Source Code
ProtectStar™ Extended AES Algorithm

Block-Size: 512-bit (64-byte)
Key Sizes: 128, 256 and 512-bit (16, 32 and 64 byte) (*default 256-bit*)
Modes of Operation: ECB, CBC, CFB, OFB and CTR (*default CTR*)
Total Round Number: 24

```
package com.crypt;


import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import com.crypt.debug.Debug;
import com.crypt.test.*;


/**
 * This class implements the core algorithm of the AES Extended for
generating round keys, encrypting
 * and decrypting data in blocks by applying AES-specific methods like
round key addition, sbox lookups,
 * shift-rows and mix-column operations.
 *
 * @ProtectStar, Inc.
 * @ver 2.0
 *
 *
 */
public class AESExtension_Algorithm {

private static final byte[] SBOX = {
(byte)99, (byte)124, (byte)119, (byte)123, (byte)242, (byte)107, (byte)111,
(byte)197,  (byte)48,   (byte)1, (byte)103,  (byte)43, (byte)254,
(byte)215, (byte)171, (byte)118,
       (byte)202, (byte)130, (byte)201, (byte)125, (byte)250,  (byte)89,
(byte)71, (byte)240, (byte)173, (byte)212, (byte)162, (byte)175, (byte)156,
(byte)164, (byte)114, (byte)192,
       (byte)183, (byte)253, (byte)147,  (byte)38,  (byte)54,  (byte)63,
(byte)247, (byte)204,  (byte)52, (byte)165, (byte)229, (byte)241,
(byte)113, (byte)216,  (byte)49,  (byte)21,
         (byte)4, (byte)199,  (byte)35, (byte)195,  (byte)24, (byte)150,
(byte)5, (byte)154,   (byte)7,  (byte)18, (byte)128, (byte)226, (byte)235,
(byte)39, (byte)178, (byte)117,
         (byte)9, (byte)131,  (byte)44,  (byte)26,  (byte)27, (byte)110,
(byte)90, (byte)160,  (byte)82,  (byte)59, (byte)214, (byte)179,  (byte)41,
(byte)227,  (byte)47, (byte)132,
```

```
        (byte)83, (byte)209,   (byte)0, (byte)237,  (byte)32, (byte)252,
(byte)177,  (byte)91, (byte)106, (byte)203, (byte)190,  (byte)57,
(byte)74,  (byte)76,  (byte)88, (byte)207,
       (byte)208, (byte)239, (byte)170, (byte)251,  (byte)67,  (byte)77,
(byte)51, (byte)133,  (byte)69, (byte)249,   (byte)2, (byte)127,  (byte)80,
(byte)60, (byte)159, (byte)168,
        (byte)81, (byte)163,  (byte)64, (byte)143, (byte)146, (byte)157,
(byte)56, (byte)245, (byte)188, (byte)182, (byte)218,  (byte)33,  (byte)16,
(byte)255, (byte)243, (byte)210,
       (byte)205,  (byte)12,  (byte)19, (byte)236,  (byte)95, (byte)151,
(byte)68,  (byte)23, (byte)196, (byte)167, (byte)126,  (byte)61, (byte)100,
(byte)93,  (byte)25, (byte)115,
        (byte)96, (byte)129,  (byte)79, (byte)220,  (byte)34,  (byte)42,
(byte)144, (byte)136,  (byte)70, (byte)238, (byte)184,  (byte)20,
(byte)222,  (byte)94,  (byte)11, (byte)219,
       (byte)224,  (byte)50,  (byte)58,  (byte)10,  (byte)73,   (byte)6,
(byte)36,  (byte)92, (byte)194, (byte)211, (byte)172,  (byte)98, (byte)145,
(byte)149, (byte)228, (byte)121,
       (byte)231, (byte)200,  (byte)55, (byte)109, (byte)141, (byte)213,
(byte)78, (byte)169, (byte)108,  (byte)86, (byte)244, (byte)234, (byte)101,
(byte)122, (byte)174,   (byte)8,
       (byte)186, (byte)120,  (byte)37,  (byte)46,  (byte)28, (byte)166,
(byte)180, (byte)198, (byte)232, (byte)221, (byte)116,  (byte)31,
(byte)75, (byte)189, (byte)139, (byte)138,
       (byte)112,  (byte)62, (byte)181, (byte)102,  (byte)72,   (byte)3,
(byte)246,  (byte)14,  (byte)97,  (byte)53,  (byte)87, (byte)185,
(byte)134, (byte)193,  (byte)29, (byte)158,
       (byte)225, (byte)248, (byte)152,  (byte)17, (byte)105, (byte)217,
(byte)142, (byte)148, (byte)155,  (byte)30, (byte)135, (byte)233,
(byte)206,  (byte)85,  (byte)40, (byte)223,
       (byte)140, (byte)161, (byte)137,  (byte)13, (byte)191, (byte)230,
(byte)66, (byte)104,  (byte)65, (byte)153,  (byte)45,  (byte)15, (byte)176,
(byte)84, (byte)187,  (byte)22,
   };

   private static final byte[] SBOX_Inverse = {
        (byte)82,   (byte)9, (byte)106, (byte)213,  (byte)48,  (byte)54,
(byte)165,  (byte)56, (byte)191,  (byte)64, (byte)163, (byte)158,
(byte)129, (byte)243, (byte)215, (byte)251,
       (byte)124, (byte)227,  (byte)57, (byte)130, (byte)155,  (byte)47,
(byte)255, (byte)135,  (byte)52, (byte)142,  (byte)67,  (byte)68,
(byte)196, (byte)222, (byte)233, (byte)203,
        (byte)84, (byte)123, (byte)148,  (byte)50, (byte)166, (byte)194,
(byte)35,  (byte)61, (byte)238,  (byte)76, (byte)149,  (byte)11,  (byte)66,
(byte)250, (byte)195,  (byte)78,
         (byte)8,  (byte)46, (byte)161, (byte)102,  (byte)40, (byte)217,
(byte)36, (byte)178, (byte)118,  (byte)91, (byte)162,  (byte)73, (byte)109,
(byte)139, (byte)209,  (byte)37,
       (byte)114, (byte)248, (byte)246, (byte)100, (byte)134, (byte)104,
(byte)152,  (byte)22, (byte)212, (byte)164,  (byte)92, (byte)204,
(byte)93, (byte)101, (byte)182, (byte)146,
       (byte)108, (byte)112,  (byte)72,  (byte)80, (byte)253, (byte)237,
(byte)185, (byte)218,  (byte)94,  (byte)21,  (byte)70,  (byte)87,
(byte)167, (byte)141, (byte)157, (byte)132,
       (byte)144, (byte)216, (byte)171,   (byte)0, (byte)140, (byte)188,
(byte)211,  (byte)10, (byte)247, (byte)228,  (byte)88,   (byte)5,
