Serpent Cipher

Final Report

Cryptography

Team Rikki-Tikki-Tavi

Authors:
Nicholas Sereni
Dan Grau
Karl Berger

Prof.:
Alan Kaminsky

February 8, 2013

Contents

1 Serpent Cipher

1.1 Background

The Serpent cipher was designed by Ross Angerson, Eli Biham, and Lars Knudsen. It was created as candidate for the Advanced Encryption Standard. Based on AES requirements, it has a 128 bit block length and a 256 bit key length. It also supports keys sizes of 128 and 192 bits.

1.2 The Algorithm

Serpent splits the 128 bit block into four 32-bit words. There are 32 rounds. Each round uses a subkey generated from the user key. The user key does not have a size requirement, but it becomes fixed at 128, 192, or 256 bits. Padding is achieved by appending a "1" followed by "0" bits. The algorithm can be summarized as:

- An initial permutation
- 32 rounds consisting of:
  - key mixing operation
  - S-boxes
  - linear transformation (replaced by a a key mixing operation in the final round)
- A final permutation

  This process is explained visually in Figure 1.

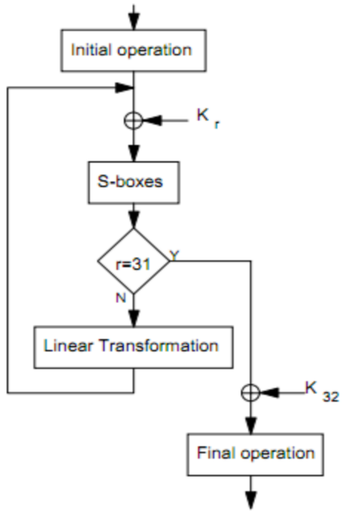

Figure 1: Block Diagram of the Encryption Process

1.2.1 Initial and Final Permutations

The initial and final permutations are simply bit mappings. This is a very simple method and is especially effective in hardware. In permutations, each bit on the input is assigned to a different index on the output. There are no operations performed, only reassignments. Figure 2 shows this general idea. Please note that this diagram does not represent the diagram for Serpent(it's acutally for DES) and is only being used for an example. The actual permutations can be found in [Anderson, R., Biham, E., & Knudsen, L. 1998].

