 uses the AES block-decryption algorithm with a 128-bit key.
3. Step 5 of the algorithm of Section 2.2.1 uses the RSA algorithm to encrypt the 128-bit MEK.
4. Step 3 of the algorithm of Section 2.2.2 uses the AES algorithm to decrypt the recipient’s encrypted RSA private key.
5. Step 5 of the algorithm of Section 2.2.2 uses the RSA algorithm to decrypt the EMEK.

The individual cryptographic algorithms, particularly the AES, are too fine-grained to be directly represented at a Project Proposal level. The full specification for the AES algorithm is available in references [2], [4] and for RSA, the algorithm is discussed along with proofs of its working in references [2], [5] and [6]. Only an overview at a very high level is provided for both algorithms in this section.

### An Overview of the Advanced Encryption Standard (AES)

Figure 3 depicts the block diagram of the AES. AES takes in a 128-bit block of data and gives out a 128-bit encrypted block. The input block is subjected to 10 *rounds*. Figure 4 depicts one round of encryption.

A *state* is a 128-bit block that moves as the output of a previous round (*r* – 1) to the input of the next round *r*, where 1 ≤ *r* ≤ 10. Each state is organized into a 4-element array of 32-bit *words*.

The 128-bit encryption key is also expanded into a 44-word array. In the encryption process, words w0,w1,w2,w3 are used in Round 1. Words w4, w5, w6 and w7 are used in Round 2. In general, w4r, w4r+1,w4r+2 and w4r+3 are used in Round *r*, where 1 ≤ *r* ≤ 10.

Round 1

Round 2

Round 10

Add Round Key

Plaintext Block

(128 bits)

Ciphertext Block (128 bits)

Round 10

Round 9

Round 1

Add Round Key

Plaintext Block

(128 bits)

Ciphertext Block (128 bits)

Key

Expand Key

w0..w3

w4..w7

w8..w11

w40..w43

**Encryption**

**Decryption**

**Figure 3. Block Diagram of the AES Algorithm.**

Figure 3 also shows the decryption process. Decryption uses the expanded keys in the reverse order.

Figure 4 shows the internals of one round. An encryption round consists of the following operations: Substitute Bytes, Shift Rows, Mix Columns, and Add Round Key. The last operation “Add Round Key” is also shown separately in Figure 3 applied to the plaintext block before input to Round 1. Among all these operations, the expanded key is added (exclusively-ORed) into the state only in the operation “Add Round Key”. The operation “Substitute Bytes” applies a fixed set of substitution boxes (S-Boxes) to a state. The operation “Shift Row” visualizes the state array as a 4 × 4 matrix of 8-bit words each (with each column as a 32-bit word array), and shifts the rows of the array in a circular manner. The operation “Mix Column” multiplies the matrix with a known polynomial modulo another.

Substitute Bytes

Shift Rows

Mix Column

Add Round Key

State (128 bits)

State (128 bits)

Inverse Mix Column

Add Round Key

Inverse Subst Bytes

Inverse Shift Row

State (128 bits)

State (128 bits)

**Encryption**

**Decryption**

Expand-ed Key

**Figure 4. One AES Round *r*. A State is the output of a previous round (*r* – 1) of Figure 3, being input to the current round *r*, 1 ≤ *r* ≤ 10. There is no Mix Column and Inverse Mix Column operation when r = 10.**

In the decryption process shown in Figure 4, corresponding to each operation, there is an “inverse” operation. For example, the operation “Inverse Mix Column” multiplies the state-matrix by the inverse of the matrix used to multiply in “Mix Column” during encryption. The operation “Inverse Shift Row” is the reverse of the operation “Shift Row”, while the operation “Inverse Subst Bytes” performs an inverse lookup of S-Boxes. The only operation that is not mentioned with the “Inverse” is the “Add Round Key” since this exclusive-OR operation is the same as its inverse.

Further details of these operations and key expansion are discussed in [2], [4].

### An Overview of the Rivest-Shamir-Adleman (RSA) Algorithm

The RSA algorithm works on Number Theory and consists of three parts:

1. Public-Private Key-pair Generation
2. Encryption, and
3. Decryption

These are now discussed. Encryption and Decryption are discussed first, and the generation of key-pairs is dealt with separately in Section 2.3.4.

In our Project, RSA encryption and decryption will not be done on files. They will be done only on memory variables since the MEK is not to be stored in files.

#### Algorithm: RSA Encryption

Inputs:

* The recipient’s public key, a pair of integers (*e, n)*, stored in a file
* The text to encrypt (the MEK).

Outputs:

* The ciphertext (the EMEK).

Steps:

1. Read the public key file and retrieve the integers *e* and *n*.
2. Convert the 128-bit text into an integer *x* using base-256 conversion. Since the length of *x* in bits is at most 128 bits while the public key – the size of *n* – is 1,024 bits, clearly, 0 ≤ *x* < *n*.
3. Set y = xe (mod n).
4. Convert *y* to a bit-stream using base-256 conversion to yield the ciphertext. The size of *y* will be at most 1,024 bits long.

#### Algorithm: RSA Decryption

Inputs:

* The recipient’s private key, a pair of integers (*d, n)*. It is assumed that this private key is already decrypted using a passphrase.
* The ciphertext to decrypt (the EMEK).

Outputs:

* The plaintext (the MEK).

Steps:

1. Read the private key and obtain the integers *d* and *n*.
2. Convert the 1,024-bit text into an integer *y* using base-256 conversion. Since the length of *y* in bits is at most 1,024 bits while the private key – the size of *n* – is 1,024 bits, clearly, 0 ≤ *y* < *n*.
3. Set x = yd (mod n).
4. Convert *x* to a bit-stream using base-256 conversion to yield the plaintext.

### Algorithm : RSA Key-pair Generation

Inputs:

* A passphrase to protect the private key

Outputs:

* The public key consisting of two integers (*e*, *n*), stored in a file
* The private key consisting of two integers (*d*, *n*) stored in a file *encrypted using the passphrase*.

Steps:

1. Randomly generate two *distinct* prime numbers *p* and *q*, each of size 512 bits ensuring that the two most significant bits of each of these are equal to **11** (binary).
2. Compute *n* = *p* × *q*. Note that *n* will be 1,024 bits long, and is called the *modulus*..
3. Compute φ(*n*) = (*p* – 1) × (*q* – 1)
4. Choose *e* such that *e* is relatively prime to φ(*n*).
5. Compute *d* such that (*e* × *d* – 1), when divided by φ(*n*), leaves a remainder of 1.
6. The public key is (*e*, *n*). Store these in a public key output file.
7. The private key is (*d*, *n*). Encrypt this pair using the algorithm of Section 2.1.1 into an output file using the passphrase as the encryption key.

# Implementation

The Project is been implemented using the Java programming language. Java version 1.6 is used.

The RSA part of the Project is implemented using the Java class BigInteger. The advantage in using Java is that this class is already part of Java. Thus all multiprecision integer processing is already taken care of. The other advantage of using Java is that no effort for porting to other platforms is required.

Development is carried out on Linux.

Files to be encrypted are either Text Files or Binary Files. After decryption, the file is usable. For example, if the binary file is a video file, then after decryption, the video is playable. The maximum size of a is typically 4 GB.

The hardware used is an x86-based personal computer. No special software has been required.

# Project Schedule – The Gantt Chart

The Gantt chart for the Project Schedule is depicted below.

08-Feb

**Date**

**Activity**

1

15-Feb

2

22-Feb

01-Mar

25-Jan

3

4

08-Mar

15-Mar

5

Legend:

1 – AES Block Encryption & Decryption

2 – File Encryption & Decryption

3 – RSA Key-pair Gen, Encryption & Decryption

4 – Message Envelope Creation & Message Retrieval

5 – A GUI Application

22-Mar

01-Feb

# Results

This section discusses the experiences while implementing the solution. Problems and challenges encountered are outlined. Reasons for the way the solutions have been chosen among several alternatives, are discussed. Finally, the lessons from this project are mentioned.

## Implementation Experience

The best part of the Project has been the focused manner in which the tasks that were outlined in the schedule were implemented. The tasks were usually completed on time. There were challenges that surfaced during the implementation. These are outlined below.

### Buffered Reading / Writing of Binary files for Encryption / Decryption

The files to be protected need not always be text files. They could be multimedia streaming files, video files, or even binary images. Java processes these as binary files. The challenges experienced in Java were how to manipulate binary files since we had not studied binary file-processing during our Java course.

These files had to be read (written) using *buffered* reading (writing). Byte-reading (writing) was possible, reading (writing) one byte at a time, but this was grossly inefficient since every byte had to be accessed from (written to) secondary storage resulting in too many disk-head movements and a consequent slowdown in speed. In buffered reading (writing), files are divided into blocks and each block, read from (written into) a buffer, the amount of disk-head movements reduces significantly, and the read (write) operation is more efficient. The challenge in Java was how to use buffered reading (writing) of binary files.

Once this challenge was overcome, encryption and decryption did not pose significant problems.

### Test Vectors stored as Files

The Advanced Encryption Standard (AES) document specifies exactly what the values of the plaintext and expected ciphertext are. However, it does not specify how they should be stored. Any programmer may readily assume that the plaintext should be stored into a memory variable, encrypted, and the resulting ciphertext stored into memory, to compare with existing ciphertext also stored in memory. However this need not always be the case, and the plaintext and expected ciphertext may be stored as separate text files. Our application performs file encryption. It generates an output file. Our application should be able to encrypt a test vector plaintext stored as a *file* to yield a ciphertext in a *file*. The output ciphertext file should be file-compared with the expected test vector, also saved as a file from the AES document. This should happen with every test vector in the standard.

The problem was how to create binary files. A binary visual-editor, namely, **bvim**, (based on the **vi**–editor) had to be downloaded and installed for the purpose of creating and editing binary test-vector files. Once this challenge was overcome, encryption and decryption and cross-checking with test vectors that were saved as files was successful with no extra effort. Luckily the program did not pose a problem.

### Decrypted files should be usable

For example, if a JPEG file was encrypted and then decrypted back, the JPEG file should still display the image properly just as the original file. A video movie-file that was encrypted and decrypted back should be playable, with no slow-motion in speed. Simply doing a file-compare with the original plaintext and decrypted text is not sufficient.

Our application performs this job without any extra effort.

### The BigInteger Challenge while implementing the RSA Algorithm

The **BigInteger** class provided by Java handles numbers in two’s-complement form. When converting to byte form, it sometimes appends a zero byte at the most significant position. When writing to disk, these zeros appear. Upon reading from the disk, these zeroes cannot be ignored otherwise the methods of the **BigInteger** class raises an exception. This causes problems when the number of bytes of the modulus, public key exponent and private key exponent are expected to be 128, but sometimes they show up as 129 bytes.

The problem is solved by designing the public and private key file formats to also include the byte-lengths of these numbers.

### The BigInteger Challenge while implementing Message Envelopes

The **BigInteger** class provided by Java handles numbers in two’s-complement form. As a result, whenever the most significant bit (MSB) is **1**, the number is interpreted as a negative number. We are required to encrypt the Message Encryption Key (MEK) with the recipient’s public key using RSA. However, this MEK can be any random byte-sequence, meaning, it could have **1** at its MSB position. To ensure that this is not misinterpreted by the RSA module, an extra byte should be padded. This extra byte should be greater than zero, and also, not have **1** at its MSB position. Any number between 1 (decimal) and 127 (decimal) should work.

Having discovered this fact, a one-byte padding string has been designed with a value of **01** (having neither **1** at its MSB position, nor all zeros in this byte) to be added to the start of the MEK prior to RSA encryption.

### GUI Development Challenges

There are two challenges being faced in this task, namely, using the class **JDialog** to implement dialogs, and using the class **JFileChooser** to select files.

The reason for the difficulty in using these classes is the lack of availability of relevant material in Java textbooks (the one by Deitel is being referred).

As a result, a trial-and-error approach is made to use these classes, iteratively improving upon the GUI after learning from errors encountered.

## Reasons for Selection of Different Parts of the Solution

The part of the application targeted for a discussion of different solutions is the encryption and decryption algorithms, along with their key-sizes.

### Symmetric Key Algorithms

Several algorithms are available in the literature [2,3,4,5] – Data Encryption Standard (DES), International Data Encryption Algorithm (IDEA), CAST, Blowfish, MARS, RC-6, Rijndael (including the AES), Serpent and Twofish. Variants such as Triple-DES and Twofish-3-fish were also available.

However we have selected the Advanced Encryption Standard, because:

* It is the latest definitive standard, and will continue to be so for at least another 50 years
* It is current
* It has been subjected to strong cryptanalytic attacks by the Cryptography community worldwide
* It has a well-specified document [4]. The implementation of the algorithm was easy for this one important reason.
* Test vectors were available and testing was simplified. One did not have to exercise extra effort in searching for test vectors.

### Asymmetric Key Algorithms

Several algorithms were available, such as El Gamal and Paillier. However the RSA algorithm is selected, since:

* The RSA algorithm has been widely-used since the last thirty years
* The other algorithms, although reasonably well-known, pose efficiency issues in implementation. The numbers involved are 1,024-bit integers, and the modular exponentiation is a high-intensity operation in most of these algorithms.
* An enhancement would be to digitally sign message envelopes. For this, a digital signature algorithm will need to be implemented. RSA does this straightforwardly since the public and private keys are interchangeable. This is not true of the other algorithms.

## Lessons learned

The important, technical lessons learned are the following, particularly while handling the BigInteger class of JAVA, handling random numbers, and implementing GUI’s.

* Never convert and store a **BigInteger** as it is into a text file. There is no guarantee that upon reading back from the file, the size would be consistent. This is particularly true when one stores multiple **BigInteger** numbers as strings of numbers in the file, together. (For example, the RSA public and private keys always consists of *pairs* of integers.) Always specify the byte-length of the converted BigInteger *in addition* to the byte-form in the file. Ensure that this length information is read first. That way one will know how many more bytes to read from.
* Whenever a random number is going to be read into a BigInteger, ensure that it *does not* contain a **1** as its most significant bit. Otherwise the results become unpredictable. Always ensure that this random number has an extra byte at its leftmost position, with **0** as the most significant bit of this byte.
* When a random number is seeded, ensure that this seeding is done at only one place in the entire program. There are several places where random numbers are used. An example is, to generate the factors of the RSA modulus. Ensure that the random number is seeded only once.
* When one does not know how to use an object while implementing a GUI, the best strategy is to read a book or an Internet site. Focus on the first few pages, implement the GUI, test it, and keep modifying the GUI. Look up more pages if necessary. This saves time, and is a smart approach. This was how **JFileChooser** and **JDialog** classes were used, by trial-and-error.
* When a random number is seeded, ensure that this seeding is done at only one place in the entire program. There are several places where random numbers are used. An example is, to generate the factors of the RSA modulus. Ensure that the random number is seeded only once.