(byte)184, (byte)179,  (byte)69,   (byte)6,
       (byte)208,  (byte)44,  (byte)30, (byte)143, (byte)202,  (byte)63,
(byte)15,   (byte)2, (byte)193, (byte)175, (byte)189,   (byte)3,   (byte)1,
(byte)19, (byte)138, (byte)107,
```

```
        (byte)58, (byte)145,  (byte)17,  (byte)65,  (byte)79, (byte)103,
(byte)220, (byte)234, (byte)151, (byte)242, (byte)207, (byte)206,
(byte)240, (byte)180, (byte)230, (byte)115,
       (byte)150, (byte)172, (byte)116,  (byte)34, (byte)231, (byte)173,
(byte)53, (byte)133, (byte)226, (byte)249,  (byte)55, (byte)232,  (byte)28,
(byte)117, (byte)223, (byte)110,
        (byte)71, (byte)241,  (byte)26, (byte)113,  (byte)29,  (byte)41,
(byte)197, (byte)137, (byte)111, (byte)183,  (byte)98,  (byte)14,
(byte)170,  (byte)24, (byte)190,  (byte)27,
       (byte)252,  (byte)86,  (byte)62,  (byte)75, (byte)198, (byte)210,
(byte)121,  (byte)32, (byte)154, (byte)219, (byte)192, (byte)254,
(byte)120, (byte)205,  (byte)90, (byte)244,
        (byte)31, (byte)221, (byte)168,  (byte)51, (byte)136,   (byte)7,
(byte)199,  (byte)49, (byte)177,  (byte)18,  (byte)16,  (byte)89,
(byte)39, (byte)128, (byte)236,  (byte)95,
        (byte)96,  (byte)81, (byte)127, (byte)169,  (byte)25, (byte)181,
(byte)74,  (byte)13,  (byte)45, (byte)229, (byte)122, (byte)159, (byte)147,
(byte)201, (byte)156, (byte)239,
       (byte)160, (byte)224,  (byte)59,  (byte)77, (byte)174,  (byte)42,
(byte)245, (byte)176, (byte)200, (byte)235, (byte)187,  (byte)60,
(byte)131,  (byte)83, (byte)153,  (byte)97,
        (byte)23,  (byte)43,   (byte)4, (byte)126, (byte)186, (byte)119,
(byte)214,  (byte)38, (byte)225, (byte)105,  (byte)20,  (byte)99,
(byte)85,  (byte)33,  (byte)12, (byte)125,
       };


   private int[] RCON=new int[100]; // maximum required RCON size is around
100

      protected static final int BLOCKSIZE= 64; // Block Size in bytes (=
512 Bits)
      private final int COLUMNSIZE = 16;

      private int NK; // key size in word

      private final int BLOCKSIZE_WORD= BLOCKSIZE / 4; // Block Size in
words

      private int KEYSIZE;

      //    total round numbers for 512-bit Block Size (22 rounds is due to
the block-size + 2 extra rounds due to the shifting operation)
      private final int TOTALROUNDNUMBER = 24;

      private byte[] originalKey;

      private byte[] roundKeys= new byte[(TOTALROUNDNUMBER+1)*BLOCKSIZE];


      /**
       * Constructor
       *
       * @param keySize 16, 32 and 64 bytes are the possible key sizes
       */
      public AESExtension_Algorithm(int keySize) {
            KEYSIZE=keySize;
            NK = keySize / 4;
            originalKey = new byte[KEYSIZE];
            loadRCON();
      }
```

```
	/**
	 * Generates the elements of RCON array which is used in the round
key generation process.
	 *
	 */
	private void loadRCON() {

		int r=1;
		RCON[0]= r << 24;

		for (int i=1; i< RCON.length; i++){
			r <<=1;
			if (r>=0x100) r ^=0x11B;
			RCON[i]= r << 24;
		}
	}


	/**
	 * Encrypts a single block with the size 64-bytes
	 *
	 * @param theBlock the bytes to be encrypted
	 *
	 */
	protected void encBlockProcess(byte[] theBlock) {

		// Initial Round

		addRoundKey(theBlock,roundKeys,0);

		for (int i=0;i<TOTALROUNDNUMBER-1;i++) {

			SBOX_Substitution(theBlock);

			shiftRows(theBlock,true);

			mixColumns(theBlock);

			addRoundKey(theBlock,roundKeys,(i+1)*BLOCKSIZE);
		}

		// last finalization round without mix column operation

		SBOX_Substitution(theBlock);
		shiftRows(theBlock,true);
		addRoundKey(theBlock,roundKeys,roundKeys.length-BLOCKSIZE);

	}

	/**
	 * Decrypts a single block with the size 64-bytes
	 *
	 * @param cipherBlock the bytes to be decrypted
	 */
	protected void decBlockProcess(byte[] cipherBlock) {

		// Initial Round

		addRoundKey(cipherBlock,roundKeys,roundKeys.length-BLOCKSIZE);
```

```
            for (int i=0;i<TOTALROUNDNUMBER-1;i++) {

                  shiftRows(cipherBlock,false);

                  Inv_SBOX_Substitution(cipherBlock);

                  addRoundKey(cipherBlock,roundKeys,roundKeys.length-
(i+2)*BLOCKSIZE);

                  Inv_mixColumns(cipherBlock);


            }

            // last finalization round without mix column operation

            shiftRows(cipherBlock,false);

            Inv_SBOX_Substitution(cipherBlock);

            addRoundKey(cipherBlock,roundKeys,0);

      }
```

```
      /**
       * Enforces the mix-column operation for encryption process specified
in the official AES Algorithm.
       *
       * @param theBlock the bytes on which mix-column operation is applied
       *
       */
      private void mixColumns(byte[] theBlock) {

            byte b2 = (byte)0x02;
          byte b3 = (byte)0x03;

            int sp[] = new int[4];

          for (int i = 0; i < COLUMNSIZE; i++) {
              sp[0] = byteMultiplication(b2, theBlock[i]) ^
byteMultiplication(b3, theBlock[COLUMNSIZE+i]) ^ theBlock[2*COLUMNSIZE+i]
^ theBlock[3*COLUMNSIZE+i];
              sp[1] = theBlock[i]  ^ byteMultiplication(b2,
