Performance Evaluation of Three Encryption/Decryption Algorithms on the SunOS and Linux Operating Systems
Final Report

Turki Al-Somani 220106
Khalid Al-Zamil 913052

Submitted to
Dr. Kofahi

Department of Information and Computer Science
College of Computer Science and Engineering
King Fahad University of Petroleum and Minerals
Performance Evaluation of Three Encryption/Decryption Algorithms on the SunOS and Linux Operating Systems

Turki Al-Somani\textsuperscript{1} and Khalid Al-Zamil\textsuperscript{2}

\textsuperscript{1}Computer Engineering Department, King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia, tsomani@ccse.kfupm.edu.sa

\textsuperscript{2}Aramco, EXPEC Computing Center, PEASD, Dhahran 31311, Saudi Arabia, khalid.zamil@aramco.com

Abstract. This paper presents an implementation of three encryption algorithms and a comparison between them based on the CPU execution time. The CPU execution time can be broken down to kernel and user time. The selected algorithms are: DES, Triple-DES and Blowfish which are all symmetric block encryption algorithms. The objective of this research is to evaluate the performance of the three cryptography algorithms in terms of the processing time required in the kernel and user space for generating the secret key, encryption and decryption operations. The powerful portable programming language Java and JCA (Java Cryptography Architecture) is used in implementing the encryption algorithms. The performance of the implemented encryption algorithms will be evaluated on two operating systems namely: SunOS and Linux platforms. The results of this work will support the selection of the best encryption algorithm in terms of speed and will help in capacity planning of the overall system. The results show that the Blowfish algorithm is the fastest, followed by the DES algorithm then the Triple-DES algorithm. From the results obtained, we propose to either embed a security module within the processor to provide ultimate speeds for cryptography operations or to dedicate a special server which is responsible for providing cryptographic services for highly secured environments.

Keywords: DES, Triple-DES, Blowfish, Performance Evaluation, JCA & JCE
1 Introduction

Cryptography has a long and fascinating history [1] and has become an essential component of modern operating systems [2]. Cryptography provides the mechanisms necessary to provide accountability, accuracy and confidentiality of data. There are two general types of key-based encryption algorithms (also known as ciphers): symmetric (or conventional) and asymmetric (or public-key). Symmetric algorithms are designed in a way such that any two parties interested in encrypting/decrypting data have to use the same (secret) key generated for both encryption and decryption. Public-key algorithms follow a different approach. They are designed to allow for encrypting data using one key and decrypting it using another key. Public-key algorithms are much slower than Symmetric algorithms.

Symmetric algorithms can be divided into two categories: block ciphers [3] and stream ciphers [4]. The block ciphers operate on data in groups or blocks. On the other hand, stream ciphers only operate on a single bit at a time which makes them more suitable for real time applications such as multimedia.

There are many existing encryption ciphers that are already integrated into various operating systems. Several factors are behind the selection of a particular encryption algorithm such as its strength and immunity. The running time of the algorithm is also another factor that should be considered. Encryption algorithms may consume a huge amount of system resources for generating the secret key and for the actual work needed for encrypting or decrypting the data. If the running time of the algorithm is
neglected then this might lead to jeopardizing the overall performance of the operating system. Because of this reason, the primary objective of this paper is to evaluate the selected encryption algorithms based on their running time.

Three symmetric block ciphers have been selected for the work in this paper: Data Encryption Standard (DES) [5], Triple-DES and Blowfish [6]. These three symmetric block cipher have been chosen because DES has been a worldwide standard for 20 years, it has also has been standardized by ANSI and ISO. Triple-DES resolves the flaws in DES and is a much more complicated version. The blowfish is a cipher with a different structure and functionality.