# Enhancements

There are a number of ways in which the project can be improved upon. Some of these are discussed in this section.

## Message Digests and Signed Message Envelopes

One way of enhancing security is to affix a digital signature to the message envelope format. A digital signature is performed when the sender of a message envelope uses the private key to sign the message.

The message is typically a file which could go well up to 4 GB. Performing a private-key operation involving 1,024-bit arithmetic on such large files is very inefficient. Therefore the file itself is not signed. A *message digest* of the file is instead, signed. This digest is obtained from the file by, loosely speaking, *compressing* the file. The size of a message digest ranges from 128-bit (16 bytes) up to 512-bit (64 bytes). Well-known message digest algorithms are the Secure Hash Algorithm (SHA) that generates 224-bit digests, 256-bit digests, 384-bit digests and 512-bit digests [11]. Other digest algorithms are the RIPEMD-160 (160-bit algorithm) and H-MAC (Message Authentication Code) [12] algorithms.

The recipient, upon receiving the signed envelope, will verify the signature by recomputing the digest of the message and using the sender’s public key on this digest, prior to retrieving the message from the envelope.

The objective of digital signatures is non-repudiation. A sender who has signed a message can never deny not having sent the message earlier, since there is proof of the sender’s signature.

## Elimination of Temporary Files

In our application, the private key is stored encrypted onto the hard disk. However, there exists a moment in time when the private key is stored as a temporary file on the hard disk *in cleartext format* before the passphrase is provided for encryption. This is a serious threat.

To eliminate this loophole, the *memory* holding the private key should be encrypted. In other words, encryption and decryption of memory buffers to yield output memory buffers could be made available as part of the application to achieve the objective of eliminating temporary files.

## Modular Exponentiation

In the RSA algorithm, the primary operation involves modular exponentiation. This operation is performed using a naïve approach in our implementation. The technique can be significantly optimized by implementing the Montgomery multiplication and Montgomery exponentiation that uses this multiplication operation [10]. In the naïve approach, a division step is involved to determine the product modulo the modulus. This division step is eliminated and is reduced to another form of multiplication modulo the modular inverse, which is done only once rather than at every step, resulting in significant speed-up.

# References

1. *Florida Institute of Technology Technical Support - About Your Florida Tech Email* (“How do I change my password?”), Author Unknown, <http://www.fit.edu/webservices/email.htm>, Year Unknown.
2. *Cryptography and Network Security*, William Stallings, 3rd edition, Pearson Education, Inc., 2003
3. *Data Encryption Standard (DES)*, Federal Information Processing Standard (FIPS) 46-2, Author Unknown, <http://www.itl.nist.gov/fipspubs/fip46-2.htm>, 1988
4. *Advanced Encryption Standard (AES)*, Federal Information Processing Standard (FIPS) 197, Author Unknown, <http://.csrc.nist.gov/fips/fips197/fips-197.pdf>, 2001
5. *Handbook of Applied Cryptography*, Alfred J.Menezes and Paul C. van Oorschot and A.Scott, <http://www.cacr.math.uwaterloo.ca/hac>, 1996
6. Applied Cryptography – Protocols, Algorithms and Source Code in C, Bruce Schneier, 2002.
7. *Public-Key Cryptosystems based on Composite Degree Residuosity Classes*, Pascal Paillier, Eurocrypt’99, pp. 223 – 238, 1999.
8. *Cryptographic Message Syntax (CMS)*, RFC 3852, R.Housley, <http://tools.ietf.org/html/rfc3852>, Section 6.3, July 2004.
9. *Cryptographic Message Syntax (CMS)*, RFC 3852, R.Housley, <http://tools.ietf.org/html/rfc3852>, Section 6.1, July 2004.
10. *Modular Multiplication without Trial Division*, P.L.Montgomery, Mathematics of Computation, vol. 44, pp. 519-521, 1985
11. *RIPEMD-160, a strengthened version of RIPEMD*, Fast Software Encryption Lecture Notes in Computer Science 1039, H. Dobbertin, A. Bosselaers and B. Preneel, pp. 71 – 82, 1996
12. *H-MAC: Keyed Hashing for Message Authentication*, RFC 2104, H.Krawczyk, M.Bellare and R.Canetti, <http://www.ietf.org/rfc/rfc2104.txt>, February 1997.

**APPENDIX A**

**A.1 Status Reports**

**Capstone Project Week 5 Status Report for “A JAVA Application for File Protection and Envelopes”**

**Date:** 07-Feb-2009

**Accomplishments**

Activity 1: AES Block Encryption and Decryption (25-Jan-2009 – 08-Feb-2009)

1. Key Expansion – completed
2. Byte Substitution and Inverse Byte Substitution – completed
3. Shift Row and Inverse Shift Row – completed
4. Mix Column and Inverse Mix Column – completed
5. Add Round Key – completed
6. Block Encryption and Decryption – completed
7. File Encryption and Decryption – This is Activity 2 (incorrectly mentioned here)
8. Test Vector Testing – completed
9. Unit Testing – completed

**Current Activities (08-Feb-2009 – 15-Feb-2009)**

Activity 2: File Encryption and Decryption

1. File Encryption

**Challenges**

The challenges are:

1. How to perform binary (non-text) input/output on files in Java, using buffered reading
2. Ensure that the test vector testing (Sub-activity 8 in the section on Accomplishments above) also works on files, particularly binary files
3. Ensure that after decryption, the binary file is usable. For example, if it is a video file, that file should be re-playable. If the file is a JPEG file, the image should be viewable.

None of these areas are envisaged to cause any problems.

**Work to be Completed by 15-Feb-2009**

1. File Decryption
2. Unit Testing of all of the above

**Capstone Project Week 7 Status Report for “A JAVA Application for File Protection and Envelopes”**

**Date:** 19-Feb-2009

**Accomplishments**

Activity 2: File Encryption and Decryption

1. File Encryption – Completed
2. File Decryption – Completed
3. Unit Testing of all of the above – Completed

Activity 3: RSA

1. RSA Key-pair Generation – Completed
2. RSA Encryption – Completed
3. RSA Decryption – Completed

**Current Activities (15-Feb-2009 – 01-Mar-2009)**

Activity 3: RSA

1. Unit Testing of all of the above – In Progress

**Challenges**

The **BigInteger** class provided by Java handles numbers in two’s-complement form. When converting to byte form, it sometimes appends a zero byte at the most significant position. When writing to disk, these zeros appear. Upon reading from the disk, these zeroes cannot be ignored otherwise the methods of the **BigInteger** class raises an exception. This causes problems when the number of bytes of the modulus, public key exponent and private key exponent are expected to be 128, but sometimes they show up as 129. The problem is solved by designing the public and private key file formats to also include the byte-lengths of these numbers.

Having discovered the correct way to manipulate binary files and **BigInteger** objects, no new problems are now envisaged.

**Work to be completed by 01-Mar-2009**

Activity 3: RSA

1. Unit Testing

**Capstone Project Week 9 Status Report for “A JAVA Application for File Protection and Envelopes”**

**Date:** 06-March-2009

**Accomplishments**

Activity 4: Message Envelope Creation and Message Retrieval

1. Design of the Message Envelope Format and updating into the Report – Completed
2. Message Envelope Generation from input file – Completed
3. File Retrieval from Message Envelope – Completed

**Current Activities (01-Mar-2009 – 15-Mar-2009)**

Activity 4: Message Envelope Creation and Message Retrieval

1. Unit Testing of all of the above – In Progress

**Challenges**

The **BigInteger** class provided by Java handles numbers in two’s-complement form. As a result, whenever the most significant bit (MSB) is **1**, the number is interpreted as a negative number. We are required to encrypt the Message Encryption Key (MEK) with the recipient’s public key using RSA. However, this MEK can be any random byte-sequence, meaning, it could have **1** at its MSB position. To ensure that this is not misinterpreted by the RSA module, an extra byte should be padded. This extra byte should be greater than zero, and also, not have **1** at its MSB position.

Having discovered this fact, we have designed a one-byte padding string with a value of **01** (having neither **1** at its MSB position, nor all zeros in this byte) to be added to the start of the MEK prior to RSA encryption. No new problems are now envisaged.

**Work to be completed by 15-Mar-2009**

Activity 4: Message Envelope Creation and Message Retrieval

1. Unit Testing of all of the above – In Progress

**Capstone Project Week 11 Status Report for “A JAVA Application for File Protection and Envelopes”**

**Date:** 20-March-2009

**Accomplishments**

Activity 4: Message Envelope Creation and Message Retrieval

1. Unit Testing of all of the above – Completed

**Current Activities (16-Mar-2009 – 22-Mar-2009)**

Activity 5: A GUI Application

1. Development of the GUI – In Progress

**Challenges**

There are two challenges being faced in this task.

1. Using the class **JDialog** to implement dialogs.
2. Using the class **JFileChooser** to select files.

The reason for the difficulty in using these classes is the lack of availability of relevant material in Java textbooks (the one by Deitel is being referred). As a result, a trial-and-error approach is made to use these classes, iteratively improving upon the GUI after learning from errors encountered. These challenges are gradually being overcome.

**Work to be completed by 22-Mar-2009**

Activity 4: Message Envelope Creation and Message Retrieval

1. Integration and Final Testing

**A.2 Software Installation**

**Requirements**

It is assumed that Java 1.5.0 (also called Java 5.0) and above are installed on the target machine and the paths, appropriately set.

**Availability of the programs**

The programs are available in a ZIP file, **Capstone.zip**

**Source Files**

There are four **.java** programs:

1. Aes.java
2. FileProtect.java
3. Rsa.java
4. MsgEnvelope.java

The main program is available in the file **FileProtect.java**.

**Installation Steps**

On both, Linux and Windows, the installation is carried out by following the steps below.

1. Extract the source files
2. Compile the source files
3. Execute the application

**Installation Steps – Linux**

**Step 1: Extract the source files**

In Linux / UNIX, assume that you are at the following prompt:

**bash$**

Next, copy (or download and save) the **Capstone.zip** file into a suitable directory. After doing this, the file should be unzipped as follows:

**bash$ unzip Capstone.zip**

Once the file is unzipped, you will see the following contents in your directory.

**bash$ ls**

**Aes.java Capstone.zip FileProtect.form FileProtect.java MsgEnvelope.java Rsa.java**

**bash$**

**Step 2: Compile the source files**

Next, the source files should be compiled using javac.

**bash$ javac Aes.java**

**bash$ javac Rsa.java**

**bash$ javac MsgEnvelope.java**

**bash$ javac FileProtect.java**

**bash$**

Your files are now compiled and ready for use.

**Installation Steps – Windows**

In Windows, assume that you are at the DOS – prompt:

**C:\>**

**Step 1: Extract the source files**

Next, copy (or download and save) the **Capstone.zip** file into a suitable folder, say, **MyFolder**, in Drive C:. After doing this, the file should be unzipped (using, for example, WinZip). Once the file is unzipped, you will see the following contents in your folder.

**C:\MyFolder> dir**

**Volume in drive C is salsagoo**

**Volume Serial Number is 4752-258E**

**Directory of C:\MyFolder**

**04/10/2009 12:20 AM <DIR> .**

**03/22/2009 12:37 AM <DIR> ..**

**03/02/2009 10:54 PM 30,338 Aes.java**

**04/01/2009 10:00 AM 20,773 Capstone.zip**

**03/22/2009 12:35 AM 7,114 FileProtect.form**

**03/22/2009 12:48 AM 23,018 FileProtect.java**

**03/06/2009 12:03 AM 7,637 MsgEnvelope.java**

**02/18/2009 09:44 PM 22,911 Rsa.java**

**9 File(s) 117,605 bytes**

**2 Dir(s) 96,066,338,816 bytes free**

**C:\MyFolder>**

**Step 2: Compile the source files**

Next, the source files should be compiled using javac.

**C:\MyFolder> javac Aes.java**

**C:\MyFolder> javac Rsa.java**

**C:\MyFolder> javac MsgEnvelope.java**

**C:\MyFolder> javac FileProtect.java**

**C:\MyFolder>**

Your files are now compiled and ready for use.

**A.3 User Manual**

The system is assumed to run on the Linux operating system. The screens are identical on the Windows operating system.

The directory is assumed to be the location where the class files are available after installation.

**A.3.1 Executing the Application**

To execute the application, type the following command:

**bash$ java FileProtect**

Immediately the following menu appears:



The following is the sequence in which the application will be discussed:

1. Secure Storage
2. Key Management, and
3. Secure Transmission

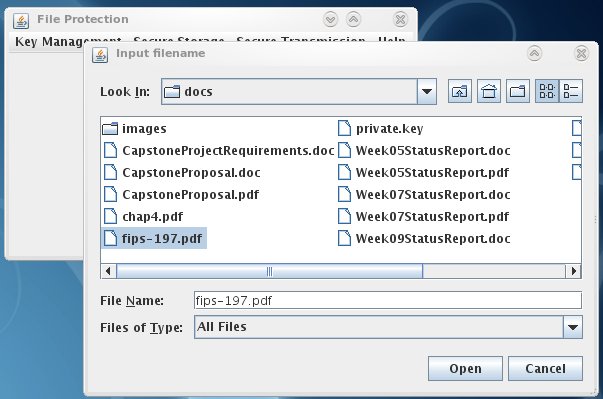
**A.3.2 Secure Storage**

This part of the application secures a file on disk with a 16-character passphrase. It consists of two parts: File Protect and File Unprotect.

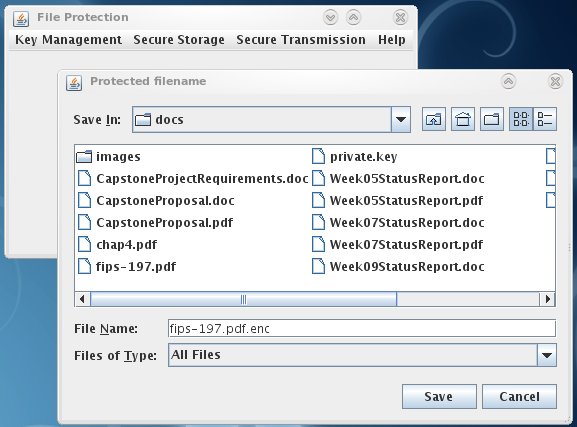
**A.3.2.1 File Protect**

In the main menu, select Secure Storage -> File Protect.

The application prompts the user to select a file to protect. Navigate into the directory or folder containing the file, and select the same.



Next, the application prompts the user to specify which name this selected file should be protected into. The user could, after navigating into the appropriate directory or folder, either select an existing file, or type a new filename from the keyboard. The user should ensure that the name of the protected file be different from that of the unprotected file.