theBlock[COLUMNSIZE+i]) ^ byteMultiplication(b3, theBlock[2*COLUMNSIZE+i])
^ theBlock[3*COLUMNSIZE+i];
              sp[2] = theBlock[i]  ^ theBlock[COLUMNSIZE+i]  ^
byteMultiplication(b2, theBlock[2*COLUMNSIZE+i]) ^ byteMultiplication(b3,
theBlock[3*COLUMNSIZE+i]);
              sp[3] = byteMultiplication(b3, theBlock[i]) ^
theBlock[COLUMNSIZE+i]  ^ theBlock[2*COLUMNSIZE+i]  ^
byteMultiplication(b2, theBlock[3*COLUMNSIZE+i]);
```

```
                for (int j = 0; j< 4; j++) theBlock[j*COLUMNSIZE+i] = (byte)
(sp[j]);
            }
        }

        /**
         * Multiplies two byte values (required by the mix-column operation).
         *
         * @param x the first byte value
         * @param y the second byte value
         * @return the multiplication result as byte value
         */
        private byte byteMultiplication(byte x, byte y) {

              byte aa = x, bb = y;
              byte result = 0;
              byte temp;

              while (aa != 0) {
                 if ((aa & 1) != 0)
                    result = (byte)(result ^ bb);

                 temp = (byte)(bb & 0x80);
                 bb = (byte)(bb << 1);

                 if (temp != 0) bb = (byte)(bb ^ 0x1b);

                 aa = (byte)((aa & 0xff) >> 1);
              }
              return result;
        }

        /**
         * Enforces the inverse mix-column operation for decryption process
specified in the official AES Algorithm.
         *
         * @param cipherBlock the bytes on which inverse mix-column operation
is applied
         */

        private void Inv_mixColumns(byte[] cipherBlock) {

              int[] sp = new int[4];
              byte b0b = (byte)0x0b;
              byte b0d = (byte)0x0d;
              byte b09 = (byte)0x09;
              byte b0e = (byte)0x0e;

              for (int i = 0; i < COLUMNSIZE; i++) {
                 sp[0] = byteMultiplication(b0e, cipherBlock[i]) ^
byteMultiplication(b0b, cipherBlock[COLUMNSIZE+i]) ^
byteMultiplication(b0d, cipherBlock[2*COLUMNSIZE+i]) ^
byteMultiplication(b09, cipherBlock[3*COLUMNSIZE+i]);
                 sp[1] = byteMultiplication(b09, cipherBlock[i]) ^
byteMultiplication(b0e, cipherBlock[COLUMNSIZE+i]) ^
byteMultiplication(b0b, cipherBlock[2*COLUMNSIZE+i]) ^
byteMultiplication(b0d, cipherBlock[3*COLUMNSIZE+i]);
                 sp[2] = byteMultiplication(b0d, cipherBlock[i]) ^
byteMultiplication(b09, cipherBlock[COLUMNSIZE+i]) ^
byteMultiplication(b0e, cipherBlock[2*COLUMNSIZE+i]) ^
byteMultiplication(b0b, cipherBlock[3*COLUMNSIZE+i]);
```

```
            sp[3] = byteMultiplication(b0b, cipherBlock[i]) ^
byteMultiplication(b0d, cipherBlock[COLUMNSIZE+i]) ^
byteMultiplication(b09, cipherBlock[2*COLUMNSIZE+i]) ^
byteMultiplication(b0e, cipherBlock[3*COLUMNSIZE+i]);

            for (int j = 0; j < 4; j++) cipherBlock[j*COLUMNSIZE+i] =
(byte)(sp[j]);
         }
      }

      /**
       *  Enforces the shift-row operation. The official AES algorithm
shifts based on {0,1,2,3} shifting offsets.
       *  But the ProtectStar™ Extended AES Algorithm shifts based on
{0,1,4,5} shiftings as computed with {@link OptimumShiftRow} program.
       *  Normally the official AES-shifting provides full-diffusion after
2 rounds, but for the ProtectStar™ Extended AES Algorithm
       *  we need 4 rounds for full-diffusion due to the big block size.
Hence, two extra rounds are added
       *  to the total rounds number which is 22.
       *
       *  @param theBlock the bytes on which the shift-row operation is
applied.
       *  @param enc for encryption this parameter should be true,
otherwise for decryption it is false
       */
      private void shiftRows(byte[] theBlock,boolean enc) {

            byte[] temp_theBlock= new byte[theBlock.length];

            System.arraycopy(theBlock,0,temp_theBlock,0,theBlock.length);

            int[] shifting_offsets={0,1,4,5};


            for (int i=1;i < 4;i++) {

                  int sm=shifting_offsets[i];
                  int current_cell;

                  for (int j=0;j < COLUMNSIZE ; j++) {

                        current_cell=i*COLUMNSIZE+j;

                        if (enc){ // for encryption, left shifting

                              if ( current_cell-sm < i*COLUMNSIZE)
                                    theBlock[current_cell-
sm+COLUMNSIZE]=temp_theBlock[current_cell];
                              else
                                    theBlock[current_cell-
sm]=temp_theBlock[current_cell];
                        }
                        else { // for decryption, right shifting

                              if ( current_cell+sm >= (i+1)*COLUMNSIZE)
                                    theBlock[current_cell+sm-
COLUMNSIZE]=temp_theBlock[current_cell];
                              else

theBlock[current_cell+sm]=temp_theBlock[current_cell];
```

```
                        }
                  }
            }
      }


      /**
       * Enforces left shift-row operation which is also needed in the