The three symmetric block ciphers mentioned earlier will be implemented using Java and JCA. Java will enable the implemented program to be portable across platforms. A comparison of the processing time needed for both the kernel and the user to generate the secret key, encrypt and decrypt the data will be recorded. Section 2 will provide a literature survey on the related work. The three implementations are discussed in section 3. The experiment setup is mentioned in section 4. Section 5 will introduce the testing methodology and the results. Section 6 points to the future work and section 7 will finally conclude this work.
2 Related Work

Much research has been carried out to optimize and achieve the maximum performance of operating systems. The optimization techniques followed are for optimizing both the software and hardware. The researches are mainly benchmarks or are a comparison to other famous benchmarks [8, 9 & 40]. Other researches focused on the kernel’s point of view and have been categorized into different areas. Kernel performance evaluations have been explored in [11, 12, 14, 15 and 17]. Kernel monitoring and profiling was also addressed in [36]. Several performance analysis tools [27] and measurement techniques [25, 34] have been mentioned in the literature. Some of these measurement techniques were interested in measuring the hardware performance based on specific kernels such as [10]. Other studies targeted the best application optimization reachable by measuring the application performance [13, 16] or by optimizing the application [22-24 and 30].

Various efforts have also been accomplished in the area of operating system architecture and design [32 and 33] or concentrated on the overall system performance [18]. This type of research might be accomplished using an operating system simulator [39] or by using a normal PC operating system [26 & 35]. The outcomes of these studies have helped enormously in both fine-tuning the operating systems and the hardware needed [20, 21 & 28].

The importance, necessity and future of security in the web and e-commerce were best studied in [44]. Cryptography and its implementations in applied fields are exposed and explored in [47 & 53]. However, DES description, strength and attacks are in [41-43, 45, 48, 52 & 56]. The discussion in [46] covered multiple encryptions
security and performance. Finally blowfish structure and characteristics is well addressed in [54 & 55].

To the best of our knowledge, there has not been any work done in measuring the processing time for different encryption algorithms based on the CPU execution time. Table 1 summarizes the related work covered in the literature survey.

**Table 1: Related Work Categories**

<table>
<thead>
<tr>
<th>Categories</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Benchmarks</td>
<td>[8, 9 &amp; 40].</td>
</tr>
<tr>
<td>Kernel Performance Evaluations</td>
<td>[11, 12, 14, 15 and 17]</td>
</tr>
<tr>
<td>Kernel Monitoring and Profiling</td>
<td>[36]</td>
</tr>
<tr>
<td>Performance Analysis Tools</td>
<td>[27]</td>
</tr>
<tr>
<td>Measurement Techniques</td>
<td>[25 &amp; 34]</td>
</tr>
<tr>
<td>H/W Performance Analysis Based on Kernels</td>
<td>[10]</td>
</tr>
<tr>
<td>Application Performance</td>
<td>[13 &amp; 16]</td>
</tr>
<tr>
<td>Application Optimization</td>
<td>[22-24 &amp; 40]</td>
</tr>
<tr>
<td>OS Design &amp; Architecture</td>
<td>[32 &amp; 33]</td>
</tr>
<tr>
<td>System Performance</td>
<td>[18]</td>
</tr>
<tr>
<td>OS Simulation</td>
<td>[39]</td>
</tr>
<tr>
<td>PC OS</td>
<td>[26 &amp; 35]</td>
</tr>
<tr>
<td>Applied Cryptography</td>
<td>[44, 47 &amp; 53]</td>
</tr>
<tr>
<td>DES Description, Strength, Attacks</td>
<td>[5, 41-43, 45, 48, 52 &amp; 56]</td>
</tr>
<tr>
<td>Multiple Encryption Security &amp; Performance</td>
<td>[46]</td>
</tr>
<tr>
<td>Blowfish Structure &amp; Description</td>
<td>[54 &amp; 55]</td>
</tr>
</tbody>
</table>

3 Implementation

This section discusses the structure of the selected cipher algorithms, the implementations and gives an overview on Java Cryptography Architecture (JCA).
3.1 Cipher Algorithms

Three symmetric block ciphers have been selected for the work in this paper: (DES) [5], Triple-DES and Blowfish [6]. In DES, a 16 cycle Feistel system is used for encryption, with an overall 56-bit key permuted into 16 48-bit sub keys, one for each cycle. For decryption, the identical algorithm is used, but the order of sub keys is reversed. The left (L) and right (R) blocks are 32 bits each, yielding an overall block size of 64 bits. The hash function "f", specified by the standard using the so-called "S-boxes", takes a 32-bit data block and one of the 48-bit sub keys as input and produces 32 bits of output. Sometimes DES is said to use a 64-bit key, but 8 of the 64 bits are used only for parity checking, so the effective key size is 56 bits.