After specifying the name of the protected file, the application prompts the user for the 16-character passphrase to be used to protect the file.



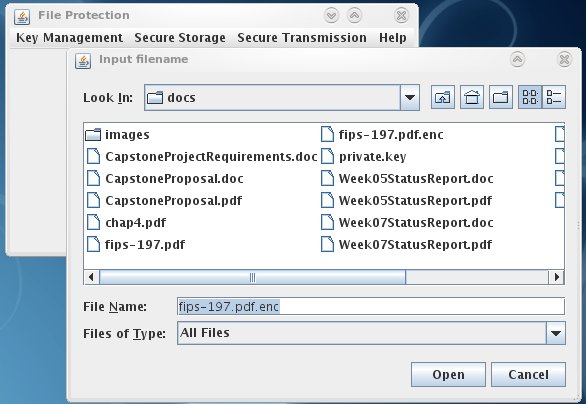
Upon specifying the passphrase, there is a short wait involved after which a message appears that file protection is successful. The main menu appears when the user clicks on the OK button.



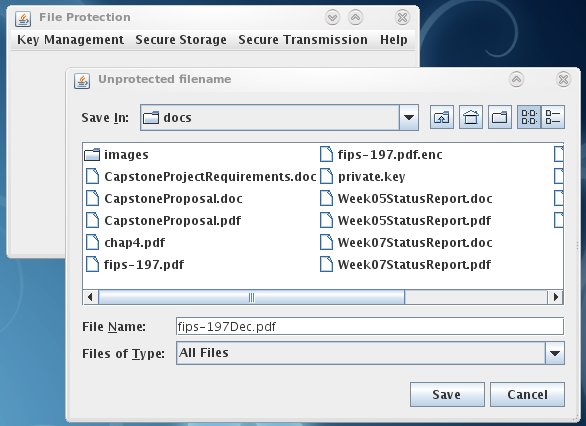
**A.3.2.2 File Unprotect**

In the main menu, select Secure Storage -> File Unprotect.

The application prompts the user to select a file to protect. Navigate into the directory or folder containing the file, and select the same.



Next, the application prompts the user to specify which name this selected file should be unprotected into. The user could, after navigating into the appropriate directory or folder, either select an existing file, or type a new filename from the keyboard. The user should ensure that the name of the unprotected file be different from that of the protected file.



After specifying the name of the protected file, the application prompts the user for the 16-character passphrase to be used to protect the file.



Upon specifying the passphrase, there is a short wait involved after which a message appears that file unprotection is successful. The main menu appears when the user clicks on the OK button.



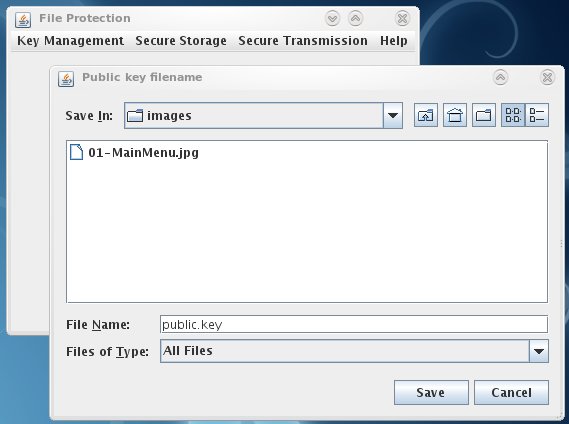
**A.3.3 Key Management**

A public-private key-pair is required to be generated in order to secure files for transmission. Usually the sender of the file to be transmitted requests for the recipient’s public key. The recipient uses this part of the application to generate the RSA public-private key-pair. The application generates the two keys forming the pair into two files.

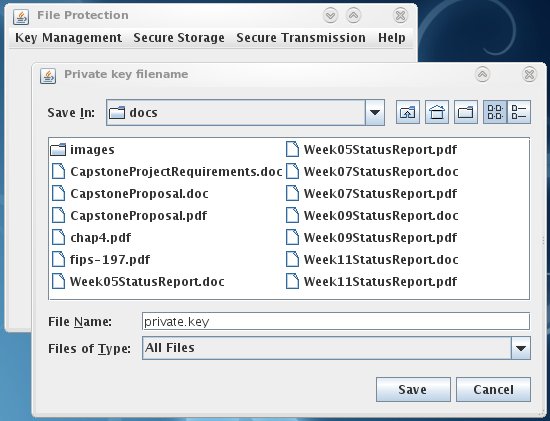
The recipient e-mails the public key file to the sender as attachment, for the sender to use the same to secure the message into an envelope for transmission to the recipient. The recipient *does not* send the private key file to anyone. This private key is kept protected on the recipient’s storage using a 16-character passphrase.

In the main menu, select Key Management -> Key-pair Generation.

The application prompts for the name of the file in which the public key is to be stored.



Next, the application prompts the user to specify the name of the file to store the private key into. The user could, after navigating into the appropriate directory or folder, either select an existing file, or type a new filename from the keyboard. The user should ensure that the name of the private key file be different from that of the public key file.



After specifying the name of the private key file, the application prompts the user for the 16-character passphrase to be used to protect the private key file.



Upon specifying the passphrase, there is a short wait involved after which a message appears that the public-private key-pair generation is successful. The main menu appears when the user clicks on the OK button.



The public-private key-pair is generated, and will now be used to secure files for transmission to by e-mail.

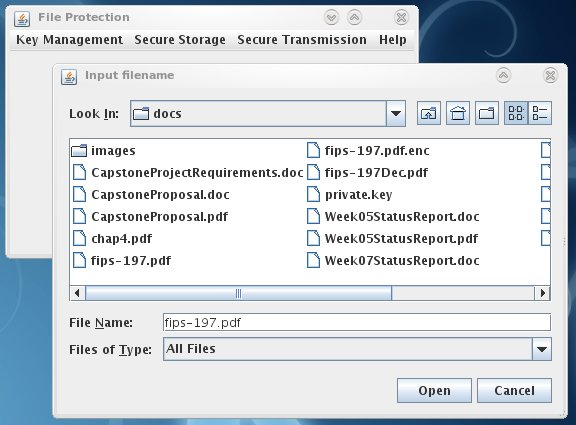
**A.3.4 Secure Transmission**

This part of the application secures a file on disk for transmission to a recipient by creating a message envelope. It consists of two parts: Create Message Envelope and Retrieve Message from Envelope.

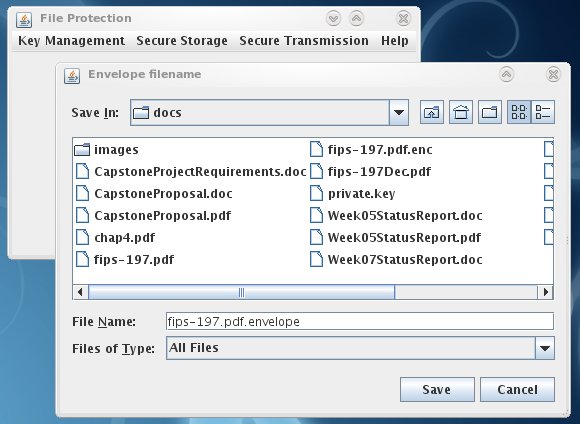
**A.3.4.1 Create Message Envelope**

In the main menu, select Secure Transmission -> Create Message Envelope.

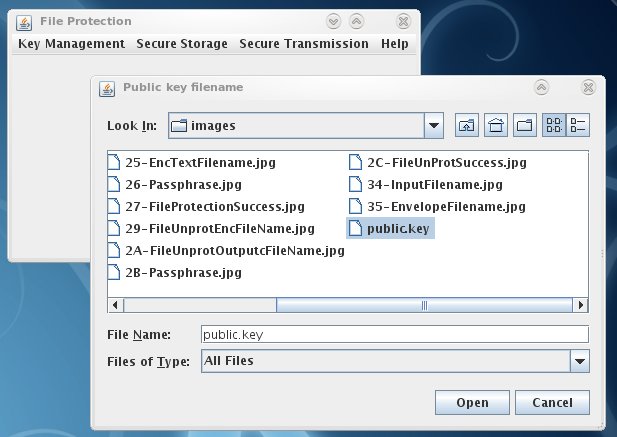
The application prompts the user to select a file to be inserted into the envelope being created. Navigate into the directory or folder containing the file, and select the same.



Next, the application prompts the user to specify which name this selected file should be enveloped into. The user could, after navigating into the appropriate directory or folder, either select an existing file, or type a new filename from the keyboard. The user should ensure that the name of the enveloped file be different from that of the input data file specified earlier.



After specifying the envelope filename, the next step is to specify the name of recipient’s the public key file.



The application now does the following. A 16-character random number is generated and is used to protect the input file being enveloped. This 16-character random number is protected with the recipient’s public key. Only the recipient will be able to retrieve the file because only the recipient possesses the private key corresponding to this public key.

The protected random number, and the protected file, are bundled together into the envelope which is the output file specified with filename earlier.

There is a short wait involved after which a message appears that the envelope creation is successful. The main menu appears when the user clicks on the OK button.



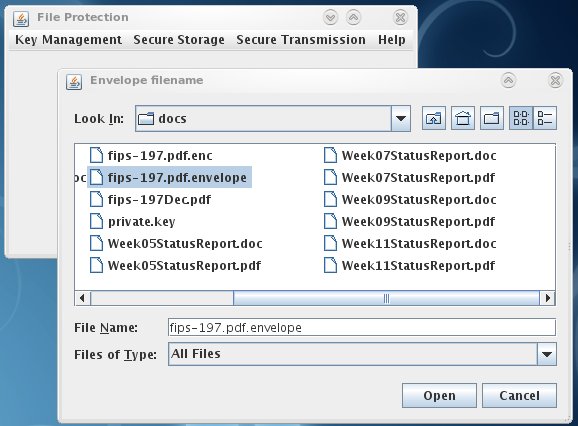
The sender now transmits this envelope file to a recipient as an attachment.

**A.3.4.2 Retrieve Message from Envelope**

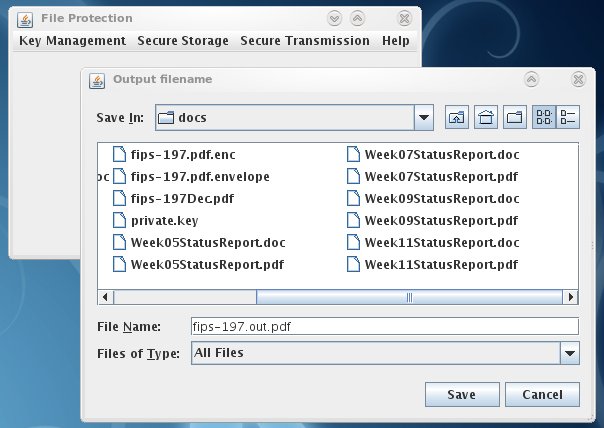
This part of the application is used by the recipient of the message. (A copy of the application should be installed and made available on the recipient’s computer.)

In the main menu, the recipient selects Secure Transmission -> Retrieve Message From Envelope.

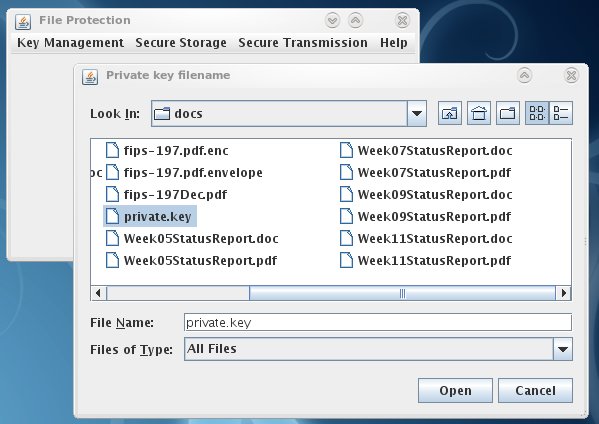
The application prompts the user to select the name of the envelope file to be opened. After downloading the attachment from the e-mail, navigate into the directory or folder containing the downloaded attachment file, and select the same.



Next, the application prompts the recipient to specify the name of the file into which the data should be retrieved from the envelope. The recipient could, after navigating into the appropriate directory or folder, either select an existing file, or type a new filename from the keyboard. The recipient should ensure that the name of this output file be different from that of the envelope file specified earlier.



After specifying the output file, the application prompts for the recipient’s private key filename. The recipient selects the private key filename accordingly.



This private key is protected by the recipient’s passphrase that specified during generation of the public-private key-pair earlier. The application now prompts for this 16-character passphrase.



The application now performs the following steps. It first decrypts the recipient’s private key file using the passphrase. It now opens the envelope and retrieves random number protected by the sender using the public key corresponding to private key. The random number is now used to decrypt the data file in the envelope. This data file is saved onto disk as the output file whose name was specified earlier.

There is a short wait involved after which a message appears that the message retrieval from envelope is successful. The main menu appears when the user clicks on the OK button.



**A.3.4 Exiting the Application**

To exit from the main menu, the user clicks on the X at the top-right corner of the window bar. The user is now back to the command-prompt.

**Appendix A.4. Source Code**

/\*

\* FileProtect.java

\*

\* Created on March 18, 2009, 10:15 PM

\*/

/\*\*

\*

\* @author Saleh Alsagoor

\*/

import java.io.File;

import java.io.IOException ;

import java.io.FileNotFoundException ;

import java.awt.event.ActionListener ;

import java.awt.event.ActionEvent ;

import javax.swing.JOptionPane ;

import javax.swing.JPasswordField ;

import javax.swing.JFileChooser ;

public class FileProtect extends javax.swing.JFrame {

/\* File-chooser GUI, defined here to persist across method calls. \*/

JFileChooser fChooser ;

/\*\* Creates new form FileProtect \*/

public FileProtect() {

super("File Protection") ;

initComponents();

fChooser = new JFileChooser() ;

}

/\*\* This method is called from within the constructor to

\* initialize the form.

\* WARNING: Do NOT modify this code. The content of this method is

\* always regenerated by the Form Editor.