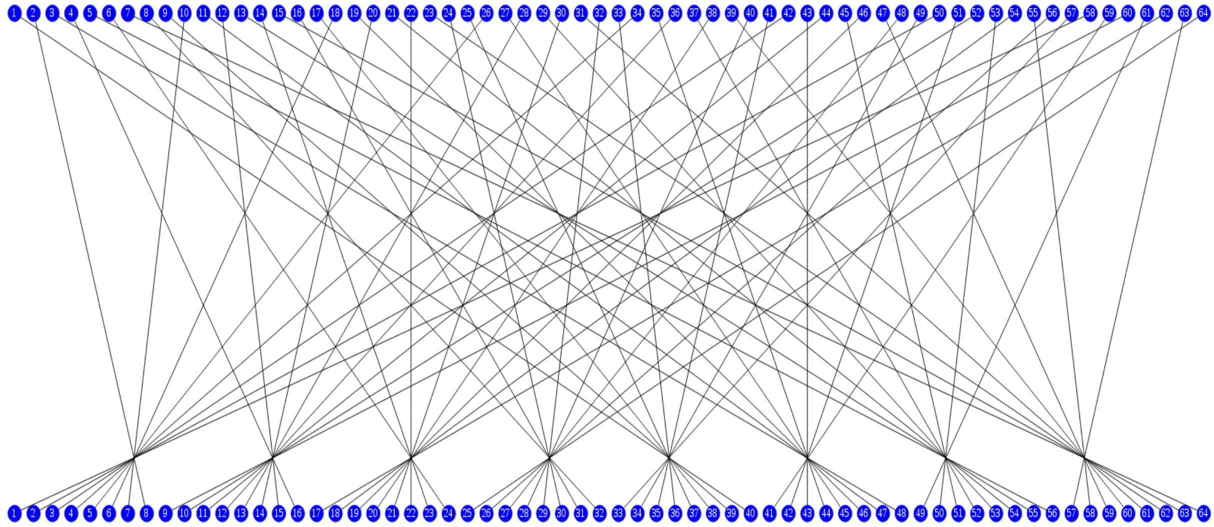

Figure 2: A General Permutation

1.2.2 S-boxes

An S-box is simply a look-up-table. In Serpent, the S-boxes are 4-bit permutations. The advantage of an S-box is that for a 1-bit change of an input value, the output is guaranteed to be altered by more than one bit (at least for the Serpent S-boxes). An exmaple S-box can be seen in Figure 3. Please note that this diagram does not represent the S-box for Serpent. The Serpent S-boxes can be seen in [Anderson, R., Biham, E., & Knudsen, L. 1998].

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3: A General S-box

1.2.3 Linear Transformation

The linear transformation functions acts on the 128-bit block as four 32-bit words. Each word is linearly adjusted and combined with other words according to Figure 4. In this figure, $<<<$ denotes a left rotation, and $<<$ denotes a left shift.

$$\begin{aligned}
X_0, X_1, X_2, X_3 &:= \mathcal{S}_i(B_i \oplus K_i) \\
X_0 &:= X_0 <<< 13 \\
X_2 &:= X_2 <<< 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &:= X_1 <<< 1 \\
X_3 &:= X_3 <<< 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &:= X_0 <<< 5 \\
X_2 &:= X_2 <<< 22 \\
B_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}$$

Figure 4: Linear Transformation

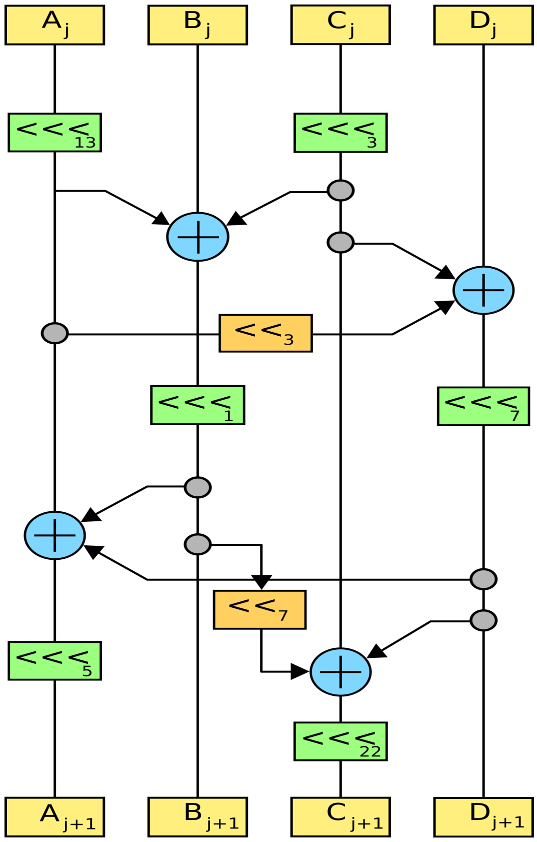

Figure 5: Linear Transformation

1.2.4 Decryption

Decrption is very similar to encryption. However, inverse S-boxes and linear transformations are used as well as a reverse order of subkeys. This is made most clear with the use of Figure 6.