Triple-DES is a much more complicated version of DES and was developed to address the obvious flaws in DES without designing a whole new cryptosystem. Triple DES simply extends the key size of DES by applying the algorithm three times in succession with three different keys. The combined key size is thus 168 bits (3 times 56), beyond the reach of brute-force techniques.

Blowfish is a cipher with a different structure and functionality than the previously mentioned cipher algorithms. Blowfish is a variable-length key 64-bit block cipher. The algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several sub key arrays totaling 4168 bytes. Data encryption occurs via a 16-round Feistel network. Each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.
3.2 Implemented Program

The program implemented is constructed into 3 modules. The first module is responsible for generating secret keys to be used for encrypting and decrypting messages. A secret key has to be generated for each cipher algorithm. The second module is the encryption module that takes a secret key generated for the same cipher algorithm and uses this key to encrypt a message (e.g. a DES secret key to encrypt a DES message). The decryption module is the third module that takes a secret key and decrypts a message. The message will be decrypted successfully only if the key used for decryption is the same key used for encrypting the message initially (see Appendix A).

Designing the encryption/decryption program this way allowed for timing each operation (key generation, encryption and decryption) separately. The program also does not interact with the user while it is running (prompting where to save the secret key or the name of the file to encrypt), instead it expects all of the required parameters to be supplied when invoking the application. This is necessary to obtain accurate measurements of the time spent for each operation without being influenced by the time it took the user to interact with the program.

3.3 Java Cryptography Architecture (JCA)

The powerful portable programming language Java and JCA (Java Cryptography Architecture) is used in the implementations. JCA is a core API of the Java programming language and is designed to allow developers to incorporate both low-level and high-level security functionality into their programs [60, 61]. It encompasses the parts of the Java 2 SDK Security API related to cryptography, as well as a set of conventions and specifications. It also includes a "provider"
architecture that allows for multiple and interoperable cryptography implementations. The JCA was designed around 2 principles:

1. implementation independence and interoperability
2. algorithm independence and extensibility

Implementation independence and algorithm independence are complementary; cryptographic services can be used, such as digital signatures and message digests, without worrying about the implementation details or even the algorithms that form the basis for these concepts. When complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate a specific implementation. Implementation interoperability means that various implementations can work with each other, use each other's keys, or verify each other's signatures. This would mean, for example, that for the same algorithms, a key generated by one provider would be usable by another, and a signature generated by one provider would be verifiable by another. Algorithm extensibility means that new algorithms that fit in one of the supported engine classes can be added easily. The Java Cryptography Extension (JCE) extends the JCA API to provide a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms.

4 Experiment Setup

The selected encryption algorithms will be tested on 2 operating systems. The first test will be conducted on a Sun workstation with a SunOS 5.7 operating system and a CPU speed of 400 MHz with a total of 1024 MB of RAM. The second test will be performed on a PC running GNU/Linux version 2.4.18-14
with a Pentium III CPU that has a speed of 860 MHz and 256 MB of RAM (see Table 2 for a summary).

All the encryption and decryption operations will be performed using a 10 MB file. Java version 1.4.1 has been used for the implementing the 3 cipher algorithms. The encryption algorithms will be tested using a key size of 56 bits for the DES and Blowfish algorithms, a key size of 112 bits will be used for the Triple-DES algorithm.

Table 2: The Platforms Specifications

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Colonel</th>
<th>Swan</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>1024MB</td>
<td>256 MB</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>400 MHz</td>
<td>860.844 MHz Pentium III</td>
</tr>
<tr>
<td>Operating System</td>
<td>SunOS 5.7</td>
<td>GNU/Linux 2.4.18-14</td>
</tr>
</tbody>
</table>