\*/

@SuppressWarnings("unchecked")

// <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents

private void initComponents() {

jMenuBar1 = new javax.swing.JMenuBar();

jMenuKeyMgmt = new javax.swing.JMenu();

jMenuItemKeyGen = new javax.swing.JMenuItem();

jMenuStorage = new javax.swing.JMenu();

jMenuItemEncryptFile = new javax.swing.JMenuItem();

jMenuItemDecryptFile = new javax.swing.JMenuItem();

jMenuTransmission = new javax.swing.JMenu();

jMenuItemCreateEnv = new javax.swing.JMenuItem();

jMenuItemRetrieveEnv = new javax.swing.JMenuItem();

jMenuHelp = new javax.swing.JMenu();

jMenuItemAbout = new javax.swing.JMenuItem();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT\_ON\_CLOSE);

jMenuKeyMgmt.setText("Key Management");

jMenuItemKeyGen.setText("Generate public-private key-pair");

jMenuKeyMgmt.add(jMenuItemKeyGen);

jMenuItemKeyGen.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Generate public-private key-pair into two different files. \*/

keyPairGen() ;

}

}

) ;

jMenuBar1.add(jMenuKeyMgmt);

jMenuStorage.setText("Secure Storage");

jMenuItemEncryptFile.setText("Protect file");

jMenuStorage.add(jMenuItemEncryptFile);

jMenuItemEncryptFile.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Encrypt an input file into an output file. \*/

EncryptDecryptFile(0) ;

}

}

) ;

jMenuItemDecryptFile.setText("Unprotect file");

jMenuStorage.add(jMenuItemDecryptFile);

jMenuItemDecryptFile.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Decrypt an input file into an output file. \*/

EncryptDecryptFile(1) ;

}

}

) ;

jMenuBar1.add(jMenuStorage);

jMenuTransmission.setText("Secure Transmission");

jMenuItemCreateEnv.setText("Create envelope");

jMenuTransmission.add(jMenuItemCreateEnv);

jMenuItemCreateEnv.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Protect an input data file into an envelope. \*/

CreateMessageEnvelope() ;

}

}

) ;

jMenuItemRetrieveEnv.setText("Retrieve file from envelope");

jMenuTransmission.add(jMenuItemRetrieveEnv);

jMenuItemRetrieveEnv.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Retrieve a data file from an envelope. \*/

RetrieveFromMessageEnvelope() ;

}

}

) ;

jMenuBar1.add(jMenuTransmission);

jMenuHelp.setText("Help");

jMenuItemAbout.setText("About...");

jMenuHelp.add(jMenuItemAbout);

jMenuItemAbout.addActionListener(

new ActionListener()

{

public void actionPerformed(ActionEvent event)

{

/\* Display an About Box. \*/

String line1 = "This application secures files for\n" ;

String line2 = "storage on disk, and for transmission\n" ;

String line3 = "to a recipient.\n" ;

String heading = "About" ;

JOptionPane.showMessageDialog(FileProtect.this,

line1+line2+line3, heading, JOptionPane.PLAIN\_MESSAGE) ;

}

}

) ;

jMenuBar1.add(jMenuHelp);

setJMenuBar(jMenuBar1);

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());

getContentPane().setLayout(layout);

layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGap(0, 400, Short.MAX\_VALUE)

);

layout.setVerticalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGap(0, 279, Short.MAX\_VALUE)

);

pack();

}// </editor-fold>//GEN-END:initComponents

/\* This method helps choose a file to save to.

\* It uses an object of class "JFileChooser".

\* The method returns the filename (along with path) if selected correctly,

\* or the empty string "" if the "Cancel" button got pressed.

\*/

private String chooseFileToSave(String heading)

{

boolean valid = true;

File fileObj;

String fileName = "";

int retval = 0 ;

fChooser.setDialogTitle(heading);

/\* Repeat the following steps until either a valid filename

\* is specified, or the Cancel button is pressed. \*/

while (valid)

{

retval = fChooser.showSaveDialog(FileProtect.this);

if (retval == JFileChooser.APPROVE\_OPTION)

{

/\* "Save" button got pressed. Obtain filename. \*/

fileObj = fChooser.getSelectedFile() ;

/\* Obtain pathname. \*/

fileName = fileObj.getAbsolutePath() ;

/\* Does the file already exist? \*/

if (fileObj.exists())

{

/\* It does. Prompt for overwriting the file. \*/

int yes\_no = JOptionPane.showConfirmDialog(FileProtect.this,

"File " + fileName + " already exists. Overwrite?",

"File exists", JOptionPane.YES\_NO\_OPTION, JOptionPane.WARNING\_MESSAGE);

if (yes\_no != JOptionPane.YES\_OPTION)

{

fileName = "" ;

}

valid = (yes\_no != JOptionPane.YES\_OPTION);

}

else

{

/\* The file does not exist. This is a valid case.

\* Terminate the loop. \*/

valid = false;

}

}

else if (retval == JFileChooser.CANCEL\_OPTION)

{

/\* The "cancel" button got pressed. Terminate the loop. \*/

fileName = "" ;

valid = false;

}

}

return (fileName);

}

/\* This method helps choose a file to open.

\* It uses an object of class "JFileChooser".

\* The method returns the filename (along with path) if selected correctly,

\* or the empty string "" if the "Cancel" button got pressed.

\*/

private String chooseFileToOpen(String heading)

{

Object[] options =

{

"Yes", "No"

};

boolean valid = true;

File fileObj;

String fileName = "";

fChooser.setDialogTitle(heading);

/\* Repeat the following steps until either a valid filename

\* is specified, or the Cancel button is pressed. \*/

while (valid)

{

int retval = fChooser.showOpenDialog(FileProtect.this);

if (retval == JFileChooser.APPROVE\_OPTION)

{

/\* "Open" button got pressed. Obtain filename. \*/

fileObj = fChooser.getSelectedFile() ;

/\* Obtain pathname. \*/

fileName = fileObj.getAbsolutePath() ;

valid = !(fileObj.exists()) ;

/\* Does the file exist? \*/

if (valid)

{

/\* It does not. Report error and go back to loop to

\* show the file-chooser dialog. \*/

JOptionPane.showMessageDialog(FileProtect.this,

"File does not exist.", "Error", JOptionPane.ERROR\_MESSAGE) ;

}

}

else if (retval == JFileChooser.CANCEL\_OPTION)

{

/\* The "cancel" button got pressed. Terminate the loop. \*/

fileName = "" ;

valid = false;

}

}

return (fileName);

}

/\* This method outputs two selected, \_different\_ filenames (along with paths)

\* or two empty strings. The last parameter determines whether each file

\* being selected is for saving (true) or for opening (false).

\* If different filenames got selected, the return value is true and if they

\* are both empty, the return value is false. \*/

boolean choose2Files(String[] fileName, String[] heading, String Operation,

boolean[] save)

{

boolean validFiles = false ;

/\* Choose first file. \*/

if (save[0])

{

fileName[0] = this.chooseFileToSave(heading[0]) ;

}

else

{

fileName[0] = this.chooseFileToOpen(heading[0]) ;

}

if (fileName[0].length() > 0)

{

/\* Non-empty first filename specified. Select the second filename. \*/

do

{

/\* Choose second file. \*/

if (save[1])

{

fileName[1] = this.chooseFileToSave(heading[1]) ;

}

else

{

fileName[1] = this.chooseFileToOpen(heading[1]) ;

}

if (fileName[1].length() > 0)

{

/\* Non-empty second filename specified.

\* Ensure that the two filenames are different. \*/

if (fileName[1].equals(fileName[0]))

{

/\* The two filenames cannot be the same. Error-trap this. \*/

JOptionPane.showMessageDialog(FileProtect.this,

"Files should be different.",

"Error", JOptionPane.ERROR\_MESSAGE) ;

}

else

{

/\* Valid second filename specified. \*/

validFiles = true ;

}

}

else

{

/\* Empty second filename specified. (Cancel button got pressed. \*/

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

}

} while (fileName[1].equals(fileName[0])) ;

}

else

{

/\* Empty first filename specified. (Cancel button got pressed. \*/

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

}

return (validFiles) ;

}

/\* This method returns a 16-character passphrase typed in a password field,

\* or the empty-string if the Cancel button got pressed on the top-right

\* corner of the passphrase-dialog. \*/

private String getPassphrase()

{

String passPhrase = "";

int retval = 0 ;

JPasswordField passwordField = new JPasswordField(Aes.\_KEYLENBYTES) ;

do

{

/\* Prompt for entering the passphrase. \*/

retval = JOptionPane.showConfirmDialog (FileProtect.this, passwordField,

"Enter 16-character passphrase", JOptionPane.OK\_CANCEL\_OPTION );

char[] passChar = passwordField.getPassword() ;

passPhrase = new String(passChar) ;

/\* Is the OK key pressed AND the password exactly 16 bytes long? \*/

if ((retval != JOptionPane.CANCEL\_OPTION) && (retval != JOptionPane.CLOSED\_OPTION)

&& (passPhrase.length() != Aes.\_KEYLENBYTES))

{

/\* OK is pressed, but the password is not 16 bytes long. Error-trap this. \*/

JOptionPane.showMessageDialog(FileProtect.this,

"Passphrase should be 16 characters long.",

"Error", JOptionPane.ERROR\_MESSAGE) ;

}

else

{

/\* OK is not pressed. (The only other options are Cancel and Kill the dialog.)

\* This is valid, regardless of passphrase length. Reset the passphrase to

\* the empty string "" and terminate the loop. \*/

if ((retval == JOptionPane.CANCEL\_OPTION) || (retval == JOptionPane.CLOSED\_OPTION))

{

passPhrase = "" ;

}

}

} while ((retval != JOptionPane.CANCEL\_OPTION) && (retval != JOptionPane.CLOSED\_OPTION)

&& (passPhrase.length() != Aes.\_KEYLENBYTES)) ;

return (passPhrase) ;

}

/\* This method generates an RSA public-private key-pair. \*/

private void keyPairGen()

{

boolean validFiles = false ;

String passPhrase = "" ;

String[] fileName = new String[2] ;

String[] heading = {"Public key filename", "Private key filename"} ;

String Operation = "Key-pair generation" ;

boolean[] save = {true, true} ;

validFiles = choose2Files(fileName, heading, Operation, save) ;

if (validFiles)

{

/\* Obtain the passphrase used to protect the private key. \*/

passPhrase = this.getPassphrase() ;

if (passPhrase.equals(""))

{

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

}

else

{

/\* Generate public-private key-pair into the two files. \*/

Rsa rsa = new Rsa() ;

try

{

rsa.keyGen(Rsa.\_KEYLENGTH, fileName[0], fileName[1], passPhrase);

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " successful", "Success",

JOptionPane.INFORMATION\_MESSAGE) ;

}

catch (FileNotFoundException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "File not found exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

catch (IOException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "Input/output exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

}

}

}

/\* This method protects / unprotects a file using a passphrase.

\* The operation is selected by the parameter "operation".

\* If operation = 0, it means protection. If 1, it means unprotection. \*/

private void EncryptDecryptFile(int operation)

{

boolean validFiles = false ;

String passPhrase = "" ;

String[] fileName = new String[2] ;

String[][] heading = {{"Input filename", "Protected filename"},{"Input filename", "Unprotected filename"}} ;

String[] Operation = {"File protection", "File unprotection"} ;

boolean[] save = {false, true} ;

/\* Choose the two files. \*/

validFiles = choose2Files(fileName, heading[operation], Operation[operation], save) ;

if (validFiles)

{

/\* Obtain the passphrase used to protect / unprotect the input file. \*/

passPhrase = this.getPassphrase() ;

if (passPhrase.equals(""))

{

JOptionPane.showMessageDialog(FileProtect.this,

Operation[operation] + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

}

else

{

/\* Protect / unprotect the input file into the output file. \*/

Aes aes = new Aes(passPhrase) ;

try

{

if (operation == 0)

{

aes.encryptFile(fileName[0], fileName[1]);

}

else

{

aes.decryptFile(fileName[0], fileName[1]);

}

JOptionPane.showMessageDialog(FileProtect.this,

Operation[operation] + " successful", "Success",

JOptionPane.INFORMATION\_MESSAGE) ;

}

catch (FileNotFoundException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "File not found exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

catch (IOException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "Input/output exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

}

}

}

/\* This method protects a file for transmission using a Message Envelope. \*/

private void CreateMessageEnvelope()

{

boolean validFiles = false ;

String[] fileName = new String[2] ;

String pubKeyFileName = "" ;

String[] heading = {"Input filename", "Envelope filename"} ;

String Operation = "Envelope creation" ;

boolean[] save = {false, true} ;

/\* Choose the two files. \*/

validFiles = this.choose2Files(fileName, heading, Operation, save) ;

if (validFiles)

{

/\* Obtain the filename for the public key used to create the message

\* envelope. Ensure that this filename is different from the other two. \*/

do

{

pubKeyFileName = this.chooseFileToOpen("Public key filename") ;

if (pubKeyFileName.length() == 0)

{

/\* The cancel button got pressed. Report this and stop the loop. \*/

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

validFiles = false ;

}

else

{

/\* Error-trap the fact that the public key filename may be identical

\* to the other two. \*/

if (pubKeyFileName.equals(fileName[0])

|| pubKeyFileName.equals(fileName[1]))

{

JOptionPane.showMessageDialog(FileProtect.this,

"Files should be different.",

"Error", JOptionPane.ERROR\_MESSAGE) ;

}

}

} while (validFiles && (pubKeyFileName.equals(fileName[0])

|| (pubKeyFileName.equals(fileName[1])))) ;

if (validFiles)

{

/\* Protect the input data file into the message envelope. \*/

try

{

MsgEnvelope.createEnvelope(fileName[1], fileName[0], pubKeyFileName);

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " successful", "Success",

JOptionPane.INFORMATION\_MESSAGE) ;

}

catch (FileNotFoundException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "File not found exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

catch (IOException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "Input/output exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

}

}

}

/\* This method retrieves a file from a Message Envelope. \*/

private void RetrieveFromMessageEnvelope()

{

boolean validFiles = false ;

String[] fileName = new String[2] ;

String pvtKeyFileName = "" ;

String passPhrase = "" ;

String[] heading = {"Envelope filename", "Output filename"} ;

String Operation = "Retrieval from envelope" ;

boolean[] save = {false, true} ;

/\* Choose the two files. \*/

validFiles = this.choose2Files(fileName, heading, Operation, save) ;

if (validFiles)

{

/\* Obtain the filename for the private key used to retrieve the data file

\* from the message envelope. Ensure that this filename is different from

\* the other two. \*/

do

{

pvtKeyFileName = this.chooseFileToOpen("Private key filename") ;

if (pvtKeyFileName.length() == 0)

{

/\* The cancel button got pressed. Report this and stop the loop. \*/

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

validFiles = false ;

}

else

{

/\* Error-trap the fact that the private key filename may be identical

\* to the other two. \*/

if (pvtKeyFileName.equals(fileName[0])

|| pvtKeyFileName.equals(fileName[1]))

{

JOptionPane.showMessageDialog(FileProtect.this,

"Files should be different.",

"Error", JOptionPane.ERROR\_MESSAGE) ;

}

}

} while (validFiles && (pvtKeyFileName.equals(fileName[0])

|| (pvtKeyFileName.equals(fileName[1])))) ;

if (validFiles)

{

/\* Obtain the passphrase used to retrieve the private key. \*/

passPhrase = this.getPassphrase() ;

if (passPhrase.equals(""))

{

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " canceled", "Failure", JOptionPane.WARNING\_MESSAGE) ;

}

else

{

/\* Retrieve the data file into the message envelope. \*/

try

{

MsgEnvelope.retrieveData(fileName[1], fileName[0], pvtKeyFileName, passPhrase) ;

JOptionPane.showMessageDialog(FileProtect.this,

Operation + " successful", "Success",

JOptionPane.INFORMATION\_MESSAGE) ;

}

catch (FileNotFoundException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "File not found exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

catch (IOException e)

{

JOptionPane.showMessageDialog(FileProtect.this, "Input/output exception.",

"Error", JOptionPane.ERROR\_MESSAGE);

}

}

}

}

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String args[]) {

java.awt.EventQueue.invokeLater(new Runnable() {

public void run() {

new FileProtect().setVisible(true);

}

});

}

// Variables declaration - do not modify//GEN-BEGIN:variables

private javax.swing.JMenuBar jMenuBar1;

private javax.swing.JMenu jMenuHelp;

private javax.swing.JMenuItem jMenuItemAbout;

private javax.swing.JMenuItem jMenuItemCreateEnv;

private javax.swing.JMenuItem jMenuItemDecryptFile;

private javax.swing.JMenuItem jMenuItemEncryptFile;

private javax.swing.JMenuItem jMenuItemKeyGen;

private javax.swing.JMenuItem jMenuItemRetrieveEnv;

private javax.swing.JMenu jMenuKeyMgmt;

private javax.swing.JMenu jMenuStorage;

private javax.swing.JMenu jMenuTransmission;

// End of variables declaration//GEN-END:variables

}

/\*

\* This class encapsulates the Advanced Encryption Standard (AES).