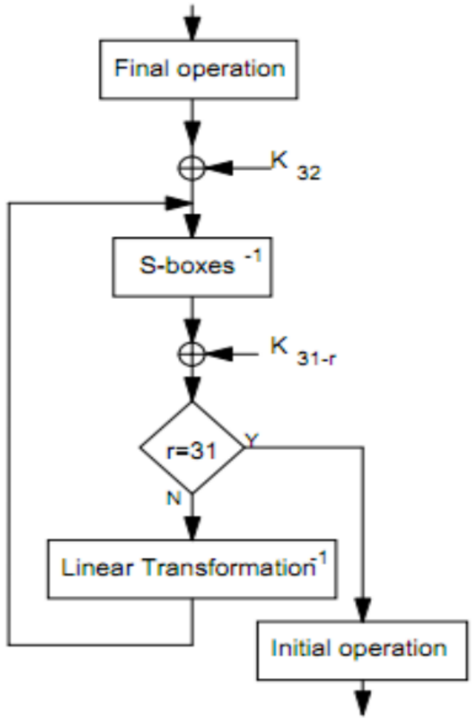

Figure 6: Block Diagram of the Decryption Process

2 Original Implementation

2.1 Timing Results

All timing results were measured on the CS machine, Joplin. Source-code level results were truncated as problem methods are easily identifiable within the first few lines.

2.1.1 Total Running Time with no JIT compiler

$ time java -Xint Serpent 1
49672ba898d98df95019180445491089
real 0m0.135s

user 0m0.040s
sys 0m0.024s

2.1.2 Runtime Profiles of over 100 Seconds

$java -Xint -Xprof Serpent 100000

d3f68d0623563be822d68dde8f4ad282

Flat profile of 156.42 secs (15402 total ticks): main

Interpreted		+	native	Method
26.7	% 4119	+	0	Serpent.sBox
22.8	% 3511	+	0	Serpent.getRoundKey
21.4	% 3290	+	0	Serpent.initPermutation
21.3	% 3274	+	0	Serpent.linearTransform
3.5	% 540	+	0	Serpent.setKey
3.5	% 537	+	0	Serpent.finalPermutation
0.8	% 128	+	0	Serpent.encrypt
0.0	% 0	+	2	java.io.FileInputStream.readBytes
0.0	% 0	+	1	java.io.UnixFileSystem.getBooleanAttributes0
100.0	% 15399	+	3	Total interpreted

$time java -Xint -agentlib:hprof=cpu=samples,depth=10 Serpent 100000

d3f68d0623563be822d68dde8f4ad282

Dumping CPU usage by sampling running threads ... done.

real 2m38.259s

user 2m38.062s

sys 0m0.488s

rank	self	accum	count	trace	method
1	7.06	% 7.06	% 1092	300041	Serpent.initPermutation
2	6.91	% 13.97	% 1070	300033	Serpent.linearTransform
3	6.57	% 20.54	% 1016	300042	Serpent.getRoundKey
4	6.30	% 26.84	% 975	300027	Serpent.sBox
5	6.19	% 33.03	% 958	300074	Serpent.initPermutation
6	4.25	% 37.27	% 657	300032	Serpent.sBox
7	2.43	% 39.70	% 376	300040	Serpent.initPermutation
8	2.35	% 42.05	% 364	300045	Serpent.initPermutation
9	2.31	% 44.36	% 357	300086	Serpent.finalPermutation
10	2.29	% 46.66	% 355	300034	Serpent.finalPermutation
11	2.27	% 48.93	% 352	300051	Serpent.initPermutation

2.2 Analysis

These timing results both show the major functions taking the majority of the CPU as expected. Namely, initPermutations. Logically, initial permutations should take very little time as it is only called once for each encryption. However, an implementation was made which uses the inital permutation of the linear transform of the final permutation (IP(LT(FP(x)))) as noted in [Anderson, R., Biham, E., & Knudsen, L. 1998].