5 Results

The elapsed (wall-clock or real) time counts everything such as disk or memory access, idle time, I/O, operating system overheads, etc. This gives a useful number but not for the comparison purpose we intend. The CPU executing time (or the CPU time) is the time the CPU spends computing a task and doesn’t consider the time spent waiting for I/O or running other programs. The CPU time could be further broken down to system (or kernel) time and user time. The system time is the time spent executing instructions in the kernel on behalf of the user program. The user time is the time spent executing instructions in the user program. For programs running on dedicated systems and spending most of their time doing computation, the elapsed time and CPU time should be approximately equal. Discrepancies between the CPU time and elapsed time will usually occur in cases where a program is paging (or swapping) due to insufficient memory on the system.
The `ptime` Sun utility was used to measure the elapsed time, system time and user time of the implemented program running on the SunOS platform (see Table 3). Similarly, the `time` Linux utility was used to record the results of the implemented program running on the Linux platform (see Table 4). To record the overhead introduced by the Java Virtual Machine (JVM), a program was implemented to start and stop the JVM directly. This allowed for obtaining the actual time spent in initializing and freeing the JVM. Table 5 contains the kernel and user time recorded for invoking the JVM.

Table 3: Algorithms processing time in seconds on SunOS

<table>
<thead>
<tr>
<th></th>
<th>DES</th>
<th></th>
<th></th>
<th>Triple DES</th>
<th></th>
<th></th>
<th>Blowfish</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
</tr>
<tr>
<td>Key Generation</td>
<td>5.569</td>
<td>0.303</td>
<td>6.450</td>
<td>5.721</td>
<td>0.220</td>
<td>6.063</td>
<td>5.514</td>
<td>0.211</td>
<td>5.842</td>
</tr>
<tr>
<td>Decryption</td>
<td>25.17</td>
<td>1.174</td>
<td>26.499</td>
<td>36.942</td>
<td>1.293</td>
<td>38.450</td>
<td>12.480</td>
<td>0.985</td>
<td>13.655</td>
</tr>
</tbody>
</table>

Table 4: Algorithms processing time in seconds on Linux

<table>
<thead>
<tr>
<th></th>
<th>DES</th>
<th></th>
<th></th>
<th>Triple DES</th>
<th></th>
<th></th>
<th>Blowfish</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
<td>User</td>
<td>Kernel</td>
<td>Real</td>
</tr>
<tr>
<td>Key Generation</td>
<td>2.925</td>
<td>0.087</td>
<td>3.69</td>
<td>2.851</td>
<td>0.080</td>
<td>3.02</td>
<td>3.062</td>
<td>0.060</td>
<td>4.13</td>
</tr>
<tr>
<td>Encryption</td>
<td>6.533</td>
<td>0.603</td>
<td>8.77</td>
<td>12.007</td>
<td>0.488</td>
<td>14.12</td>
<td>5.115</td>
<td>0.496</td>
<td>7.09</td>
</tr>
<tr>
<td>Decryption</td>
<td>6.669</td>
<td>0.431</td>
<td>8.91</td>
<td>12.156</td>
<td>0.488</td>
<td>15.69</td>
<td>5.195</td>
<td>0.515</td>
<td>7.95</td>
</tr>
</tbody>
</table>

Table 5: Java Virtual Machine Overheads in seconds

<table>
<thead>
<tr>
<th></th>
<th>User</th>
<th>Kernel</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun OS</td>
<td>0.411</td>
<td>0.100</td>
<td>0.531</td>
</tr>
<tr>
<td>Linux</td>
<td>0.435</td>
<td>0.021</td>
<td>0.50</td>
</tr>
</tbody>
</table>
Most encryption algorithms were designed to be implemented in hardware. However, implementing encryption algorithms in software have a tremendous impact on the CPU resulting in degrading its overall performance. Table 6 & 7 show the CPU executing time for the selected algorithms on the SunOS and Linux platforms. The results illustrated in Figure 1 & 2 show that the key generation operation for all algorithms are almost identical. The major difference between these algorithms is the CPU time spent while performing the encryption and decryption operations. It is clear, as illustrated in the figures, that the Blowfish algorithm is the fastest algorithm due to the mechanism it follows for ciphering data. The DES algorithm comes in second place followed by the Triple-DES algorithm. It is obvious that because of the increased complexity incorporated within the Triple-DES, which is needed to address the security issues that exist in the DES algorithm, is degrading the performance in terms of the CPU time.