roundkeys generation process.
       *
       *  @param bs the 4-bytes of the key as integer that will be shifted
to the left.
       *  @return bs the shifted 4-bytes as integer representation
       */
      private int shiftRows_RoundKeys(int bs) {
            byte[] shiftedArray;

            byte firstByte= (byte)(bs >> 24);

            int lastBytes= bs << 8;
            shiftedArray=integerToByteArray(lastBytes);
            shiftedArray[3]=firstByte;

            return byteArrayToInteger(shiftedArray);

      }


      /**
       * Enforces the official SBOX-lookup operation.
       *
       * @param theBlock the bytes that will be replaced with the new bytes
after the SBOX-lookups.
       *
       */
      private void SBOX_Substitution(byte[] theBlock) {

            for (int i=0;i<theBlock.length;i++) {
                  int firstPart=(theBlock[i]&0xff)>>4;
                  int secondPart=(theBlock[i] & 0x0f);

                  theBlock[i]=SBOX[COLUMNSIZE*firstPart+secondPart];
            }

      }


      /**
       * Enforces the official inverse SBOX-lookup operation for decryption
process.
       *
       * @param cipherBlock the bytes that will be replaced with the new
bytes after the inverse SBOX-lookups.
       *
       */
      private void Inv_SBOX_Substitution(byte[] cipherBlock) {

            for (int i=0;i<BLOCKSIZE;i++) {
                  int firstPart=(cipherBlock[i]&0xff)>>4;
                  int secondPart=(cipherBlock[i] & 0x0f);
```

```
cipherBlock[i]=SBOX_Inverse[COLUMNSIZE*firstPart+secondPart];
            }
      }


      /**
       * Adds (xor) the values of the round key to the input block values.
       *
       * @param theBlock the input block (for both encryption and
decryption)
       * @param roundKeys the generated round keys
       * @param start the start index of the round keys array for the
adding operation
       *
       */
      private void addRoundKey(byte[] theBlock, byte[] roundKeys, int
start) {

            for (int i=0;i<BLOCKSIZE;i++)
                  theBlock[i]=(byte) (theBlock[i] ^ roundKeys[i+start]);

      }


      /**
       * Converts a byte array to integer value.
       *
       * @param data the byte array
       * @return the integer value
       *
       */
      private int byteArrayToInteger(byte[] data) {

            int number = 0;

            for (int i = 0; i < 4; ++i) {
                  number |= (data[3-i] & 0xff) << (i << 3);
            }

            return number;
      }


      /**
       * Converts an integer value to a byte array.
       *
       * @param number the integer value
       * @return the byte array
       *
       */
      private byte[] integerToByteArray(int number) {

            byte[] data=new byte[4];

            for (int i = 0; i < 4; ++i) {
                  int shift = i << 3; // i * 8
                  data[3-i] = (byte)((number & (0xff << shift)) >>> shift);
            }
```

```
            return data;
      }

      /**
       * Converts an integer value to a byte array in the least-significant
byte order.
       *
       * @param number the integer value
       * @return the byte array
       *
       */

      protected byte[] invIntegerToByteArray(int number) {

            byte[] data=new byte[4];

            for (int i = 0; i < 4; ++i) {
                  int shift = i << 3; // i * 8
                  data[i] = (byte)((number & (0xff << shift)) >>> shift);
            }

            return data;
      }

      /**
       * Generates the round keys from a given password.
       *
       * The initial AES key is produced based on PKC#5 standard explained
in
       * ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2_1.pdf (Section
PBKDF2)
       *
       * @param password the user password
       *
       */
      protected void generateRoundKeys(String password) {

            int temp;
            byte[] temp_ba;

            final  String SALT=new String("aes-extended");
            final  int ITERATION=100;


            try {
                  originalKey =
PBKDF2(password.getBytes(),SALT.getBytes(),ITERATION,KEYSIZE);
            } catch (Exception e) {
                  Debug.debug(e.getMessage());
            }

            // copy original key into the round keys array
            System.arraycopy(originalKey,0,roundKeys,0,originalKey.length);


            int W[]=new int[BLOCKSIZE_WORD*(TOTALROUNDNUMBER +1)];

            for (int i=0;i<NK;i++) {

                  byte[] wi=new byte[4];
```

```
                wi[0]=originalKey[4*i];
                wi[1]=originalKey[4*i+1];
                wi[2]=originalKey[4*i+2];
                wi[3]=originalKey[4*i+3];

                W[i]=this.byteArrayToInteger(wi);

          }

          for (int i=NK; i<BLOCKSIZE_WORD*(TOTALROUNDNUMBER +1);i++) {

                temp= W[i-1];

                if ((i % NK == 0) || ((i%NK==4)&& (NK > 6))){