2.3 Example Runs

```
java Serpent 1
49672ba898d98df95019180445491089

java Serpent 100
5a445efd4923ebddea1d5be4511bd4d6

java Serpent 1000
d72ec2b7b93fbb567cefbab3fab43fb4
```

Each of these runs use a key of all “0’s” and a plaintext of all “0’s.” Please note that these values match the values specified at: http://www.cs.technion.ac.il/ biham/Reports/Serpent/Se
256-128.verified.test-vectors

3 Optimized Code

3.1 Timing Results

3.1.1 Total Running Time with no JIT compiler

3.1.2 Runtime Profiles of over 100 Seconds

3.2 Analysis

3.3 Example Runs

4 Division of Labor

- Nicholas Sereni

- Linear Transform
  - Final Report
  - Runtime Results
  - General Debugging

- Dan Grau
  - Initial and Final Permutations
  - File Reading
  - Optimizations
  - General Debugging

- Karl Berger
  - Key Scheduler
  - S-Boxes
  - Decryption
  - General Debugging

5 Manuals

5.1 Developer's Guide

If the source code has been removed from the given archive, then all that is required is BlockCipher.java and Serpent.java (or SerpentOptimized.java to run the optimized code). Execute the following command to compile the program:

```
javac -classpath <path to Parallel Java library> BlockCipher.java Serpent.java
```

5.2 User's Guide

Once the program has been compiled, there are two ways in which it can be run. The first is to encrypt a block of 0s N number of times with a key of all 0s.

```
java Serpent <N>
```

The second way to run the program is the encrypt or decrypt a file. There are 5 arguments to the program in this case and are as follows.

```
java Serpent <input filename> <output filename> <key in hex> \
  <integer Nonce> <'e' to encrypt or 'd' to decrypt>
```

For example:

```
java Serpent cat.jpg cat.encrypt 112233445566778899aabbccddeeff 12345 e
```