\* AES is a block cipher with a block length of 128 bits and a

\* key-length of 128 bits.

\* The algorithm for this program is specified in the Federal Information

\* Processing Standard document FIPS-197.

\*/

/\*\*

\*

\* @author Saleh Alsagoor

\*/

import java.io.FileInputStream ;

import java.io.FileOutputStream ;

import java.io.IOException ;

import java.io.FileNotFoundException ;

public class Aes

{

public static final int \_KEYLENBYTES = 16 ;

public static final int \_BLOCKLENBYTES = 16 ;

public static final int \_NUMOFROUNDS = 10 ;

private static final int \_NUMOFSUBKEYS = 44 ;

private static final int \_ROUNDCONSTANT[] =

{0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000,

0x20000000, 0x40000000, 0x80000000, 0x1b000000, 0x36000000} ;

private final int[][] subBytes =

{

{0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76},

{0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0},

{0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15},

{0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75},

{0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84},

{0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf},

{0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8},

{0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2},

{0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73},

{0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb},

{0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79},

{0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08},

{0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a},

{0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e},

{0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf},

{0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16},

} ;

private final int[][] subBytesInverse = {

{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb},

{ 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb},

{ 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e},

{ 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25},

{ 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92},

{ 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84},

{ 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06},

{ 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b},

{ 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73},

{ 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e},

{ 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b},

{ 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4},

{ 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f},

{ 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef},

{ 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61},

{ 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d},

} ;

/\* Regular constructor. This takes in a passphrase as a byte

\* array and sets up the expanded subkeys. \*/

public Aes(byte[] passPhrase)

{

/\* Set up the subkey array. \*/

expand(passPhrase) ;

}

/\* Regular constructor. This takes in a passphrase as a character

\* array and sets up the expanded subkeys. \*/

public Aes(char[] passPhrase)

{

/\* Set up the subkey array. \*/

expand(passPhrase) ;

}

/\* Regular constructor. This takes in a passphrase as a string

\* and sets up the expanded subkeys. \*/

public Aes(String passPhrase)

{

/\* Set up the subkey array. \*/

expand(passPhrase.toCharArray()) ;

}

/\* The subkey array. \*/

private int[] word ;

/\* Expand the passphrase as a character-array into

\* the AES subkey array "word" having 44 words. \*/

private void expand(char [] passPhrase)

{

int index ;

byte[] key = new byte[\_KEYLENBYTES] ;

for (index = 0; index < \_KEYLENBYTES; index++)

{

key[index] = (byte) ((int) (passPhrase[index] & 0xff)) ;

}

expand(key) ;

}

/\* Expand the passphrase as a byte-array into

\* the AES subkey array "word" having 44 words. \*/

private void expand(byte [] passPhrase)

{

/\* First, convert each byte in "passPhrase" into a

\* byte and store in a byte array "key". \*/

int index, tempWord ;

byte[] key = new byte[\_KEYLENBYTES] ;

for (index = 0; index < \_KEYLENBYTES; index++)

{

key[index] = passPhrase[index] ;

}

/\* Set up the subkey array of 44 words and initialize to zero. \*/

word = new int[\_NUMOFSUBKEYS] ;

for (index = 0; index < \_NUMOFSUBKEYS; index++)

word[index] = 0 ;

/\* The array "key" contains 16 bytes. Convert groups

\* of 4 bytes into 32-bit words. There should be four

\* such 32-bit words, occupying word[0], word[1], word[2]

\* and word[3]. \*/

for (index = 0; index < 4; index++)

{

word[index] = (key[4\*index] << 24)

| ((key[4\*index+1] << 16) & 0x00ff0000)

| ((key[4\*index+2] << 8) & 0x0000ff00)

| ((key[4\*index+3]) & 0x000000ff) ;

}

/\* Words word[0..3] are ready. Now prepare groups

\* word[4..7], word[8..11], .... word[39..43]. \*/

for (index = 4; index < \_NUMOFSUBKEYS; index++)

{

tempWord = word[index-1] ;

if (index % 4 == 0)

{

tempWord = subWord(rotateLeft(tempWord, 8)) ;

tempWord ^= \_ROUNDCONSTANT[(index >> 2) - 1] ;

}

word[index] = word[index-4] ^ tempWord ;

}

}

/\* Substitute groups of 4 bytes by repeatedly accessing the

\* S-Box array"subBytes" for each byte. \*/

private int subWord(int input)

{

return ((subBytes[((input >> 28) & 0x0f)][((input >> 24) & 0x0f)] << 24)

| (subBytes[((input >> 20) & 0x0f)][((input >> 16) & 0x0f)] << 16)

| (subBytes[((input >> 12) & 0x0f)][((input >> 8) & 0x0f)] << 8)

| (subBytes[((input >> 4) & 0x0f)][((input) & 0x0f)])) ;

}

/\* Substitute groups of 4 bytes by repeatedly accessing the

\* inverse S-Box array"subBytesInverse" for each byte. \*/

private int subWordInverse(int input)

{

return ((subBytesInverse[((input >> 28) & 0x0f)][((input >> 24) & 0x0f)] << 24)

| (subBytesInverse[((input >> 20) & 0x0f)][((input >> 16) & 0x0f)] << 16)

| (subBytesInverse[((input >> 12) & 0x0f)][((input >> 8) & 0x0f)] << 8)

| (subBytesInverse[((input >> 4) & 0x0f)][((input) & 0x0f)])) ;

}

/\* The operation "shiftRow".

\* The input to this function is "state", which itself gets modified.

\* The input is a array of four words (32-bit quantities).

\* The first column is not changed. The column is left-rotated by one byte.

\* The third column is left-rotated by 2 bytes and the fourth, by 3 bytes.

\*/

private void shiftRow(int[] state)

{

int index, outState[] = new int[4] ;

outState[0] = outState[1] = outState[2] = outState[3] = 0 ;

for (index = 0; index < 4; index++)

{

outState[index] = (state[index] & 0xff000000)

| (state[(index+1) & 0x03] & 0x00ff0000)

| (state[(index+2) & 0x03] & 0x0000ff00)

| (state[(index+3) & 0x03] & 0x000000ff) ;

}

state[0] = outState[0] ; state[1] = outState[1];

state[2] = outState[2] ; state[3] = outState[3];

}

/\* The inverse of operation "shiftRow". \*/

private void shiftRowInverse(int[] state)

{

int index, outState[] = new int[4] ;

outState[0] = outState[1] = outState[2] = outState[3] = 0 ;

for (index = 0; index < 4; index++)

{

outState[index] = (state[index] & 0xff000000)

| (state[(index-1) & 0x03] & 0x00ff0000)

| (state[(index-2) & 0x03] & 0x0000ff00)

| (state[(index-3) & 0x03] & 0x000000ff) ;

}

state[0] = outState[0] ; state[1] = outState[1];

state[2] = outState[2] ; state[3] = outState[3];

}

/\* This method multiplies two bytes in GF(2^8)

\* modulo the polynomial (x^8+x^4+x^3+1). \*/

private byte multiplyGF256(byte multiplicand, byte multiplier)

{

int mult1 = multiplicand & 0xff ;

int mult2 = multiplier & 0xff ;

int product = 0, check ;

do

{

/\* Exclusive-OR partial product if multiplier is odd. \*/

product = product ^ (mult1 \* (mult2 & 0x01)) ;

check = ((mult1 & 0x80) >> 7) & 0x01 ;

/\* Double the multiplicand in GF(2^8). \*/

mult1 = ((mult1 << 1) ^ (check \* 0x1b)) & 0xff ;

/\* Advance to the next bit of the multiplier. \*/

mult2 = (mult2 >> 1) & 0xff ;

} while (mult2 > 0) ;

return ((byte) product) ;

}

/\* This method mixes a fixed matrix into the columns of "state",

\* which is a 4-word array. The fixed matrix is given by:

\* 0x02 0x03 0x01 0x01

\* 0x01 0x02 0x03 0x01

\* 0x01 0x01 0x02 0x03

\* 0x03 0x01 0x01 0x02

\* Mixing means, multiplying "state" with this matrix. The multiplication

\* and addition are done in GF(2^8) modulo the polynomial (x^8+x^4+x^3+x+1).

\* The multiplication of each byte in the state is done with each element of

\* the above matrix using the method "multiplyGF256".

\*/

private void mixColumn(int[] state)

{

int outState0, outState1, outState2, outState3 ;

int index ;

outState0 = outState1 = outState2 = outState3 = 0 ;

// int[] outState = new int[state.length] ;

for (index = 0; index < state.length; index++)

{

outState0 = (int) ((multiplyGF256((byte) 0x02, (byte) ((state[index]>>24) & 0xff))

^ multiplyGF256((byte) 0x03, (byte) ((state[index]>>16) & 0xff))

^ ((state[index]>>8) & 0xff)

^ (state[index] & 0xff)) << 24) & 0xff000000 ;

outState1 = (int) (((state[index]>>24) & 0xff

^ multiplyGF256((byte) 0x02, (byte) ((state[index]>>16) & 0xff))

^ multiplyGF256((byte) 0x03, (byte) ((state[index]>>8) & 0xff))

^ (state[index] & 0xff)) << 16) & 0x00ff0000 ;

outState2 = (int) ((((state[index]>>24) & 0xff)

^ ((state[index]>>16) & 0xff)

^ multiplyGF256((byte) 0x02, (byte) ((state[index]>>8) & 0xff))

^ multiplyGF256((byte) 0x03, (byte) (state[index] & 0xff))) << 8) & 0x0000ff00 ;

outState3 = (int) ((multiplyGF256((byte) 0x03, (byte) ((state[index]>>24) & 0xff))

^ ((state[index]>>16) & 0xff)

^ ((state[index]>>8) & 0xff)

^ multiplyGF256((byte) 0x02, (byte) (state[index] & 0xff)))) & 0x000000ff ;

state[index] = outState0 | outState1 | outState2 | outState3 ;

}

}

/\* This method mixes a fixed matrix into the columns of "state",

\* which is a 4-word array. The fixed matrix is given by:

\* 0x0e 0x0b 0x0d 0x09

\* 0x09 0x0e 0x0b 0x0d

\* 0x0d 0x09 0x0b 0x0e

\* 0x0e 0x0d 0x09 0x0b

\* Mixing means, multiplying "state" with this matrix. The multiplication

\* and addition are done in GF(2^8) modulo the polynomial (x^8+x^4+x^3+x+1).

\* The multiplication of each byte in the state is done with each element of

\* the above matrix using the method "multiplyGF256".

\*/

private void mixColumnInverse(int[] state)

{

int outState0, outState1, outState2, outState3 ;

int index ;

outState0 = outState1 = outState2 = outState3 = 0 ;

for (index = 0; index < state.length; index++)

{

outState0 = (int) (((multiplyGF256((byte) 0x0e, (byte) ((state[index]>>24) & 0xff))

^ multiplyGF256((byte) 0x0b, (byte) ((state[index]>>16) & 0xff))

^ multiplyGF256((byte) 0x0d, (byte) ((state[index]>>8) & 0xff))

^ multiplyGF256((byte) 0x09, (byte) (state[index] & 0xff)))) << 24) & 0xff000000 ;

outState1 = (int) (((multiplyGF256((byte) 0x09, (byte) ((state[index]>>24) & 0xff))

^ multiplyGF256((byte) 0x0e, (byte) ((state[index]>>16) & 0xff))

^ multiplyGF256((byte) 0x0b, (byte) ((state[index]>>8) & 0xff))

^ multiplyGF256((byte) 0x0d, (byte) (state[index] & 0xff)))) << 16) & 0x00ff0000 ;

outState2 = (int) (((multiplyGF256((byte) 0x0d, (byte) ((state[index]>>24) & 0xff))

^ multiplyGF256((byte) 0x09, (byte) ((state[index]>>16) & 0xff))

^ multiplyGF256((byte) 0x0e, (byte) ((state[index]>>8) & 0xff))

^ multiplyGF256((byte) 0x0b, (byte) (state[index] & 0xff)))) << 8) & 0x0000ff00 ;

outState3 = (int) ((multiplyGF256((byte) 0x0b, (byte) ((state[index]>>24) & 0xff))

^ multiplyGF256((byte) 0x0d, (byte) ((state[index]>>16) & 0xff))

^ multiplyGF256((byte) 0x09, (byte) ((state[index]>>8) & 0xff))

^ multiplyGF256((byte) 0x0e, (byte) (state[index] & 0xff)))) & 0x000000ff ;

state[index] = outState0 | outState1 | outState2 | outState3 ;

}

}

/\* This function adds the key of a round into the current state.

\* Addition is done bit-wise modulo-2, meaning the exclusive-OR operation.

\* In this function, "round" lies between 0 (inclusive) and 10 (inclusive).

\* The inverse of this function is the same as this function due to the

\* identity property of the exclusive-OR operation. Hence this inverse

\* function is not implemented.