<table>
<thead>
<tr>
<th></th>
<th>DES</th>
<th>Triple-DES</th>
<th>Blowfish</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Generation</strong></td>
<td>5.872</td>
<td>5.941</td>
<td>5.725</td>
</tr>
<tr>
<td><strong>Encryption</strong></td>
<td>25.073</td>
<td>39.096</td>
<td>13.429</td>
</tr>
<tr>
<td><strong>Decryption</strong></td>
<td>26.344</td>
<td>38.235</td>
<td>13.465</td>
</tr>
</tbody>
</table>

**Table 6**: CPU execution time in seconds (Sun OS)

<table>
<thead>
<tr>
<th></th>
<th>DES</th>
<th>Triple-DES</th>
<th>Blowfish</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Generation</strong></td>
<td>3.012</td>
<td>2.931</td>
<td>3.122</td>
</tr>
<tr>
<td><strong>Encryption</strong></td>
<td>7.136</td>
<td>12.495</td>
<td>5.611</td>
</tr>
<tr>
<td><strong>Decryption</strong></td>
<td>7.1</td>
<td>12.644</td>
<td>5.71</td>
</tr>
</tbody>
</table>

**Table 7**: CPU execution time in seconds (Linux)
The results indicate that on the SunOS platform the Blowfish algorithm is faster than the DES algorithm by 87% for encryption and 82% for decryption. The DES algorithm is 56% faster than the Triple-DES for encryption and 45% faster for...
decryption. For the results obtained on the Linux platform, the Blowfish algorithm is faster than DES algorithm by 27% when encrypting and 24% for decrypting. The DES algorithm is 75% faster than the Triple-DES algorithm for encryption and 78% faster when decrypting.

In order to increase the speed of these algorithms we propose to have a built in cipher module within the processor. This will eliminate any overheads introduced when implementing encryption algorithms in software. Another proposal is to have a dedicated server responsible for secret key generation and distribution, encryption, decryption, authentication … etc. This approach is very useful for highly secured environments (e.g. the military).

6 Future Work

Two interesting ciphers will be under investigation. The first cipher is one of the COS cipher family (COS standing for Crossing Over Systems). This acronym refers to how each cipher internally works. COS ciphers are block ciphers, built only from stream ciphers primitives (nonlinear feedback shift registers and Boolean functions). These ciphers have been especially designed to yield a very high encryption security and a very high encryption speed with an internal secret 256-bit key [58].

The second is the elliptic curves cryptography [59]. Elliptic Curve Cryptosystem (ECC), which was originally proposed by Niel Koblitz and Victor Miller in 1985, is seen as a serious alternative to RSA with a much shorter world length. ECC with a key size of 128-256 bits is shown to offer equal security to that of RSA with key size of 1-2Kbits. To date, no significant breakthroughs have been made in determining weaknesses in the ECC algorithm, which is based on the discrete logarithm problem over points on an elliptic curve.
In terms of the implementation, we are planning to use both “C” as a high level programming language and Assembly as a low level machine language which will enable us to optimize computation intensive parts of the ciphers. The implementation will also include dedicated types of hardware that will provide at the hardware level random number generator and built-in co-processors.

7 Conclusion

In this paper we presented an implementation of three symmetric block encryption algorithms using Java and JCA. The main objective was to evaluate the performance of these algorithms in terms of CPU execution time. The measurements were performed on two platforms; SunOS and Linux. The analyzed time was the CPU execution time for generating the secret key, encryption and decryption on a 10MB file. The results showed that the Blowfish algorithm was the fastest algorithm followed by the DES algorithm then the Triple-DES algorithm. The Triple-DES algorithm was slow in its performance due to the added complexity and security it has over the DES algorithm.

The results obtained play a major role on selecting the appropriate encryption algorithm to use for software implementations. The proposal in this paper supports the idea of using a built in module within the processor dedicated for security considerations. This will have the advantage of relieving the processor from the overhead associated with implementing encryption algorithms in software. Having a built in module within the processor will provide speed for encryption algorithms used in e-commerce and m-commerce which will be heavily demanded in the near future. Using a dedicated server was also proposed in this paper. This server will have the responsibility of controlling and performing all security tasks such as key
generation and distribution, encryption and decryption. This server will prove beneficial in highly secured environments.

Acknowledgements

We would like to thank and express our gratitude to Dr. Najib Kofahi for his valuable suggestions, guidance and help that improved the overall quality of this paper.