6 Source Code

6.1 Original Code

```java
import edu.rit.util.Hex;
import edu.rit.util.Packing;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.Arrays;
import java.lang.Integer;
import java.io.*;

public class Serpent implements BlockCipher {

    private static final byte xFF = (byte)0xFF;
    private int keySize;
    private byte[] key;
    private int[] prekeys;

    public Serpent() {
        prekeys = new int[140];
    }
```

```
/**
 * Returns this block cipher's block size in bytes.
 *
 * @return  Block size.
 */
public int blockSize() {
    return 16;
}

/**
 * Returns this block cipher's key size in bytes.
 *
 * @return  Key size.
 */
public int keySize() {
    return 32;
}

/**
 * Set the key for this block cipher. If <TT>key</TT> is an
     array of bytes
 * whose length is less than <TT>keySize()</TT>, it will be
     padded to
 * <TT>keySize()</TT>
 *
 * @param  key  Key.
 */
public void setKey(byte[] key) {
    if (key.length != keySize()) {
        this.key = new byte[keySize()];
        for( int i = 0; i < key.length; i++ ) {
            this.key[i] = key[i];
        }
                          //Pad key to 256-bit
        for( int i = key.length; i < keySize(); i++ ) {
```

```
            if( i == key.length ) {
                //Start of padding!
                this.key[i] = (byte)0x80;
            }else {
                this.key[i] = (byte)0x00;
            }
        }
    }else {
        this.key = key;
    }

    //prekey initialization from K
    for(int i = 0; i < 8; i++) {
        prekeys[i] = Packing.packIntBigEndian(new byte[]{
           this.key[4*i],this.key[4*i+1],this.key[4*i+2],
           this.key[4*i+3]}, 0);
    }
    //Build out prekey array
                //There's a shift of 8 positions here
                    because I build the intermediate keys in
                     the same
                //array as the other prekeys.
    for( int i = 8; i < prekeys.length; i++ ) {
        byte[] prnt = new byte[4];
                            //Phi is the fractional part of
                                the golden ratio
        int phi = 0x9e3779b9;
        int tmp;
        tmp = prekeys[i-8] ^ prekeys[i-5] ^ prekeys[i-3] ^
            prekeys[i-1] ^
             i-8 ^ phi;
        prekeys[i] = (tmp << 11) | (tmp >>> (21));
        prnt = new byte[4];
        Packing.unpackIntBigEndian(prekeys[i], prnt, 0);
     }
}
```

```
/**
 * Encrypt the given plaintext. <TT>text</TT> must be an
    array of bytes
 * whose length is equal to <TT>blockSize()</TT>. On input,
    <TT>text</TT>
 * contains the plaintext block. The plaintext block is
    encrypted using the
 * key specified in the most recent call to <TT>setKey()</
    TT>. On output,
 * <TT>text</TT> contains the ciphertext block.
 *
 * @param  text  Plaintext (on input), ciphertext (on
    output).
 */
public void encrypt(byte[] text) {
    byte[] data = initPermutation(text);
    byte[] temp = new byte[] {
            data[12], data[13], data[14], data[15],
            data[8], data[9], data[10], data[11],
            data[4], data[5], data[6], data[7],
            data[0], data[1], data[2], data[3],
            };
    data = temp;
    byte[] roundKey = new byte[16];
    //32 rounds
    for(int i = 0; i < 32; i++){
        roundKey = getRoundKey(i);
        for(int n = 0; n < 16; n++){
            data[n] = (byte) (data[n] ^ roundKey[n]);
        }
        data = sBox(data, i);

        if(i == 31){
```

```
                                        //For round 32, instead
                                           of a linear
                                          transform
                                        // we get the last
                                          produced round key
                                          and xor
                                        // it with the current