\*/

private void addRoundKey(int[] state, int round)

{

state[0] ^= word[4\*round] ;

state[1] ^= word[4\*round+1] ;

state[2] ^= word[4\*round+2] ;

state[3] ^= word[4\*round+3] ;

}

/\* This function encrypts in-place a 4-word (that is, 16-byte or alternatively,

\* 128-bit) block of plaintext, "block", to yield a 4-word ciphertext in

\* "block". The subkey array "word" populated from the encryption key by the

\* constructor of this class is used.

\*/

private void encryptBlock(int[] block)

{

int round ;

/\* Apply the rounds of encryption. \*/

for (round = 0; round <= 10; round++)

{

/\* All these steps are followed in rounds except Round 0. \*/

if (round > 0)

{

/\* Apply the Substitution Boxes. \*/

block[0] = subWord(block[0]) ;

block[1] = subWord(block[1]) ;

block[2] = subWord(block[2]) ;

block[3] = subWord(block[3]) ;

/\* Apply the operation "shiftRow". \*/

shiftRow(block) ;

/\* Now mix the columns (all rounds EXCEPT in Rounds 0 and 10). \*/

if (round < 10)

{

mixColumn(block) ;

}

}

/\* Add the round keys. \*/

/\* This is the ONLY step followed in Round 0. \*/

addRoundKey(block, round) ;

}

/\* "block" now contains the ciphertext. \*/

}

/\* This function encrypts an input block of bytes. \*/

public void encryptBlock(byte[] inBlock)

{

int index ;

int[] inWordBlock = new int[4] ;

/\* Marshall each group of four bytes into one word

\* and fill the array "inWordBlock" with these groups. \*/

for (index = 0; index < 4; index++)

{

inWordBlock[index] = ((inBlock[4\*index] << 24) & 0xff000000)

| ((inBlock[4\*index+1] << 16) & 0x00ff0000)

| ((inBlock[4\*index+2] << 8) & 0x0000ff00)

| ((inBlock[4\*index+3]) & 0x000000ff) ;

}

/\* Encrypt the input block of words. \*/

encryptBlock(inWordBlock) ;

/\* Unmarshall the input block of words into four groups

\* of bytes. \*/

for (index = 0; index < 4; index++)

{

inBlock[4\*index] = (byte) ((inWordBlock[index] >> 24) & 0xff) ;

inBlock[4\*index+1] = (byte) ((inWordBlock[index] >> 16) & 0xff) ;

inBlock[4\*index+2] = (byte) ((inWordBlock[index] >> 8) & 0xff) ;

inBlock[4\*index+3] = (byte) (inWordBlock[index] & 0xff) ;

}

}

/\* This function decrypts a block of plaintext. A block here is equivalent to

\* four words (four 32-bit data).

\* The steps in this function are, in a sense, the reverse of the steps in

\* the encryption function.

\*/

private void decryptBlock(int[] block)

{

int round ;

/\* Apply a round on the ciphertext block. \*/

for (round = 10; round >= 0; round--)

{

/\* Add the round keys, but ONLY in Round 10. \*/

if (round == 10)

{

addRoundKey(block, round) ;

}

/\* The following steps are now followed in all

\* rounds except Round 10 and Round 0. \*/

if (round < 10 && round >= 0)

{

/\* Apply the operation "shiftRowInverse". \*/

shiftRowInverse(block) ;

/\* Apply the inverse Substitution Boxes. \*/

block[0] = subWordInverse(block[0]) ;

block[1] = subWordInverse(block[1]) ;

block[2] = subWordInverse(block[2]) ;

block[3] = subWordInverse(block[3]) ;

/\* Add the round key. \*/

addRoundKey(block, round) ;

/\* Apply the operation "mixColumnInverse". This step is

\* not done in Round 0. \*/

if (round > 0)

{

mixColumnInverse(block) ;

}

}

}

/\* "block" now contains the plaintext. \*/

}

/\* This function decrypts an input block of bytes. \*/

public void decryptBlock(byte[] inBlock)

{

int index ;

int[] inWordBlock = new int[4] ;

/\* Marshall each group of four bytes into one word

\* and fill the array "inWordBlock" with these groups. \*/

for (index = 0; index < 4; index++)

{

inWordBlock[index] = ((inBlock[4\*index] << 24) & 0xff000000)

| ((inBlock[4\*index+1] << 16) & 0x00ff0000)

| ((inBlock[4\*index+2] << 8) & 0x0000ff00)

| ((inBlock[4\*index+3]) & 0x000000ff) ;

}

/\* Decrypt the input block of words. \*/

decryptBlock(inWordBlock) ;

/\* Unmarshall the input block of words into four groups

\* of bytes. \*/

for (index = 0; index < 4; index++)

{

inBlock[4\*index] = (byte) ((inWordBlock[index] >> 24) & 0xff) ;

inBlock[4\*index+1] = (byte) ((inWordBlock[index] >> 16) & 0xff) ;

inBlock[4\*index+2] = (byte) ((inWordBlock[index] >> 8) & 0xff) ;

inBlock[4\*index+3] = (byte) (inWordBlock[index] & 0xff) ;

}

}

/\* Rotate an input word to the left by "nBits" bits. \*/

private int rotateLeft(int wordInput, int nBits)

{

int mask = ((1 << nBits) - 1) ;

return ((wordInput << nBits) | (((wordInput) >> (32-nBits)) & mask)) ;

}

/\* Encrypt a file having the name "inFileName" into an output file

\* having name "outFileName".

\* The output file is always overwritten.

\*/

public void encryptFile(String inFileName, String outFileName)

throws FileNotFoundException, IOException

{

byte[] inBlock = new byte[\_BLOCKLENBYTES] ;

byte[] lastBlock ;

int inFileSize = 0, lastBlockLen = 0, index = 0 ;

FileInputStream inFile ;

FileOutputStream outFile ;

/\* Open input file for reading. \*/

try

{

inFile = new FileInputStream(inFileName) ;

/\* Open output file for writing. \*/

try

{

outFile = new FileOutputStream(outFileName) ;

/\* Determine the input file size (bytes). \*/

try

{

inFileSize = inFile.available() ;

/\* Determine the number of bytes in the last block. \*/

lastBlockLen = inFileSize % \_BLOCKLENBYTES ;

/\* Repeat the following steps. \*/

for (index = 0; index < inFileSize / \_BLOCKLENBYTES; index++)

{

/\* Read in a block of data from the input file. \*/

try

{

inFile.read(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error reading input block from input file.%n") ;

}

/\* Encrypt the input block. \*/

encryptBlock(inBlock) ;

/\* Write the encrypted block into the output file. \*/

try

{

outFile.write(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error writing output block into output file.%n") ;

}

}

/\* All input blocks are written to output file, encrypted.

\* Clear the input block. \*/

for (index = 0; index < inBlock.length; index++)

{

inBlock[index] = (byte) 0x00 ;

}

/\* Process the last block. \*/

if (lastBlockLen > 0)

{

/\* Input file-size is not a multiple of the last block.

\* Prepare last block. \*/

lastBlock = new byte[lastBlockLen] ;

/\* Read the last few bytes of size "lastBlockLen" from the input file. \*/

try

{

inFile.read(lastBlock) ;

}

catch (IOException e)

{

System.out.printf("Error reading last block from input file.%n") ;

}

/\* Next, copy the contents of the last block into the

\* input block. \*/

for (index = 0; index < lastBlock.length; index++)

{

inBlock[index] = lastBlock[index] ;

}

/\* And, pad the remaining bytes of the input block using

\* Cryptographic Message Syntax (CMS) padding. \*/

for (index = lastBlock.length; index < inBlock.length; index++)

{

inBlock[index] = (byte) (\_BLOCKLENBYTES - lastBlockLen) ;

}

}

else

{

/\* The input file-size is a multiple of the block-length.

\* Pad the input block with the block-length, as per CMS.

\*/

for (index = 0; index < inBlock.length; index++)

{

inBlock[index] = (byte) \_BLOCKLENBYTES ;

}

}

/\* Encrypt the last block prepared from input file, and/or padded. \*/

encryptBlock(inBlock) ;

/\* Write the encrypted block into the output file. \*/

try

{

outFile.write(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error writing encrypted last block into output file.%n") ;

}

}

catch (IOException e)

{

System.out.printf("Error reading input file size.%n") ;

}

finally

{

/\* Close the output file. \*/

try

{

outFile.close() ;

}

catch (IOException e)

{

System.out.printf("Error closing output file.%n") ;

}

}

}

finally

{

/\* Close the input file. \*/

try

{

inFile.close() ;

}

catch (IOException e)

{

System.out.printf("Error closing input file.%n") ;

}

}

}

catch (FileNotFoundException e)

{

System.out.printf("Input file not found.%n") ;

}

}

/\* Decrypt a file having the name "inFileName" into an output file

\* having name "outFileName".

\* The output file is always overwritten.

\*/

public void decryptFile(String inFileName, String outFileName)

throws FileNotFoundException, IOException

{

byte[] inBlock = new byte[\_BLOCKLENBYTES] ;

byte[] lastBlock ;

int inFileSize = 0, lastBlockLen = 0, index = 0 ;

FileInputStream inFile ;

FileOutputStream outFile ;

/\* Open input file for reading. \*/

try

{

inFile = new FileInputStream(inFileName) ;

/\* Open output file for writing. \*/

try

{

outFile = new FileOutputStream(outFileName) ;

/\* Determine the input file size (bytes). \*/

try

{

inFileSize = inFile.available() ;

/\* Repeat the following steps. \*/

for (index = 0; index < inFileSize / \_BLOCKLENBYTES-1; index++)

{

/\* Read in a block of data from the input file. \*/

try

{

inFile.read(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error reading input block from input file.%n") ;

}

/\* Encrypt the input block. \*/

decryptBlock(inBlock) ;

/\* Write the encrypted block into the output file. \*/

try

{

outFile.write(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error writing output block into output file.%n") ;

}

}

/\* All input blocks are written to output file, decrypted.

\* Clear the input block. \*/

for (index = 0; index < inBlock.length; index++)

{

inBlock[index] = (byte) 0x00 ;

}

/\* Process the last block. This last block will always be present

\* and completely filled, due to CMS padding. \*/

/\* Read the last (complete) block of size "\_BLOCKLENBYTES" from the

\* input file. \*/

try

{

inFile.read(inBlock) ;

}

catch (IOException e)

{

System.out.printf("Error reading last block from input file.%n") ;

}

/\* Decrypt this completely-filled last block. \*/

decryptBlock(inBlock) ;

/\* Now examine the contents of the last byte in the last block. \*/

lastBlockLen = \_BLOCKLENBYTES - inBlock[\_BLOCKLENBYTES-1] ;

if (lastBlockLen > 0)

{

/\* Unpadding of the last block is needed. (The size of the

\* input plaintext file is not an exact multiple of the

\* block-size.)

\* Prepare last block for unpadding. \*/

lastBlock = new byte[lastBlockLen] ;

/\* Next, copy the contents of the input block into the

\* last block. The length being copied is equal to the

\* number of bytes being unpadded.

\*/

for (index = 0; index < lastBlockLen; index++)

{

lastBlock[index] = inBlock[index] ;

}

/\* Write the unpadded block into the output file. \*/

try

{

outFile.write(lastBlock) ;

}

catch (IOException e)

{

System.out.printf("Error writing encrypted last block into output file.%n") ;

}

}

}

catch (IOException e)

{

System.out.printf("Error reading input file size.%n") ;

}

finally

{

/\* Close the output file. \*/

try

{

outFile.close() ;

}

catch (IOException e)

{

System.out.printf("Error closing output file.%n") ;

}

}

}

finally

{

/\* Close the input file. \*/

try

{

inFile.close() ;

}

catch (IOException e)

{

System.out.printf("Error closing input file.%n") ;

}

}

}

catch (FileNotFoundException e)

{

System.out.printf("Input file not found.%n") ;

}

}

public static void main(String[] args)

{

char [] password = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f} ;

Aes aes = new Aes(password) ;

int[] state = new int[4] ;

state[0] = 0x00112233 ;

state[1] = 0x44556677 ;

state[2] = 0x8899aabb ;

state[3] = 0xccddeeff ;

aes.encryptBlock(state) ;

System.out.printf("After encryption, state = %08x%08x%08x%08x%n",

state[0], state[1], state[2], state[3]) ;

try

{

aes.encryptFile("infile.txt", "outfile.txt") ;

aes.decryptFile("outfile.txt", "decryptedfile.txt") ;

}

catch (FileNotFoundException e)

{

System.out.printf("Error processing files.%n") ;

}

catch (IOException e)

{

System.out.printf("Some I/O error has occurred.%n") ;

}

}

}

/\*\*

\* Program: Rsa.java

\* @author Saleh Alsagoor

\* Purpose: This program encapsulates a class "Rsa" that specifies the

\* Rivest-Shamir-Adleman (RSA) encryption / decryption algorithm.

\* The algorithm is available from the textbook "Cryptography and Network

\* Security" by William Stallings, Pearson Education, 2003.

\*/

import java.math.BigInteger ;

import java.security.SecureRandom ;

import java.io.File ;

import java.io.FileInputStream ;

import java.io.FileOutputStream ;

import java.io.IOException ;

import java.io.FileNotFoundException ;

public class Rsa

{

public static final int \_KEYLENGTH = 1024 ; /\* Key-length measured in bits. \*/

private final int \_POSITIVE = 1 ;

private int[] oddPrimes ;

private int \_MAXODDPRIME = 20000 ; /\* Maximum number up to which primes are sought. \*/

private int \_NPRIMES = 2262 ; /\* There are 2,262 primes between 2 and 20,000. \*/

/\* Constructor.

\* This method generates and fills out primes less than 20,000

\* into an array "oddPrimes". There are 2,262 such primes.

\* It is used to test a random number for primality. \*/

public Rsa()

{

oddPrimes = new int[\_NPRIMES] ;

int index, noOfPrimes = 0, divisor ;

boolean isPrime ;

oddPrimes[0] = 2 ;

noOfPrimes = 1 ;

/\* Repeat the following steps. \*/

for (index = 3; index <= \_MAXODDPRIME; index += 2)

{

/\* Check if "index" is prime. \*/

isPrime = true ;

for (divisor = 2 ; ((divisor <= (int) Math.sqrt(index)) && isPrime); divisor++)

{

isPrime = (index % divisor != 0) ;

}

if (isPrime)

{

/\* "index" is indeed prime. Store the same into the array. \*/

oddPrimes[noOfPrimes] = index ;

noOfPrimes++ ;

}

}

}

/\* This method generates a random number between a minimum value and

\* a maximum value. \*/

private BigInteger genRandom(BigInteger mp\_min, BigInteger mp\_max, SecureRandom random)

{

/\* Computer a "modulus" big-integer equal to the difference between the

\* minimum and maximum, plus one. \*/

/\* mp\_mod = mp\_max - mp\_min + 1. \*/

BigInteger mp\_mod = (mp\_max.subtract(mp\_min)).add(BigInteger.ONE) ;

/\* Generate a random byte-sequence into an array of length equal to the

\* size of "mp\_max" (in bytes). \*/

byte[] secParam = new byte[mp\_max.bitLength() / 8] ;

random.nextBytes(secParam) ;

/\* Load the random byte-sequence into a big integer "mp\_t". \*/

BigInteger mp\_t = new BigInteger(\_POSITIVE, secParam) ;

/\* Divide "mp\_t" by "mp\_mod" and take the remainder. \*/

mp\_t = mp\_t.remainder(mp\_mod) ;

/\* 0 <= mp\_t <= (mp\_max - mp\_min).