                      int sr=shiftRows_RoundKeys(temp);
                      temp_ba=integerToByteArray(sr);
                      SBOX_Substitution(temp_ba);

                      temp=byteArrayToInteger(temp_ba)^RCON[i/NK];
                }
                else if (((i % NK == 8) || (i % NK == 12))&& (NK > 6)) {
                      temp_ba=integerToByteArray(temp);
                      SBOX_Substitution(temp_ba);
                      temp=byteArrayToInteger(temp_ba);
                }

                W[i]=W[i-NK] ^ temp;
          }

          int index=originalKey.length;
          for (int i=NK;i<W.length;i++) {
                byte[] b=this.integerToByteArray(W[i]);

                roundKeys[index++]= b[0];
                roundKeys[index++]= b[1];
                roundKeys[index++]= b[2];
                roundKeys[index++]= b[3];

          }
    }

    /**
     *
     *  @param password the password array
     *  @param salt
     *  @param iteration count
     *  @param key size
     *  @return the derived key from password (based on PBKDF2 standard)
     *
     */

    private byte[] PBKDF2(byte[] P, byte[] S, int c, int dkLen)throws
Exception {

          int hLen=20; //output-size of SHA-1

          byte[] derivedKey=new byte[KEYSIZE];

          byte[] Ti=new byte[hLen];

          int l= dkLen / hLen +1;
```

```
            int r=dkLen - (l-1)*hLen;


            if (dkLen > (2^32 - 1)*hLen) throw new Exception("Derived Key
too long");
            else {

                  for (int i=0;i<l;i++) {

                        Ti = F(P,S,c,i);

                        if (i != l-1)
                              System.arraycopy(Ti,
0,derivedKey,i*hLen,Ti.length);
                  }

                  System.arraycopy(Ti,0,derivedKey,(l-1)*hLen,r);

            }

            return derivedKey;

      }

      private byte[] F(byte[] P, byte[] S, int c, int i) {

            int hLen=20; //output-size of SHA-1

            byte[] U1=new byte[hLen];
            byte[] Ui=new byte[hLen];
            byte[] result=new byte[hLen];


            byte[] buf=new byte[4];
            buf[0]=(byte)((i & 0xff000000)>>>24);
            buf[1]=(byte)((i & 0x00ff0000)>>>16);
            buf[2]=(byte)((i & 0x0000ff00)>>>8);
            buf[3]=(byte)((i & 0x000000ff));
            U1=PRF(P,buf);

            System.arraycopy(U1,0,Ui,0,U1.length);
            System.arraycopy(U1,0,result,0,U1.length);
            for (int j=0;j<c;j++){
                  Ui=PRF(P,Ui);
                  for (int k=0;k<result.length;k++)
                        result[k] =(byte) (result[k] ^ Ui[k]);

            }

            return result;
      }

      /*
       * Calculates SHA-1 of P+Ui concatanation
       */
      private byte[] PRF(byte[] P, byte[] Ui) {

            try {
                  MessageDigest md=MessageDigest.getInstance("SHA-1");

                  md.update(P);
```

```
                md.update(Ui);

                return md.digest();

            } catch (NoSuchAlgorithmException e) {
                Debug.debug(e.getMessage());

            }

            return null;
      }
}
```

```
package com.crypt;

import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

import com.crypt.debug.Debug;


/**
 * This class initializes the required parameters like the key size, the
mode of the operation,
 * the password, the bytes to be encrypted or decrypted. Adding and
removing input paddings are
 * also enforced within this class.
 *
 * @author
 * @version
 */


public class AESExtension implements BlockCipher{

      AESExtension_Algorithm ALGORITHM;
```

```
	private final String CIPHERNAME="Extended AES (Block Size:512-bit,
Key Sizes: 128/256/512 bits, Modes: ECB, CBC, OFB, CFB, CTR";

	// Modes of Operations
	public static enum Mode_of_Op {ECB, CBC, OFB, CFB,CTR};

	private final int BLOCKSIZE = 64; // Block size in bytes (= 512 Bits)
	private int keySize = 32; // default key size 32-byte (= 256 bit)
	private Mode_of_Op mop = Mode_of_Op.CTR; // default mode of operation
is Counter mode

	int counter=0; // used for CTR mode

	public AESExtension() {

		ALGORITHM=new AESExtension_Algorithm(keySize);
	}

	public AESExtension(int keySize){
		this.keySize=keySize;
		ALGORITHM=new AESExtension_Algorithm(keySize);
	}

	public AESExtension(int keySize, Mode_of_Op mop) {
		this.keySize = keySize;
		this.mop = mop;

		ALGORITHM=new AESExtension_Algorithm(keySize);
	}


	/**
	 * Initializes the Cipher by generating the rounds keys from the
given password
	 * @param password the secret password that helps to create pseudo-
random round keys
	 */
	public void initialize(String password) {
		// generate Round keys
		ALGORITHM.generateRoundKeys(password);

	}

	/**
	 * Returns the size of blocks (i.e. 64-bytes) in the AES Extended
Cipher
	 * @return the fixed block size
	 */
	public int getBlockSize() {
		return BLOCKSIZE;
	}

	/*
	 * Returns the specified key size
	 * @return the size of the cipher key
	 */