                                          state.
            roundKey = getRoundKey(32);
            for(int n = 0; n < 16; n++){
                data[n] = (byte) (data[n] ^ roundKey[n]);
            }
        }
        else{
            data = linearTransform(data);
        }
    }
    data = finalPermutation(data);
    text[0] = data[3];
    text[1] = data[2];
    text[2] = data[1];
    text[3] = data[0];
    text[4] = data[7];
    text[5] = data[6];
    text[6] = data[5];
    text[7] = data[4];
    text[8] = data[11];
    text[9] = data[10];
    text[10] = data[9];
    text[11] = data[8];
    text[12] = data[15];
    text[13] = data[14];
    text[14] = data[13];
    text[15] = data[12];
}
```

```
/**
 * Decrypt the given ciphertext. We decrypt by performing
    the inverse
	 * operations performed to encrypt in reverse order.
 *
 * @param text ciphertext (on input), original plaintext
    (on output).
 */
public void decrypt(byte[] text) {
    byte[] temp = new byte[] {
            text[3], text[2], text[1], text[0],
            text[7], text[6], text[5], text[4],
            text[11], text[10], text[9], text[8],
            text[15], text[14], text[13], text[12],
        };
    byte[] data = initPermutation(temp);
    byte[] roundKey = getRoundKey(32);
    for(int n = 0; n < 16; n++){
        data[n] = (byte) (data[n] ^ roundKey[n]);
    }
    //32 rounds in reverse
    for(int i = 31; i >= 0; i--){
        if(i!=31){
            data = invLinearTransform(data);
        }
        data = sBoxInv(data, i);
        roundKey = getRoundKey(i);
        for(int n = 0; n < 16; n++){
            data[n] = (byte) (data[n] ^ roundKey[n]);
        }
    }
    data = finalPermutation(data);
    text[0] = data[3];
    text[1] = data[2];
    text[2] = data[1];
    text[3] = data[0];
```

```
        text[4] = data[7];
        text[5] = data[6];
        text[6] = data[5];
        text[7] = data[4];
        text[8] = data[11];
        text[9] = data[10];
        text[10] = data[9];
        text[11] = data[8];
        text[12] = data[15];
        text[13] = data[14];
        text[14] = data[13];
        text[15] = data[12];
    }

    private byte[] initPermutation(byte[] data) {
        byte[] output = new byte[16];
        for (int i = 0;  i < 128; i++) {
                                //Bit permutation based on ip
                                    lookup table
            int bit = (data[(ipTable[i]) / 8] >>> ((ipTable[i])
                % 8)) & 0x01;
            if ((bit & 0x01) == 1)
                output[15- (i/8)] |= 1 << (i % 8);
            else
                output[15 - (i/8)] &= ~(1 << (i % 8));
        }
        return output;
    }

    private byte[] finalPermutation(byte[] data) {
        byte[] output = new byte[16];
        for (int i = 0;  i < 128; i++) {
                                //Bit permutation based on fp
                                    lookup table
            int bit = (data[15-fpTable[i] / 8] >>> (fpTable[i]
               % 8)) & 0x01;
```

```
        if ((bit & 0x01) == 1)
            output[(i/8)] |= 1 << (i % 8);
        else
            output[(i/8)] &= ~(1 << (i % 8));
    }
    return output;
}

private static byte[] s0 = new byte[]
    {3,8,15,1,10,6,5,11,14,13,4,2,7,0,9,12};
private static byte[] s1 = new byte[]
    {15,12,2,7,9,0,5,10,1,11,14,8,6,13,3,4};
private static byte[] s2 = new byte[]
    {8,6,7,9,3,12,10,15,13,1,14,4,0,11,5,2};
private static byte[] s3 = new byte[]
    {0,15,11,8,12,9,6,3,13,1,2,4,10,7,5,14};
private static byte[] s4 = new byte[]
    {1,15,8,3,12,0,11,6,2,5,4,10,9,14,7,13};
private static byte[] s5 = new byte[]
    {15,5,2,11,4,10,9,12,0,3,14,8,13,6,7,1};
private static byte[] s6 = new byte[]
    {7,2,12,5,8,4,6,11,14,9,1,15,13,3,10,0};
private static byte[] s7 = new byte[]
    {1,13,15,0,14,8,2,11,7,4,12,10,9,3,5,6};
private static byte[][] sBoxes = new byte[][]