\* Add "mp\_min" to yield the required result. \*/

mp\_t = mp\_t.add(mp\_min) ;

return (mp\_t) ;

}

/\* This method implements the Miller-Rabin primality test algorithm.

\* The algorithm tests with probability that the input number "mp\_n" is prime.

\* If the number is not prime, that result is reported with certainity.

\* In this case the method returns "false".

\* If it is probably prime as reported using the security parameter

\* "mp\_t", set to 10 for this project, the probability is equal to

\* (1-1/4^bit-length of mp\_n). The method now returns "true".

\* The algorithm is available in Handbook of Applied Cryptography by Alfred

\* Menezes, P.van Oorschot and S.Vanstone, Chapter 4, pp.139, CRC Press, 1996.

\*/

private boolean millerRabin(BigInteger mp\_n, SecureRandom random)

{

boolean isPrime = true ;

BigInteger mp\_i = BigInteger.ONE ;

BigInteger mp\_s = BigInteger.ZERO ;

BigInteger mp\_two = new BigInteger("2") ;

BigInteger mp\_a, mp\_y, mp\_j, mp\_s1 ;

/\* Load the security parameter (set to 10) into a big integer "mp\_t". \*/

BigInteger mp\_t = new BigInteger("10") ;

/\* Note that "mp\_n" is odd. Subtract 1, and two, from "mp\_n". \*/

BigInteger mp\_n1 = mp\_n.flipBit(0) ;

BigInteger mp\_n2 = mp\_n1.subtract(BigInteger.ONE) ;

/\* "mp\_n1" is even.

\* Repeatedly divide "mp\_n1" by 2 until it becomes odd.

\* Count the number of 2's taken out from "mp\_n1". \*/

BigInteger mp\_r = mp\_n1.add(BigInteger.ZERO) ;

do

{

mp\_r = mp\_r.divide(mp\_two) ;

mp\_s = mp\_s.add(BigInteger.ONE) ;

} while (!(mp\_r.testBit(0))) ;

/\* "mp\_n1" is now expressed as "mp\_r" multiplied by 2^"mp\_s". \*/

/\* Compute mp\_s1 = mp\_s - 1. \*/

mp\_s1 = mp\_s.subtract(BigInteger.ONE) ;

/\* The idea of the algorithm is the following. Generate a random

\* number "mp\_a" called a "witness" to the compositeness of "mp\_n".

\* Raise "mp\_a" to the power of "mp\_n1" modulo "mp\_n". If during

\* the process, any result equal to 1 or (mp\_n-1) is encountered,

\* then, "mp\_n" is certainly not prime.

\* The above test is repeated "mp\_t" times. If in any test, a witness

\* "mp\_a" tests positive to the compositeness of "mp\_n", the algorithm

\* terminates with certainity, and the method returns false. If in no

\* test, such a compositeness of "mp\_n" is detected, the algorithm reports

\* with probability that "mp\_n" is prime, and the method returns true.

\*/

/\* Repeat the following steps "mp\_t" times. \*/

while (((mp\_i.compareTo(mp\_t) <= 0) && (isPrime)))

{

/\* Generate a random number between 2 and "mp\_n2". \*/

mp\_a = genRandom(mp\_two, mp\_n2, random) ;

/\* Compute "mp\_a" to the power of "mp\_r" modulo "mp\_n". \*/

mp\_y = mp\_a.modPow(mp\_r, mp\_n) ;

/\* Check if "mp\_y" is either equal to 1, or to (mp\_n-1).

\* If so, "mp\_n" has quadratic residues and is not prime. \*/

if ((mp\_y.compareTo(BigInteger.ONE) != 0) && (mp\_y.compareTo(mp\_n1) != 0))

{

/\* Now repeatedly square "mp\_y" so that one eventually

\* computes "mp\_a" to the power of "mp\_n1" modulo "mp\_n".

\* If at any time after squaring, "mp\_y" is equal to either

\* 1, or to "mp\_n1", modulo "mp\_n", once again, "mp\_n"

\* has quadratic residues and is not prime. \*/

mp\_j = BigInteger.ONE ;

while ((mp\_j.compareTo(mp\_s1) <= 0) && (mp\_y.compareTo(mp\_n1) != 0) && (isPrime))

{

mp\_y = mp\_y.modPow(mp\_two, mp\_n) ;

isPrime = (mp\_y.compareTo(BigInteger.ONE) != 0) ;

mp\_j = mp\_j.add(BigInteger.ONE) ;

}

isPrime = (mp\_y.compareTo(mp\_n1) == 0) ;

}

/\* Increase "mp\_i", the count of the number of witnesses. \*/

mp\_i = mp\_i.add(BigInteger.ONE) ;

}

return (isPrime) ;

}

/\* This method implements an algorithm to generate (probable) primes.

\* A cryptographically secure pseudorandom seed "random" of bit-length

\* "keySize" bits (keySize/8 bytes) is used for the purpose.

\* After preliminary checking, the Miller-Rabin algorithm for primality

\* testing is used to test the primality of the random number.

\* The top two bits of the prime number returned by this method are both

\* ones. This will ensure that any two "k"-bit numbers generated in the

\* method "keyGen" that uses this method, upon multiplication, will always

\* yield a (2\*k)-bit product.

\* The return value is the prime number generated.

\* The algorithm used is explained in Section 4.44 of "Handbook of Applied

\* Cryptography" by Alfred Menezes, P.van Oorschot and S.Vanstone, Chapter 4,

\* pp.146, CRC Press, 1996.

\*/

private BigInteger GenerateProbablePrime(SecureRandom random, int keySize)

{

BigInteger mp\_n = BigInteger.ZERO ;

int index ;

/\* Set up a 128-byte random number into a byte array. \*/

byte[] seed = new byte[keySize / 8] ;

/\* To terminate a loop that tests for probable primality. \*/

boolean isProbablePrime ;

/\* Repeat the following steps until a probable prime is found. \*/

do

{

/\* Generate a 128-bit random seed into the byte array. \*/

random.nextBytes(seed) ;

/\* Set the top two bits of this seed equal to 1. \*/

seed[0] |= (byte) 0xc0 ;

/\* Ensure that the seed is odd. \*/

seed[keySize / 8 - 1] |= (byte) 0x01 ;

/\* Load this seed into a big-integer. \*/

mp\_n = new BigInteger(\_POSITIVE, seed) ;

/\* Now divide "mp\_n" by each of the primes from 2 up to \_MAXODDPRIME,

\* as stored in the array "oddPrimes". If the remainder is zero for any

\* one divisor, then "mp\_n" is not prime. \*/

isProbablePrime = true ;

for (index = 0; (index < \_NPRIMES) && (isProbablePrime); index++)

{

isProbablePrime = ((mp\_n.remainder(BigInteger.valueOf((long) oddPrimes[index])).intValue()) != 0) ;

}

if (isProbablePrime)

{

/\* The number is divisible by all primes from 2 up to \_MAXODDPRIME.

\* Now apply the Miller-Rabin test for primality. \*/

isProbablePrime = millerRabin(mp\_n, random) ;

}

} while (!(isProbablePrime)) ;

/\* The big integer generated, now tested to be probably prime, is

\* present in the variable "mp\_n". Return the same. \*/

return (mp\_n) ;

}

/\* This method returns three integers in an array: The modulus "n",

\* the public-key exponent "e" and the private-key exponent "d".

\* "keySize" is the size of "n" measured in \_bits\_, specified by

\* the caller.

\*/

private BigInteger[] keyGen(int keySize)

{

BigInteger[] retval = new BigInteger[3] ;

/\* Secure random number generation. \*/

SecureRandom random = new SecureRandom() ;

BigInteger mp\_n, mp\_p, mp\_q, mp\_phin, mp\_phin1, mp\_e, mp\_d ;

BigInteger mp\_three = new BigInteger("3") ;

/\* Randomly generate a prime "mp\_p" of half the key-size. \*/

mp\_p = GenerateProbablePrime(random, keySize/2) ;

/\* Randomly generate a second, \_different\_ prime "mp\_q" of

\* half the key-size. \*/

do

{

mp\_q = GenerateProbablePrime(random, keySize/2) ;

} while (mp\_q.compareTo(mp\_p) == 0) ;

/\* Multiply "mp\_p" and "mp\_q" to yield "mp\_n". Note that the top two

\* bits of "mp\_p" and "mp\_q" are set and therefore their product will

\* always be "keySize" bits long. Also, note that this product is odd.

\*/

mp\_n = mp\_p.multiply(mp\_q) ;

/\* Compute the Euler-Totient number "mp\_phin". Since "mp\_p"

\* and "mp\_q" are different, this number is equal to the

\* product (mp\_p-1) \* (mp\_q-1). Note that "mp\_phin" is even.

\*/

mp\_phin = (mp\_p.subtract(BigInteger.ONE)).multiply(mp\_q.subtract(BigInteger.ONE)) ;

/\* Now generate a random number "mp\_e", such that 3 <= "mp\_e" <= (mp\_phin-1),

\* and "mp\_e" is relatively prime to "mp\_phin" (that is, the GCD between

\* "mp\_e" and "mp\_phin" is not equal to one). \*/

mp\_phin1 = mp\_phin.subtract(BigInteger.ONE) ;

do

{

mp\_e = this.genRandom(mp\_three, mp\_phin1, random).setBit(0) ;

} while (mp\_e.gcd(mp\_phin).compareTo(BigInteger.ONE) != 0) ;

/\* Compute "mp\_d" equal to the multiplicative inverse of "mp\_e" modulo

\* "mp\_phin". This inverse exists, since "mp\_e" is relatively prime to

\* "mp\_phin". Moreover, (2 <= mp\_d <= mp\_phin-1). \*/

mp\_d = mp\_e.modInverse(mp\_phin) ;

/\* Package the three integers "mp\_n", "mp\_e" and "mp\_d" into

\* the return value. \*/

retval[0] = mp\_n ;

retval[1] = mp\_e ;

retval[2] = mp\_d ;

return (retval) ;

}

/\* This method generates a public-private key-pair having key-size "keySize"

\* bits. The public key is stored as a sequence of bytes in the file

\* having filename "pubKeyFile". The private key is stored encrypted

\* using a 16-character passphrase "passPhrase" in the file having

\* filename "pvtKeyFile".

\* Each of the two files is organized as a sequence of bytes, having length

\* equal to at most (2 \* keySize / 8) bytes. In each file, the first

\* (keySize / 8) bytes holds the modulus "mp\_n", while the remaining bytes

\* holds either the public-key exponent "mp\_e" or the private-key exponent

\* "mp\_d". Before each of these byte-sequences, their lengths are stored. \*/

public void keyGen(int keySize, String pubKeyFile, String pvtKeyFile, String passPhrase)

throws FileNotFoundException, IOException

{

int index ;

String tempFile = "temp.txt" ;

FileOutputStream publicKey = null, privateKey = null ;

/\* Generate an RSA public-private key-pair. \*/

BigInteger[] keys = keyGen(keySize) ;

/\* Convert the modulus, public and private keys into byte-forms. \*/

/\* Modulus. \*/

byte[] mp\_n = keys[0].toByteArray() ;

int modulus\_size = mp\_n.length ;

/\* Public-key exponent. \*/

byte[] mp\_e = keys[1].toByteArray() ;

int pubKey\_size = mp\_e.length ;

/\* Private-key exponent. \*/

byte[] mp\_d = keys[2].toByteArray() ;

int pvtKey\_size = mp\_d.length ;

/\* Write the public and private keys into the respective files. \*/

/\* Each byte array is preceded by its size in bytes (to be used while

\* retrieving the keys from the files). \*/

try

{

publicKey = new FileOutputStream(pubKeyFile) ;

try

{

privateKey = new FileOutputStream(tempFile) ;

publicKey.write(modulus\_size) ;

publicKey.write(mp\_n) ;

publicKey.write(pubKey\_size) ;

publicKey.write(mp\_e) ;

privateKey.write(modulus\_size) ;

privateKey.write(mp\_n) ;

privateKey.write(pvtKey\_size) ;

privateKey.write(mp\_d) ;

}

catch (IOException e)

{

System.out.println("Error opening private key file.") ;

}

finally

{

if (privateKey != null)

{

privateKey.close() ;

}

}

}

catch (IOException e)

{

System.out.println("Error opening public key file.") ;

}

finally

{

if (publicKey != null)

{

publicKey.close() ;

}

}

/\* The public key is stored in the file having name specified by

\* "pubKeyFile". \*/

/\* The private key is stored in \_clear\_ form in the file "temp.txt".

\* Encrypt this file using the passphrase. \*/

Aes aes = new Aes(passPhrase) ;

try

{

aes.encryptFile(tempFile, pvtKeyFile) ;

}

catch (FileNotFoundException e)

{

System.out.println("A file is not found.") ;

}

catch (IOException e)

{

System.out.println("A file has a problem.") ;

}

/\* Delete the clear form file containing the private key. \*/

File temp ;

temp = new File(tempFile) ;

temp.deleteOnExit() ;

}

/\* The following method encrypts a block of plaintext using the public

\* key. The public key is formed by the parameters "mp\_n" and "mp\_e".

\* "mp\_n" is the modulus of computation while "mp\_e" is the public-key

\* exponent.

\* It is assumed that the BigInteger equivalent of "plainText" lies between

\* 0 and (mp\_n-1). \*/

private byte[] encryptBlock(byte[] plainText, BigInteger mp\_n, BigInteger mp\_e)

{

/\* The size of the ciphertext block is equal to the size of the modulus. \*/

int index, index2 ;

byte[] cipherText ;

/\* First, marshall the plaintext block into a BigInteger. \*/

BigInteger mp\_x = new BigInteger(plainText) ;

/\* Compute: mp\_y = mp\_x to the power of mp\_e (modulo mp\_n). \*/

BigInteger mp\_y = mp\_x.modPow(mp\_e, mp\_n) ;

/\* Un-marshall the BigInteger into a byte form. \*/

cipherText = mp\_y.toByteArray() ;

/\* "cipherText" now contains the byte form of the encrypted text.