	public int getKeySize() {
		return keySize;
	}

	/**
```

```
	 * Returns the definition of the AES Extended cipher
	 *
	 * return the definition of the implemented cipher
	 *
	 */
	public String getAlgorithmName() {

		return CIPHERNAME;
	}

	/**
	 * Returns definition of the specified mode of the operation
	 * @return mode of the operation
	 */

	public String getMode() {

		switch (mop) {

		case ECB:
			 return new String("ECB: Electronic Code Book");
		case CBC:
			 return new String("CBC: Cipher Block Chaining");
		case OFB:
			 return new String("OFB: Output Feedback");
		case CFB:
			 return new String("CFB: Cipher Feedback");
		case CTR:
			 return new String("CTR: Counter");
		}

		return null;
	}

	/**
	 * Based on the mode of the operation, calls the block encryption
function from the
	 * real algorithm class for each block in the input.
	 *
	 * @param input the plaintext to encipher
	 * @return encrypted data
	 *
	 */
	public byte[] encrypt(byte[] input) {

		byte[] paddedInput = addInputPadding(input);

		byte[] encResult;

		int encBlockNumber = paddedInput.length / BLOCKSIZE;

		byte[] tempBlock= new byte[BLOCKSIZE]; // used for saving
encryption results temporarily

		byte[] IV=new byte[BLOCKSIZE];
		byte[] Ci=new byte[BLOCKSIZE];
		byte[] Vi=new byte[BLOCKSIZE]; // for OFB mode

		switch (mop) {

		case ECB:
```

```
                encResult = new byte[paddedInput.length];

                for (int i=0;i<encBlockNumber;i++) {
                      System.arraycopy(paddedInput,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);
                      ALGORITHM.encBlockProcess(tempBlock);
                      System.arraycopy(tempBlock,
0,encResult,i*BLOCKSIZE,BLOCKSIZE);
                }

                return encResult;


          case CBC:
                encResult = new byte[paddedInput.length + BLOCKSIZE];

                IV= generateRandomIV(BLOCKSIZE); // IV
                System.arraycopy(IV,0,encResult,0,BLOCKSIZE);

                Ci= new byte[BLOCKSIZE];
                System.arraycopy(IV,0,Ci,0,BLOCKSIZE); // C0

                for (int i=0;i<encBlockNumber;i++) {
                      System.arraycopy(paddedInput,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);

                      for (int j=0;j<tempBlock.length;j++)
                            tempBlock[j] = (byte)(tempBlock[j] ^ Ci[j]);

                      ALGORITHM.encBlockProcess(tempBlock);
                      System.arraycopy(tempBlock,0,Ci,0,BLOCKSIZE);
                      System.arraycopy(tempBlock,0,encResult,
(i+1)*BLOCKSIZE,BLOCKSIZE);
                }

                return encResult;


          case OFB:

                encResult = new byte[paddedInput.length + BLOCKSIZE];

                Vi=generateRandomIV(BLOCKSIZE); // V0

                System.arraycopy(Vi,0,encResult,0,BLOCKSIZE);

                for (int i=0;i<encBlockNumber;i++) {

                      System.arraycopy(paddedInput,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);

                      ALGORITHM.encBlockProcess(Vi);

                      for (int j=0;j<tempBlock.length;j++)
                            tempBlock[j] = (byte)(tempBlock[j] ^ Vi[j]);

                      System.arraycopy(tempBlock,0,encResult,
(i+1)*BLOCKSIZE,BLOCKSIZE);

                }
```

```
                return encResult;

            case CFB:

                encResult = new byte[paddedInput.length + BLOCKSIZE];

                Vi=generateRandomIV(BLOCKSIZE); // V0

                System.arraycopy(Vi,0,encResult,0,BLOCKSIZE);

                for (int i=0;i<encBlockNumber;i++) {

                    System.arraycopy(paddedInput,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);

                    ALGORITHM.encBlockProcess(Vi);

                    for (int j=0;j<tempBlock.length;j++)
                        tempBlock[j] = (byte)(tempBlock[j] ^ Vi[j]);

                    System.arraycopy(tempBlock,0,encResult,
(i+1)*BLOCKSIZE,BLOCKSIZE);

                    System.arraycopy(tempBlock,0,Vi,0,BLOCKSIZE);

                }

                return encResult;


            case CTR:
                encResult = new byte[paddedInput.length+BLOCKSIZE];

                IV=generateRandomIV(BLOCKSIZE);

                Ci=new byte[BLOCKSIZE];


                System.arraycopy(IV,0,encResult,0,BLOCKSIZE);

                for (int i=0;i<encBlockNumber;i++) {

                    System.arraycopy(paddedInput,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE); // Mi

                    System.arraycopy(IV,0,Ci,0,BLOCKSIZE);
                    addCounter(Ci);

                    ALGORITHM.encBlockProcess(Ci);

                    for (int j=0;j<tempBlock.length;j++)
                        Ci[j]=(byte)(Ci[j] ^ tempBlock[j]);

                    System.arraycopy(Ci,0,encResult,
(i+1)*BLOCKSIZE,BLOCKSIZE);
                }

                return encResult;

            }
```

```
            return null;

