Project Report

SERPENT CIPHER:
THE SECOND FINALIST IN AES CONTEST

CSE2008-NETWORK SECURITY

Submitted by

17BCI0044 - SHUBHAM GUPTA

17BCI0058 – SAURABH AMBAR

17BCI0145- KUMAR ABHISHEK

Under the guidance of

Prof. Chandra Mohan B

Bachelor of Technology

in

Computer Science and Engineering

Spec. in Information Security

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computing Science and Engineering

TABLE OF CONTENTS

Introduction: -

Serpent is a symmetric key block cipher which comes at second place in the Advanced Encryption Standard (AES) contest after Rijndael.

Serpent has a block size of 128 bits and support a key size of 256 bits. Basically, it is a 32-round permutation-substitution network which operates on a block of four 32-bit words which is total to 128 bits. Each round applies one of eight 4-bit by 4-bit S-boxes in parallel fashion 32 times.it was designed so that all the operations of the cipher can be executed in parallel, using 32 1-bit slices. It is done to maximize parallelism.

It is inspired by ideas for bit slice implementation of ciphers. However, unlike the bit slice implementation of DES, which encrypts 64 different blocks in parallel in order to gain extra speed, Serpent is designed to allow a single block to be encrypted efficiently by bit slicing. This allows the usual modes of operations to be used, so there is no need to change the environment to gain the extra speed. It achieves its high performance by a design that makes very efficient use of parallelism, and this extends beyond the level of the algorithm itself.

Literature Survey:

Performance in various environment: On a 133MHz Pentium/MMX processor, our bitslice implementation of Serpent runs about as fast as DES: it encrypts 9,791,000 bits per second, or about 1738 clock cycles per block, while the best optimized DES implementation (Eric Young's Libdes) encrypts 9,824,864 bits per second. We estimate that on the NIST platform of a 200 MHz Pentium, it will run at about 14.7 Mbit/s. Our bitslice Java implementation performs 10,000 encryptions in 3.3 seconds on a 133 MHz Pentium MMX which translates to 388 kbit/s, and we expect 583 kbit/s on a 200 MHz machine.

Advocating for serpent cipher as AES(by Ross J. Anderson): Serpent should be chosen because it is most secure of the AES finalist. Not only does it have ample safety margin, but its simple structure enables us to be secure that none of the currently known attack will work. It is also simple to check that an implementation is correct.

Design of Serpent cipher: In the first design, the linear transformations were just bit permutations, which were applied as rotations of the 32-bit words in the bitslice implementation. In order to ensure maximal avalanche, the idea was to choose these rotations in a way that ensured maximal avalanche in the fewest number of rounds. Thus, we chose three rotations at each round: we used (0, 1, 3, 7) for the even rounds and (0, 5, 13, 22) for the odd rounds. The reason for this was that

(a) rotating all four words is useless

(b) a single set of rotations did not suffice for full avalanche

(c) these sets of rotations have the property that no difference of pairs in either of them coincides with a difference either in the same set or the other set.

However, we felt that the avalanche was still slow, as each bit affected only one bit in the next round, and thus one active S-box affected only 2–

4 out of the 32 S-boxes in the next round. As a result, we had to use 64 rounds, and the cipher was only slightly faster than triple-DES. So we moved to a more complex linear transformation; this improved the avalanche, and analysis showed that we could now reduce the number of rounds to 32. We believe that the final result is a faster and yet more secure cipher.

We also considered replacing the XOR operations by seemingly more complex operations, such as additions. We did not do this for two major reasons: (1) Our analysis takes advantage of the independence of the bits in the XOR operation, as it allows us to describe the cipher in a standard way, and use the known kinds of analysis. This would not hold if the XOR operations were replaced; (2) in some other ciphers the replacement of XORs by additions (or other operations) has turned out to weaken the cipher, rather than strengthening it.

The first published version of Serpent reused the S-boxes from DES. After this was presented at Fast Software Encryption 98, we studied a number of other linear transformations and S-boxes. We found that it was easy to construct S-boxes that gave much greater security. We were aware that any improvement might come at the expense of the public confidence generated by reusing the DES S-boxes. However, the possible improvement in security was simply too great to forego. We therefore decided to counter the fear of trapdoors by generating the new S-boxes in a simple deterministic way. Having decided not to use the full set of DES S-boxes, we were also free to change from 32 S-boxes to 8, which greatly reduces the complexity of hardware and microcontroller firmware implementations.

After the first version of Serpent was published, some people commented that the key schedule seemed overdesigned. On reflection we agreed; it was particularly heavy for hardware implementation. Much of the complexity came from using the S-boxes to operate on distant rather than consecutive words of the prekey, in order to ‘minimize the key leakage in

the event of a differential attack 22 on the last few rounds of the cipher'. But given the enormous safety margins against differential attack provided by the new S-boxes, we decided that it was safe to discard this feature. Using consecutive inputs to the key schedule S-boxes means that round keys can be computed 'on the fly' in hardware without any significant memory overhead. This further reduces the gate complexity of hardware implementations.