\* Return this byte form. \*/

return (cipherText) ;

}

/\* The following method decrypts a block of ciphertext using the private

\* key. The private key is formed by the parameters "mp\_n" and "mp\_d".

\* "mp\_n" is the modulus of computation while "mp\_d" is the private-key

\* exponent.

\* It is assumed that the BigInteger equivalent of "cipherText" lies between

\* 0 and (mp\_n-1). \*/

private byte[] decryptBlock(byte[] cipherText, BigInteger mp\_n, BigInteger mp\_d)

{

/\* The size of the plaintext block is equal to the size of the modulus. \*/

int index, index2 ;

byte[] plainText ;

/\* First, marshall the ciphertext block into a BigInteger. \*/

BigInteger mp\_y = new BigInteger(cipherText) ;

/\* Compute: mp\_z = mp\_y to the power of mp\_d (modulo mp\_n). \*/

BigInteger mp\_z = mp\_y.modPow(mp\_d, mp\_n) ;

/\* Un-marshall the BigInteger into a byte form. \*/

plainText = mp\_z.toByteArray() ;

/\* "plainText" now contains the byte form of the decrypted text.

\* Return this byte form. \*/

return (plainText) ;

}

/\* This method encrypts a block of plaintext using the public key stored

\* in the file specified by the file-name "pubKeyFile".

\* The file contains the pair (mp\_n, mp\_e) in that order, in byte form.

\* The first "\_KEYLENGTH" bits (\_KEYLENGTH / 8 bytes) in this file form the

\* modulus of the public key, "mp\_n". The remaining bytes form the exponent

\* "mp\_e". \*/

public byte[] encryptBlock(byte[] plainText, String pubKeyFile)

throws FileNotFoundException, IOException

{

int index, len ;

/\* Input binary file. \*/

FileInputStream pubKey = null ;

/\* Byte form of the modulus of the public key. \*/

byte[] pubKey\_modulus ;

/\* Byte form of the public key exponent. \*/

byte[] pubKey\_exponent ;

/\* Ciphertext block. \*/

byte[] cipherText = null ;

BigInteger mp\_n, mp\_e ;

try

{

/\* Open the input file. \*/

pubKey = new FileInputStream(pubKeyFile) ;

/\* The first byte serves as the byte-length of the modulus.

\* Read this first byte. \*/

len = pubKey.read() ;

/\* Read the first "len" bytes into the public key modulus, "mp\_n". \*/

pubKey\_modulus = new byte[len] ;

pubKey.read(pubKey\_modulus) ;

/\* The next byte serves as the byte-length of the public-key

\* exponent. Read this byte. \*/

len = pubKey.read() ;

/\* Read the remaining "len" bytes into the public key exponent,

\* "mp\_e". \*/

pubKey\_exponent = new byte[len] ;

pubKey.read(pubKey\_exponent) ;

/\* Convert the byte forms of the modulus and exponent into

\* BigInteger forms. \*/

mp\_n = new BigInteger(pubKey\_modulus) ;

mp\_e = new BigInteger(pubKey\_exponent) ;

/\* Encrypt the input plaintext block. \*/

cipherText = encryptBlock(plainText, mp\_n, mp\_e) ;

}

catch (FileNotFoundException e)

{

System.out.println("Public key file not found") ;

}

catch (IOException e)

{

System.out.println("Error reading public key file") ;

}

finally

{

if (pubKey != null)

{

pubKey.close() ;

}

}

/\* Return the encrypted block. \*/

return (cipherText) ;

}

/\* This method decrypts a block of ciphertext using the private key stored

\* in the file specified by the file-name "pvtKeyFile".

\* The file contains the pair (mp\_n, mp\_d) in that order, in byte form.

\* The first "\_KEYLENGTH" bits (\_KEYLENGTH / 8 bytes) in this file form the

\* modulus of the public key, "mp\_n". The remaining bytes form the exponent

\* "mp\_d".

\* The private key file is stored in an encrypted form, encrypted by the

\* AES algorithm using a 16-byte passphrase as the encryption key.

\*/

public byte[] decryptBlock(byte[] cipherText, String pvtKeyFile, String passPhrase)

throws FileNotFoundException, IOException

{

int len ;

/\* Input binary file. \*/

FileInputStream pvtKey = null ;

/\* Byte form of the modulus of the private key. \*/

byte[] pvtKey\_modulus ;

/\* Byte form of the private key exponent. \*/

byte[] pvtKey\_exponent ;

/\* Plaintext block. \*/

byte[] plainText = null ;

BigInteger mp\_n, mp\_d ;

/\* AES algorithm used to decrypt the private key. \*/

Aes aes = new Aes(passPhrase) ;

/\* Temporary filename. \*/

String decKeyFile = "tempDec.txt" ;

try

{

/\* First, decrypt the private key file using the AES algorithm into the

\* temporary file. \*/

aes.decryptFile(pvtKeyFile, decKeyFile) ;

/\* Open the decrypted private key file. \*/

pvtKey = new FileInputStream(decKeyFile) ;

/\* The first byte serves as the byte-length of the modulus.

\* Read this first byte. \*/

len = pvtKey.read() ;

/\* Read the first "len" bytes into the private key modulus, "mp\_n". \*/

pvtKey\_modulus = new byte[len] ;

pvtKey.read(pvtKey\_modulus) ;

/\* The next byte serves as the byte-length of the private-key

\* exponent. Read this byte. \*/

len = pvtKey.read() ;

/\* Read the remaining "len" bytes into the private key

\* exponent, "mp\_d". \*/

pvtKey\_exponent = new byte[len] ;

pvtKey.read(pvtKey\_exponent) ;

/\* Convert the byte forms of the modulus and exponent into

\* BigInteger forms. \*/

mp\_n = new BigInteger(pvtKey\_modulus) ;

mp\_d = new BigInteger(pvtKey\_exponent) ;

/\* Decrypt the input ciphertext block. \*/

plainText = decryptBlock(cipherText, mp\_n, mp\_d) ;

}

catch (FileNotFoundException e)

{

System.out.println("Private key file not found") ;

}

catch (IOException e)

{

System.out.println("Error reading private key file") ;

}

finally

{

if (pvtKey != null)

{

pvtKey.close() ;

}

}

/\* Delete the temporary private key file. \*/

File temp ;

temp = new File(decKeyFile) ;

temp.deleteOnExit() ;

/\* Return the decrypted block. \*/

return (plainText) ;

}

public static void main(String[] args)

{

String passPhrase = "abcdefghijklmnop" ;

byte[] plainText = {0x01, 0x02, 0x03, 0x04} ;

byte[] cipherText, decryptedText ;

int index ;

Rsa rsa = new Rsa() ;

try

{

rsa.keyGen(Rsa.\_KEYLENGTH, "pubKey.txt", "pvtKey.txt", passPhrase) ;

System.out.println("Plaintext block:") ;

for (index = plainText.length-1; index >= 0; index--)

{

System.out.printf("%02x ", plainText[index] );

}

System.out.println() ;

cipherText = rsa.encryptBlock(plainText, "pubKey.txt") ;

decryptedText = rsa.decryptBlock(cipherText, "pvtKey.txt", passPhrase) ;

System.out.println("Decrypted text block:") ;

for (index = decryptedText.length-1; index >= 0; index--)

{

System.out.printf("%02x ", decryptedText[index]);

}

System.out.println() ;

}

catch (FileNotFoundException e)

{

System.out.println("A file is not found.") ;

}

catch (IOException e)

{

System.out.println("A file has a problem.") ;

}

}

}

/\*

\* Program : MsgEnvelope.java

\* Author : Saleh Alsagoor

\* Date : 02-March-2009

\* Purpose : This program encapsulates a Message Envelope.

\* The envelope contains a ciphertext encrypted using a 16-byte

\* passphrase with the help of the AES-128 algorithm, with the

\* 16-byte passphrase encrypted using the recipient's public key.

\*/

import java.security.SecureRandom ;

import java.io.File ;

import java.io.FileInputStream ;

import java.io.FileOutputStream ;

import java.io.IOException ;

import java.io.FileNotFoundException ;

public class MsgEnvelope {

public static final int \_SECRETKEYLEN = Aes.\_KEYLENBYTES ;

public static final int \_EMEKLEN = 128 ;

public static final int \_BLOCKLEN = Aes.\_BLOCKLENBYTES ;

/\* This method creates an envelope that contains the plaintext file

\* "plainTextFile" encrypted using the AES with a randomly-generated

\* 16-byte string as the encryption key. The envelope also contains the

\* random string encrypted using the recipient's public key specified

\* by the filename "pubKeyFile". The envelope is finally stored

\* in the output file specified by "envFile". \*/

public static void createEnvelope(String envFile, String plainTextFile,

String publicKeyFile)

throws FileNotFoundException, IOException

{

/\* Loop variables. \*/

int index ;

/\* Temporary filename. \*/

String tempFile = "encOutFile.txt" ;

/\* Input and output file descriptors. \*/

FileInputStream inStream = null ;

FileOutputStream outStream = null ;

/\* Secure random number generation. \*/

SecureRandom random = new SecureRandom() ;

/\* Generate a random number of size 16 bytes for the

\* Message Encryption Key (MEK). \*/

byte[] mek = new byte[\_SECRETKEYLEN] ;

random.nextBytes(mek) ;

/\* Pad the message encryption key with the padding string

\* of "0x01". \*/

byte[] secretKey = new byte[mek.length+1] ;

secretKey[0] = (byte) 0x01 ;

for(index = 1; index < secretKey.length; index++)

{

secretKey[index] = mek[index-1] ;

}

try

{

/\* Encrypt the plaintext file into the temporary file. \*/

Aes aes = new Aes(mek) ;

aes.encryptFile(plainTextFile, tempFile) ;

/\* Encrypt the MEK using the public key to yield the

\* Encrypted Message Encryption Key (EMEK). \*/

Rsa rsa = new Rsa() ;

byte[] emek = rsa.encryptBlock(secretKey, publicKeyFile) ;

/\* Open the temporary file. \*/

inStream = new FileInputStream(tempFile) ;

/\* Open the message envelope file in overwrite mode. \*/

outStream = new FileOutputStream(envFile, false) ;

/\* Write the length of the EMEK into the message envelope

\* file. \*/

outStream.write(emek.length) ;

/\* Write the EMEK into the message envelope file. \*/

outStream.write(emek) ;

/\* Now repeatedly write out blocks of the encrypted file into

\* the envelope file. The block-length is equal to 16 bytes. \*/

byte[] inBlock = new byte[\_BLOCKLEN] ;

int balance = inStream.available() ;

while (balance > 0)

{

balance -= inStream.read(inBlock) ;

outStream.write(inBlock) ;

}

}

catch(FileNotFoundException e)

{

System.out.printf("Error opening input file not found: %n") ;

}

catch(IOException e)

{

System.out.printf("Input/output file exception.%n") ;

}

/\* Close output file. \*/

if (outStream != null)

{

outStream.close() ;

outStream = null ;

}

/\* Close input file. \*/

if (inStream != null)

{

inStream.close() ;

inStream = null ;

}

/\* Delete the temporary file. \*/

File temp ;

temp = new File(tempFile) ;

temp.deleteOnExit() ;

}

/\* This function retrieves the data file from a message envelope.

\* The envelope is stored in a file "envFile", encrypted with an

\* encrypted message encryption key (EMEK). The EMEK is first

\* retrieved using the private key stored in a file "privateKeyFile"

\* protected using a passphrase "passphrase". What is retrieved from

\* the EMEK is the Message Encryption Key (MEK), used to decrypt the

\* encrypted data file present in the envelope. The resulting plaintext

\* file is stored in the output file "plainTextFile". \*/

public static void retrieveData(String plainTextFile, String envFile,

String privateKeyFile, String passphrase)

throws FileNotFoundException, IOException

{

FileInputStream inFile = null ;

FileOutputStream outFile = null ;

FileOutputStream tempFile = null ;

String tempFileName = "encInFile.txt" ;

int index ; /\* Loop variable. \*/

byte[] inBlock = new byte[\_BLOCKLEN] ;

try

{

/\* First, open the message envelope file. \*/

inFile = new FileInputStream(envFile) ;

/\* The first byte of this file is the length of the

\* EMEK. Read this from the input file. \*/

int emekLen = inFile.read() ;

/\* Now read the EMEK from the input file. \*/

byte[] emek = new byte[emekLen] ;

inFile.read(emek) ;

/\* Decrypt the EMEK from this byte-sequence using

\* the passphrase-protected private key, to yield the MEK. \*/

Rsa rsa = new Rsa() ;

byte[] secretKey = rsa.decryptBlock(emek, privateKeyFile, passphrase) ;

/\* Note that the first byte of the MEK is the padding string

\* having value "0xff" and should be EXCLUDED. \*/

byte[] mek = new byte[secretKey.length-1] ;

for (index = mek.length-1; index >= 0; index--)

{

mek[index] = secretKey[index+1] ;

}

/\* The rest of the input file is the encrypted data.

\* Read this data into a temporary file. \*/

int inFileSize = inFile.available() ;

tempFile = new FileOutputStream(tempFileName, false) ;

for (index = 0; index < inFileSize / \_BLOCKLEN ; index++)

{

inFile.read(inBlock) ;

tempFile.write(inBlock) ;

}

tempFile.close() ;

/\* Decrypt the data in the temporary file using the MEK. \*/

Aes aes = new Aes(mek) ;

aes.decryptFile(tempFileName, plainTextFile) ;

}

catch (FileNotFoundException e)

{

System.out.printf("A file is not found.%n") ;

}

catch (IOException e)

{

System.out.printf("An input/output error has occurred.%n") ;

}

/\* Close files. \*/

if (outFile != null)

{

outFile.close() ;

}

if (inFile != null)

{

inFile.close() ;

}

/\* Delete the temporary file. \*/

File temp = new File(tempFileName) ;

temp.deleteOnExit() ;

}

public static void main(String[] args)

{

String pubKeyFile = "pubKey.txt" ;

String pvtKeyFile = "pvtKey.txt" ;

String passphrase = "abcdefghijklmnop" ;

String inFile = "CapstoneProposal.pdf" ;

String envFile = "envelope.dat" ;

String outFile = "OutputFile.pdf" ;

Rsa rsa = new Rsa() ;

try

{

rsa.keyGen(1024, pubKeyFile, pvtKeyFile, passphrase);

MsgEnvelope.createEnvelope(envFile, inFile, pubKeyFile) ;

MsgEnvelope.retrieveData(outFile, envFile, pvtKeyFile, passphrase);

}

catch(FileNotFoundException e)

{

System.out.printf("Error opening input file not found: %n") ;

}

catch(IOException e)

{

System.out.printf("Input/output file exception.%n") ;

}

}

}