      }

      /**
       * Based on the mode of the operation, calls the block decryption
function from the
       * real algorithm class for each block in the ciphertext.
       *
       * @param ciphertext the encrypted data to decipher
       * @return deciphered data
       *
       */

      public byte[] decrypt(byte[] ciphertext) {

            byte[] decResult=null;

            int decBlockNumber = ciphertext.length / BLOCKSIZE;

            byte[] tempBlock= new byte[BLOCKSIZE]; // used for saving
decryption results temporarily

            switch(mop) {

            case ECB:

                  decResult=new byte[ciphertext.length];

                  for (int i=0;i<decBlockNumber;i++) {

                        System.arraycopy(ciphertext,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);

                        ALGORITHM.decBlockProcess(tempBlock);

                        System.arraycopy(tempBlock,
0,decResult,i*BLOCKSIZE,BLOCKSIZE);
                  }

                  break;

            case CBC:

                  decResult=new byte[ciphertext.length-BLOCKSIZE];

                  byte[] Ci_prev=new byte[BLOCKSIZE];
                  byte[] Ci_prev_temp=new byte[BLOCKSIZE];

                  byte[] Ci=new byte[BLOCKSIZE];
                  byte[] Vi=new byte[BLOCKSIZE];


                  System.arraycopy(ciphertext,0,Ci_prev,0,BLOCKSIZE); // C0

                  for (int i=1;i<decBlockNumber;i++) {

                        System.arraycopy(ciphertext,i*BLOCKSIZE,Ci,
0,BLOCKSIZE);

                        System.arraycopy(Ci,0,Ci_prev_temp,0,BLOCKSIZE);
```

```
                        ALGORITHM.decBlockProcess(Ci);

                        for (int j=0;j<tempBlock.length;j++)
                              tempBlock[j] = (byte)(Ci[j] ^ Ci_prev[j]);


                        System.arraycopy(tempBlock,0,decResult,
(i-1)*BLOCKSIZE,BLOCKSIZE);

                        System.arraycopy(Ci_prev_temp,0,Ci_prev,
0,BLOCKSIZE);

                  }

                  break;

            case OFB:

                  decResult=new byte[ciphertext.length-BLOCKSIZE];

                  Vi=new byte[BLOCKSIZE];

                  System.arraycopy(ciphertext,0,Vi,0,BLOCKSIZE); // V0

                  for (int i=1;i<decBlockNumber;i++) {

                        ALGORITHM.encBlockProcess(Vi);

                        System.arraycopy(ciphertext,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE); //Ci

                        for (int j=0;j<tempBlock.length;j++)
                              tempBlock[j] = (byte)(tempBlock[j] ^ Vi[j]);

                        System.arraycopy(tempBlock,0,decResult,
(i-1)*BLOCKSIZE,BLOCKSIZE);
                  }

                  break;

            case CFB:

                  decResult=new byte[ciphertext.length-BLOCKSIZE];

                  Vi=new byte[BLOCKSIZE];
                  byte[] temp_Vi=new byte[BLOCKSIZE];

                  System.arraycopy(ciphertext,0,Vi,0,BLOCKSIZE); // V0

                  for (int i=1;i<decBlockNumber;i++) {

                        ALGORITHM.encBlockProcess(Vi);

                        System.arraycopy(ciphertext,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE); //Ci

                        System.arraycopy(tempBlock,0,temp_Vi,0,BLOCKSIZE);

                        for (int j=0;j<tempBlock.length;j++)
                              tempBlock[j] = (byte)(tempBlock[j] ^ Vi[j]);
```

```
                        System.arraycopy(tempBlock,0,decResult,
(i-1)*BLOCKSIZE,BLOCKSIZE);

                        System.arraycopy(temp_Vi,0,Vi,0,BLOCKSIZE);
                  }

                  break;

            case CTR:

                  counter=0;
                  decResult=new byte[ciphertext.length-BLOCKSIZE];

                  Ci=new byte[BLOCKSIZE]; // Ci=IV



                  for (int i=1;i<decBlockNumber;i++) {

                        System.arraycopy(ciphertext,0,Ci,0,BLOCKSIZE);
                        addCounter(Ci);
                        ALGORITHM.encBlockProcess(Ci);

                        System.arraycopy(ciphertext,i*BLOCKSIZE,tempBlock,
0,BLOCKSIZE);

                        for (int j=0;j<tempBlock.length;j++)
                              tempBlock[j] = (byte)(tempBlock[j] ^ Ci[j]);

                        System.arraycopy(tempBlock,0,decResult,
(i-1)*BLOCKSIZE,BLOCKSIZE);

                  }

                  break;

            }

            byte[] plainText=removeInputPadding(decResult);

            return plainText;

      }


      /**
       * Adds the value of a counter as byte array to the randomly chosen
input byte-array Ci that is needed
       * in counter mode.
       *
       * @param Ci the randomly chosen input for encryption operation in
counter mode
       */

      private void addCounter(byte[] Ci) {

            byte[] counter_array=ALGORITHM.invIntegerToByteArray(counter);

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