    {s0,s1,s2,s3,s4,s5,s6,s7};

/**
 * Perform S-Box manipulation to the given byte array of <
    TT>blocksize()</TT> length.
 *
 * @param data Input bit sequence
 * @param round Number of the current round, used to
    determine which S-Box to use.
 */
private byte[] sBox(byte[] data, int round) {
```

```
        byte[] toUse = sBoxes[round%8];
        byte[] output = new byte[blockSize()];
        for( int i = 0; i < blockSize(); i++ ) {
            //Break signed-ness
            int curr = data[i]&0xFF;
            byte low4 = (byte)(curr>>>4);
            byte high4 = (byte)(curr&0x0F);
            output[i] = (byte) ((toUse[low4]<<4) ^ (toUse[high4
                ]));
        }
        return output;
    }

    private static byte[] is0 = new byte[]
        {13,3,11,0,10,6,5,12,1,14,4,7,15,9,8,2};
    private static byte[] is1 = new byte[]
        {5,8,2,14,15,6,12,3,11,4,7,9,1,13,10,0};
    private static byte[] is2 = new byte[]
        {12,9,15,4,11,14,1,2,0,3,6,13,5,8,10,7};
    private static byte[] is3 = new byte[]
        {0,9,10,7,11,14,6,13,3,5,12,2,4,8,15,1};
    private static byte[] is4 = new byte[]
        {5,0,8,3,10,9,7,14,2,12,11,6,4,15,13,1};
    private static byte[] is5 = new byte[]
        {8,15,2,9,4,1,13,14,11,6,5,3,7,12,10,0};
    private static byte[] is6 = new byte[]
        {15,10,1,13,5,3,6,0,4,9,14,7,2,12,8,11};
    private static byte[] is7 = new byte[]
        {3,0,6,13,9,14,15,8,5,12,11,7,10,1,4,2};
    private static byte[][] isBoxes = new byte[][]
        {is0, is1, is2, is3, is4, is5, is6, is7};

    /**
     * Perform inverse S-Box manipulation to the given byte
         array of <TT>blocksize()</TT> length.
     *
```

```
 * @param data Input bit sequence
 * @param round Number of the current round, used to
     determine which inverted S-Box to use.
 */
private byte[] sBoxInv(byte[] data, int round) {
    byte[] toUse = isBoxes[round%8];
    byte[] output = new byte[blockSize()];
    for( int i = 0; i < blockSize(); i++ ) {
        //Break signed-ness
        int curr = data[i]&0xFF;
        byte low4 = (byte)(curr>>>4);
        byte high4 = (byte)(curr&0x0F);
        output[i] = (byte) ((toUse[low4]<<4) ^ (toUse[high4
           ]));
    }
    return output;
}

private static byte[] ipTable = new byte[] {
     0, 32, 64,  96,  1, 33, 65,  97,  2, 34, 66,  98,  3,
        35, 67,  99,
     4, 36, 68, 100,  5, 37, 69, 101,  6, 38, 70, 102,  7,
        39, 71, 103,
     8, 40, 72, 104,  9, 41, 73, 105, 10, 42, 74, 106, 11,
        43, 75, 107,
    12, 44, 76, 108, 13, 45, 77, 109, 14, 46, 78, 110, 15,
       47, 79, 111,
    16, 48, 80, 112, 17, 49, 81, 113, 18, 50, 82, 114, 19,
       51, 83, 115,
    20, 52, 84, 116, 21, 53, 85, 117, 22, 54, 86, 118, 23,
       55, 87, 119,
    24, 56, 88, 120, 25, 57, 89, 121, 26, 58, 90, 122, 27,
       59, 91, 123,
    28, 60, 92, 124, 29, 61, 93, 125, 30, 62, 94, 126, 31,
       63, 95, 127
};
```

```
private static byte[] fpTable = new byte[] {
     0,  4,  8, 12, 16, 20, 24, 28, 32,  36,  40,  44,  48,
         52,  56,  60,
    64, 68, 72, 76, 80, 84, 88, 92, 96, 100, 104, 108, 112,
        116, 120, 124,
     1,  5,  9, 13, 17, 21, 25, 29, 33,  37,  41,  45,  49,
         53,  57,  61,
    65, 69, 73, 77, 81, 85, 89, 93, 97, 101, 105, 109, 113,
        117, 121, 125,
     2,  6, 10, 14, 18, 22, 26, 30, 34,  38,  42,  46,  50,
         54,  58,  62,
    66, 70, 74, 78, 82, 86, 90, 94, 98, 102, 106, 110, 114,
        118, 122, 126,
     3,  7, 11, 15, 19, 23, 27, 31, 35,  39,  43,  47,  51,
         55,  59,  63,
    67, 71, 75, 79, 83, 87, 91, 95, 99, 103, 107, 111, 115,
        119, 123, 127
};

/**
 * Performs linear transformation on the input bit sequence
 *
 * @param data Input bit sequence
 * @return output bit sequence
 */
private byte[] linearTransform(byte[] data){
    data = finalPermutation(data);