References

[14] Schmidt, H. and Bond, J.W., “Performance analyses of the power-domain kernel adaptive locally optimum processor (ALOP) against interference waveforms with continuous power density”, MILCOM 97 Proceedings , Volume: 1 , 2-5 Nov 1997 Page(s): 79 -83 vol.1


public class EncryptDecrypt {
  final int GENERATE_KEY = 1;
  final int ENCRYPT = 2;
  final int DECRYPT = 3;
  String algorithm;
  int operation;
  Cipher cipher;
  CipherInputStream cipherInputStream;
  Key key;
  String fileName;

  public EncryptDecrypt(String algorithm, int operation) {
    this.algorithm = algorithm;
    this.operation = operation;
  }

  public void start() {
    // long time = System.currentTimeMillis();
    
    if (operation == GENERATE_KEY) {
      generateKey();
    } else if (operation == ENCRYPT) {
      encrypt();
    } else {
      decrypt();
    }

    // System.out.println("Total Execution Time = " + (System.currentTimeMillis()-time));
  }

  // encrypt the file using the key in the directory
  private void encrypt() {
    // load the secret key for encrypting the message, note that it should be
    // the same key used for decrypting the message
    loadKey();

    try {
      // generate a Cipher object for encoding using the algorithm supplied
      cipher = Cipher.getInstance(algorithm);
      // System.out.println("EncryptDecrypt.encrypt");
      // initialize the Cipher with the key
      cipher.init(Cipher.ENCRYPT_MODE, key);

      // creating the encrypted cipher stream
      File file = new File(fileName);
      FileInputStream fis = new FileInputStream(file);
      cipherInputStream = new CipherInputStream(fis, cipher);
    } catch (NoSuchAlgorithmException e) {
      // Handle exception
    }
  }

  // decrypt the file using the key in the directory
  private void decrypt() {
    // load the secret key for decrypting the message
    loadKey();

    try {
      // generate a Cipher object for decoding using the algorithm supplied
      cipher = Cipher.getInstance(algorithm);
      // System.out.println("EncryptDecrypt.decrypt");
      // initialize the Cipher with the key
      cipher.init(Cipher.DECRYPT_MODE, key);

      // creating the decrypted cipher stream
      FileInputStream fis = new FileInputStream(fileName);
      cipherInputStream = new CipherInputStream(fis, cipher);
    } catch (NoSuchAlgorithmException e) {
      // Handle exception
    }
  }

  // generate a key
  private void generateKey() {
    try {
      SecretKey secretKey = KeyGenerator.getInstance(algorithm).generateKey();
      key = secretKey;
      System.out.println("EncryptDecrypt.generateKey");
    } catch (NoSuchAlgorithmException e) {
      // Handle exception
    }
  }

  // load the secret key from the directory
  private void loadKey() {
    try {
      SecretKeySpec secretKeySpec = new SecretKeySpec(key.getEncoded(), algorithm);
      cipher.init(Cipher.DECRYPT_MODE, secretKeySpec);
      System.out.println("EncryptDecrypt.loadKey");
    } catch (NoSuchAlgorithmException e) {
      // Handle exception
    }
  }
}
// writing the encrypted data to the output file
file = new File(fileName.concat(".encrypted"));
FileOutputStream fos = new FileOutputStream(file);
byte[] b2 = new byte[1024];

// read 1024 byte chunks of data from the encryption stream
// and write them to the output stream (the output file)
int i2 = cipherInputStream.read(b2);
while (i2 != -1) {
    fos.write(b2, 0, i2);
    i2 = cipherInputStream.read(b2);
}

// close all the streams
fis.close();
fos.close();
cipherInputStream.close();
}

try {
    // load the secret key for decrypting the message, note that it should be
    // the same key used for encrypting the message
    loadKey();

    try {
        // generate a Cipher object for decoding using the algorithm supplied
        cipher = Cipher.getInstance(algorithm);

        // initialize the Cipher object
        cipher.init(Cipher.DECRYPT_MODE, key);

        // Open the encrypted file
        String tempFileName = fileName.replaceFirst(".encrypted", "");
        File file = new File(tempFileName);
        FileInputStream fis = new FileInputStream(file);
        cipherInputStream = new CipherInputStream(fis, cipher);