```
                  Ci[BLOCKSIZE-i-1]= (byte)(Ci[BLOCKSIZE-i-1] ^
counter_array[i]);

            counter++;

      }

      /**
       * Generates random initialization vector bytes for using in
different modes of operations.
       *
       * @param blockSize the number of bytes for IV to be generated
       * @return the IV having the size specified by blockSize
       *
       */
      private byte[] generateRandomIV(int blockSize) {


            SecureRandom random=null;

            try {
                  random=SecureRandom.getInstance("SHA1PRNG");
            } catch (NoSuchAlgorithmException e) {
                  Debug.debug(e.getMessage());
            }

            byte[] IV=new byte[blockSize];

            random.nextBytes(IV);

            return IV;

      }

      /**
       * Adds padding into a given input according to RFC 1321. The length
of the input should be multiple of 64 (i.e. the fixed
       * block size). As a padding method, after the original input bits
one "1" is inserted and the remaining bits
       * are assigned as "0". If there is no space left for any extra
padding zeros, then a new 64-byte block is
       * added at the end of the input.
       *
       *  @param input the input to be padded
       *  @return the padded input
       */
      private byte[] addInputPadding(byte[] input) {

            byte[] paddedInput;

            int size= input.length / BLOCKSIZE +1;

            if (((input.length % BLOCKSIZE)== BLOCKSIZE-1) ||
((input.length % BLOCKSIZE)== 0)){
                  paddedInput= new byte[(size+1)*BLOCKSIZE];
            }
            else {
                  paddedInput= new byte[(size)*BLOCKSIZE];
            }
```

```
            System.arraycopy(input,0,paddedInput,0,input.length);
            paddedInput[input.length]=(byte)0x80;

            for (int i=input.length+1;i<paddedInput.length;i++)
                  paddedInput[i]=(byte)0x00;

            return paddedInput;

      }

      /**
       * Removes padded bytes from the deciphered bytes.
       *
       *  @param decResult the decrypted bytes whose padded bytes are
removed
       *  @return the padding-free decrypted bytes
       */
      private byte[] removeInputPadding(byte[] decResult) {

            int realIndex=decResult.length-1;

            while (decResult[realIndex]!= (byte)0x80) realIndex--;

            byte[] plainText=new byte[realIndex];

            System.arraycopy(decResult,0,plainText,0,realIndex);

            return plainText;
      }

}
```

```
package com.crypt;

/**
 * This interface specifies the certain methods which a {@link BlockCipher}
should implement.
 *
 *  @ProtectStar, Inc.
 *  @ver 2.0
 *
 *
 */
public interface BlockCipher {

      /**
       * Initializes the round keys from the given passwords
       *
       * @param password the user password for encrypting and decrypting
the data
       *
       */
      public void initialize(String password);

      /**
       * Returns the size of the block in bytes.
       *
       * @return the size of the block in bytes
       */
      public int getBlockSize();

      /**
       * Returns the size of the key in bytes
       *
       * @return the size of the key in bytes
       */
      public int getKeySize();

      /**
       * Returns the mode of the operation used by the {@link BlockCipher}
       *
       * @return the mode of operation
       */
      public String getMode();

      /**
       * Executes the encryption process for a given byte-array input
```

```
         *
         * @param input the bytes to be encrypted
         * @return encrypted bytes
         *
         */
        public byte[] encrypt(byte[] input);

        /**
         * Executes the decryption process for a given encrypted byte-array
input
         *
         * @param input the bytes to be decrypted
         * @return decrypted bytes
         *
         */
        public byte[] decrypt(byte[] input);

        /**
         * Returns the name of the algorithm specified by the {@link
BlockCipher}
         *
         * @return the name of the BlockCipher algorithm
         */
        public String getAlgorithmName();


}
```

Contact

ProtectStar Inc.

444 Brickell Avenue

Suite 51103

Miami, FL, 33131

USA

info@protectstar.com
https://www.protectstar.com