    byte[] output = new byte[blockSize()];
    ByteBuffer buffer = ByteBuffer.wrap(data);
    int x0 =  buffer.getInt();
    int x1 =  buffer.getInt();
    int x2 =  buffer.getInt();
    int x3 =  buffer.getInt();
    x0 = ((x0 << 13) | (x0 >>> (32 - 13)));
    x2 = ((x2 << 3) | (x2 >>> (32 - 3)));
```

```
    x1 = x1 ^ x0 ^ x2;
    x3 = x3 ^ x2 ^ (x0 << 3);
    x1 = (x1 << 1) | (x1 >>> (32 - 1));
    x3 = (x3 << 7) | (x3 >>> (32 - 7));
    x0 = x0 ^ x1 ^ x3;
    x2 = x2 ^ x3 ^ (x1 << 7);
    x0 = (x0 << 5) | (x0 >>> (32-5));
    x2 = (x2 << 22) | (x2 >>> (32-22));
    buffer.clear();
    buffer.putInt(x0);
    buffer.putInt(x1);
    buffer.putInt(x2);
    buffer.putInt(x3);

    output = buffer.array();
    output = initPermutation(output);

    return output;
}

/**
 * Performs inverse linear transformation on the input bit
    sequence.
       * This is the linear transform in reverse with
          inverted operations.
 *
 * @param data Input bit sequence
 * @return output bit sequence
 */
private byte[] invLinearTransform(byte[] data){
    data = finalPermutation(data);
    byte[] output = new byte[blockSize()];
    ByteBuffer buffer = ByteBuffer.wrap(data);
    int x0 =  buffer.getInt();
    int x1 =  buffer.getInt();
    int x2 =  buffer.getInt();
```

```
    int x3 =  buffer.getInt();

    x2 = (x2 >>> 22) | (x2 << (32-22));
    x0 = (x0 >>> 5) | (x0 << (32-5));
    x2 = x2 ^ x3 ^ (x1 << 7);
    x0 = x0 ^ x1 ^ x3;
    x3 = (x3 >>> 7) | (x3 << (32-7));
    x1 = (x1 >>> 1) | (x1 << (32-1));
    x3 = x3 ^ x2 ^ (x0 << 3);
    x1 = x1 ^ x0 ^ x2;
    x2 = (x2 >>> 3) | (x2 << (32-3));
    x0 = (x0 >>> 13) | (x0 << (32-13));

    buffer.clear();
    buffer.putInt(x0);
    buffer.putInt(x1);
    buffer.putInt(x2);
    buffer.putInt(x3);

    output = buffer.array();
    output = initPermutation(output);

    return output;
}

    /**
     * Fetches round key.  Round keys are built on request
        from the
     * prekeys that were created when the key was set.
     *
     * @param round Number of the round for which a key is
        needed.
     * @return byte[] The round key for the requested round
        .
     */
private byte[] getRoundKey(int round) {
```

```
        int k0 = prekeys[4*round+8];
        int k1 = prekeys[4*round+9];
        int k2 = prekeys[4*round+10];
        int k3 = prekeys[4*round+11];
        int box = (((3-round)%8)+8)%8;
        byte[] in = new byte[16];
        for (int j = 0; j < 32; j+=2) {
            in[j/2] = (byte) (((k0 >>> j) & 0x01)      |
            ((k1 >>> j) & 0x01) << 1 |
            ((k2 >>> j) & 0x01) << 2 |
            ((k3 >>> j) & 0x01) << 3 |
            ((k0 >>> j+1) & 0x01) << 4 |
            ((k1 >>> j+1) & 0x01) << 5 |
            ((k2 >>> j+1) & 0x01) << 6 |
            ((k3 >>> j+1) & 0x01) << 7 );
        }
        byte[] out = sBox(in, box);
        byte[] key = new byte[16];
        for (int i = 3; i >= 0; i--) {
            for(int j = 0; j < 4; j++) {
                key[3-i] |= (out[i*4+j] & 0x01) << (j*2) | ((
                   out[i*4+j] >>> 4) & 0x01) << (j*2+1) ;
                key[7-i] |= ((out[i*4+j] >>> 1) & 0x01) << (j
                   *2) | ((out[i*4+j] >>> 5) & 0x01) << (j*2+1)
                   ;
                key[11-i] |= ((out[i*4+j] >>> 2) & 0x01) << (j
                   *2) | ((out[i*4+j] >>> 6) & 0x01) << (j*2+1)
                   ;
                key[15-i] |= ((out[i*4+j] >>> 3) & 0x01) << (j
                   *2) | ((out[i*4+j] >>> 7) & 0x01) << (j*2+1)
                   ;
            }
        }
        return initPermutation(key);
    }
}
```

6.2 Optimized Code

References

[Anderson, R., Biham, E., & Knudsen, L. 1998] Anderson, R., Biham, E., & Knudsen, L. (1998). Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, 123. Retrieved from ftp://ftp-prod-srv04.it.su.se/pub/security/docs/crypt/Ross_Anderson/serpent.pdf