        // writing the decrypted data to the output file
        fileName = fileName.replaceAll(".encrypted", "");
        FileOutputStream fos = new FileOutputStream(fileName);

        // read 1024 byte chunks of data from the decryption stream
        // and write them to the output stream (the output file)
        byte[] b2 = new byte[1024];
        int i2 = cipherInputStream.read(b2);
        while (i2 != -1) {
            fos.write(b2, 0, i2);
            i2 = cipherInputStream.read(b2);
        }

        // close all the streams
        fis.close();
fos.close();
cipherInputStream.close();
}

} catch (Exception ex) {
System.out.println("Exception: " + ex.getMessage());
}

// generate a secret key for encryption
private void generateKey() {
    try {
        // generate a keyGenerator object for this algorithm
        KeyGenerator KG = KeyGenerator.getInstance(algorithm);
        // System.out.println("Using Algorithm "+KG.getAlgorithm());
        // generate a key from the keyGenerator for the algorithm supplied
        key = KG.generateKey();
        // save the generated key to a file for future use
        saveKey();
    } catch (NoSuchAlgorithmException ex) {
        System.out.println("Exception: " + ex.getMessage());
    } catch (Exception e) {
        System.out.println("Serialization Exception: " + e.getMessage());
        e.printStackTrace();
    }
    // save the secret key to the file
    private void saveKey() {
        try {
            // create a file then write the encoded key to it
            File f = new File("./secretKey.dat");
            FileOutputStream fout = new FileOutputStream(f);
            fout.write(key.getEncoded());
            fout.flush();
            fout.close();
            // System.out.println("Stored the Secret key");
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
            e.printStackTrace();
        }
    }
    // load the secret key from the file
    private void loadKey() {
        try {
            // read the secret key from file
            FileInputStream fin = new FileInputStream(f);
            byte[] decodedKeyC = new byte[fin.available()];
            fin.read(decodedKeyC);
            fin.close();
            // create the secret key with the algorithm type supplied
            SecretKeySpec desKeySpec = new SecretKeySpec(decodedKeyC, algorithm);
            key = (javax.crypto.SecretKey) desKeySpec;
            // System.out.println("Loaded the Secret key");
        } catch (IOException e) {
            System.out.println("IOException: " + e.getMessage());
            e.printStackTrace();
        }
    }
    public void setFileName(String name)
    {
        this.fileName = name;
    }
    public static void main(String args[])
    {
    }
int operation = 0;
String algorithm;

if (args.length < 2) {
    printUsage();
    return;
}

// check if the supplied algorithm name is supported
algorithm = args[0];
if ((!algorithm.equals("DES") && (!algorithm.equals("Blowfish")) && (!algorithm.equals("TripleDES"))) {
    System.out.println("The ALGORITHM supplied is not a valid algorithm name!");
    printUsage();
    return;
}  
try {
    operation = (new Integer(args[1])).intValue();
} catch (NumberFormatException ex) {
    System.out.println("The OPERATION supplied is not a valid operation ");
    printUsage();
    return;
}

// check for the existance of the required parameters
if ((operation > 3) || (operation < 1)) {
    System.out.println("The OPERATION supplied is not a valid operation ");
    printUsage();
    return;
} else if ((args.length != 2) && (operation == 1)) {
    System.out.println("Not valid Parameters");
    printUsage();
    return;
} else if ((args.length != 3) && ((operation == 2) || (operation == 3))) {
    System.out.println("Not valid Parameters");
    printUsage();
    return;
}

// create the class for the benchmarking of the different encryption algorithms
// as well to the different operations (key generation, encryption and decryption)
EncryptDecrypt encryptor = new EncryptDecrypt(algorithm, operation);

// pass the input file name to be modified slightly for the output file
if (operation > 1) {
    encryptor.setFileName(args[2]);
}

// start working!
encryptor.start();

public static void printUsage() {
    System.out.println("Usage: java EncryptDecrypt <ALGORITHM> <OPERATION>");
    System.out.println("
<ALGORITHM> = DES, TripleDES, Blowfish*");
    System.out.println("<OPERATION> = 1 ==> Generate Secret Key");
    System.out.println("<OPERATION> = 2 ==> Encrypt");
    System.out.println("<OPERATION> = 3 ==> Decrypt");
}