Working of the Serpent Cipher

The Cipher:

Serpent is a 32-round SP-network operating on four 32-bit words, thus giving a block size of 128 bits. All values used in the cipher are represented as bitstreams. The indices of the bits are counted from 0 to bit 31 in one 32-bit word, 0 to bit 127 in 128-bit blocks, 0 to bit 255 in 256-bit keys, and so on. For internal computation, all values are represented in little-endian, where the first word (word 0) is the least significant word, and the last word is the most significant, and where bit 0 is the least significant bit of word 0. Externally, we write each block as a plain 128-bit hex number.

Serpent encrypts a 128-bit plaintext P to a 128-bit ciphertext C in 32 rounds under the control of 33 128-bit subkeys $K_{0,}$…..,K_{32}. The user key length is variable, but for

the purposes of this submission we fix it at 128, 192 or 256 bits; short keys with less than 256 bits are mapped to full-length keys of 256 bits by appending one “1” bit to the MSB end, followed by as many “0” bits as required to make up 256 bits. This mapping is designed to map every short key to a full-length key, with no two short keys being equivalent.

The cipher is consisting of:

- An initial permutation IP.
- 32 rounds, each consisting of a key mixing operation, a passthrough S-boxes, and (in all but the last round) a linear transformation. In the last round, this linear transformation is replaced by an additional key mixing operation.
- A final permutation FP.

The initial and final permutations do not have any cryptographic significance. They are used to simplify an optimized implementation of the cipher and are used to improve its computational efficiency.

The S-box:

The S-boxes of Serpent are 4-bit permutations with the following properties:

- each differential characteristic has a probability of at most 1=4, and a one-bit input difference will never lead to a one-bit output difference;

- each linear characteristic has a probability in the range 1=2 _ 1=4, and a linear relation between one single bit in the input and one single bit in the output has a probability in the range 1=2 ± 1=8;

- the nonlinear order of the output bits as a function of the input bits is the maximum, namely 3.

- The S-boxes were generated in the following manner, which was inspired by RC4. We used a matrix with 32 arrays each with 16 entries. The matrix was initialised with the 32 rows of the DES S-boxes and transformed by swapping the entries in the r^{th} array depending on the value of the entries in the $(r+1)^{st}$ array and on an initial string representing a key. If the resulting array has the desired (differential and linear) properties, save the array as a Serpent S-box.

- Repeat the procedure until 8 S-boxes have been generated. More formally, let serpent[°] be an array containing the least significant four bits of each of the 16 ASCII characters in the expression “sboxesforserpent".

Pseudo code to generate s-box:

```
index := 0
repeat
  currentsbox := index modulo 32;
  for i:=0 to 15 do
    j := sbox[(currentsbox+1) modulo 32][serpent[i]];
    swapentries (sbox[currentsbox][i],sbox[currentsbox][j]);
  if sbox[currentsbox][.] has the desired properties, save it;
  index := index + 1;
until 8 S-boxes have been generated
```

Encryption:

The initial permutation IP is applied to the plaintext P giving B_0, which is the input to the first round. The rounds are numbered from 0 to 31, where the first round is round 0 and the last is round 31. The output of the first round (round 0) is B_1, the output of the second round (round 1) is B_2, the output of round i is B_{i+1}, and so on, until the output of the last round (in which the linear transformation is replaced by an additional key mixing) is denoted by B_{32}. The final permutation FP is now applied to give the ciphertext C.

Each round function R_i ($i \in \{0,...,31\}$) uses only a single replicated S-box. For example, R_0 uses S_0, 32 copies of which are applied in parallel. Thus the first copy of S_0 takes bits 0, 1, 2 and 3 of$\hat{}$ $B_0 \oplus K_0$ as its input and returns as output the first four bits of an intermediate vector. The next copy of S_0 inputs bits 4–7 of $B_0 \oplus K_0$ and returns the next four bits of the intermediate vector, and so on. The intermediate vector is then transformed using the linear transformation, giving B_1. Similarly, R_1 uses 32 copies of S_1 in parallel on $B_0 \oplus K_0$ and transforms their output using the linear transformation, giving B_2.

The set of eight S-boxes is used four times. Thus after using S_7 in round 7, we use S_0 again in round 8, then S_1 in round 9, and so on. The last round R_{31} is slightly different from the others. Here, we apply S_7 on $B_{31} \oplus K_{31}$, and XOR the result with K_{32} rather than applying the linear transformation. The result $\hat{}$ B_{32} is then permuted by FP, giving the ciphertext. Thus the 32 rounds use 8 different S-boxes each of which maps four input bits to four output bits. Each S-box is used in precisely four rounds, and in each of these it is used 32 times in parallel.

The equation for the cipher is:

$$B_0 := IP(P)$$

$$B_{i+1} := R_i(B_i)$$

$$C := FP(B_{32})$$

Where

$R_i(X) = L(S_i(X \oplus K_i))$ $\quad i = 0 \ldots\ldots 30$

$R_i(X) = S_i(X \oplus K_i) \oplus K_{32}$ $\quad i = 31$

where S_i is the application of the S-box $S_{i \bmod 8}$ 32 times in parallel, and Lis the linear transformation.

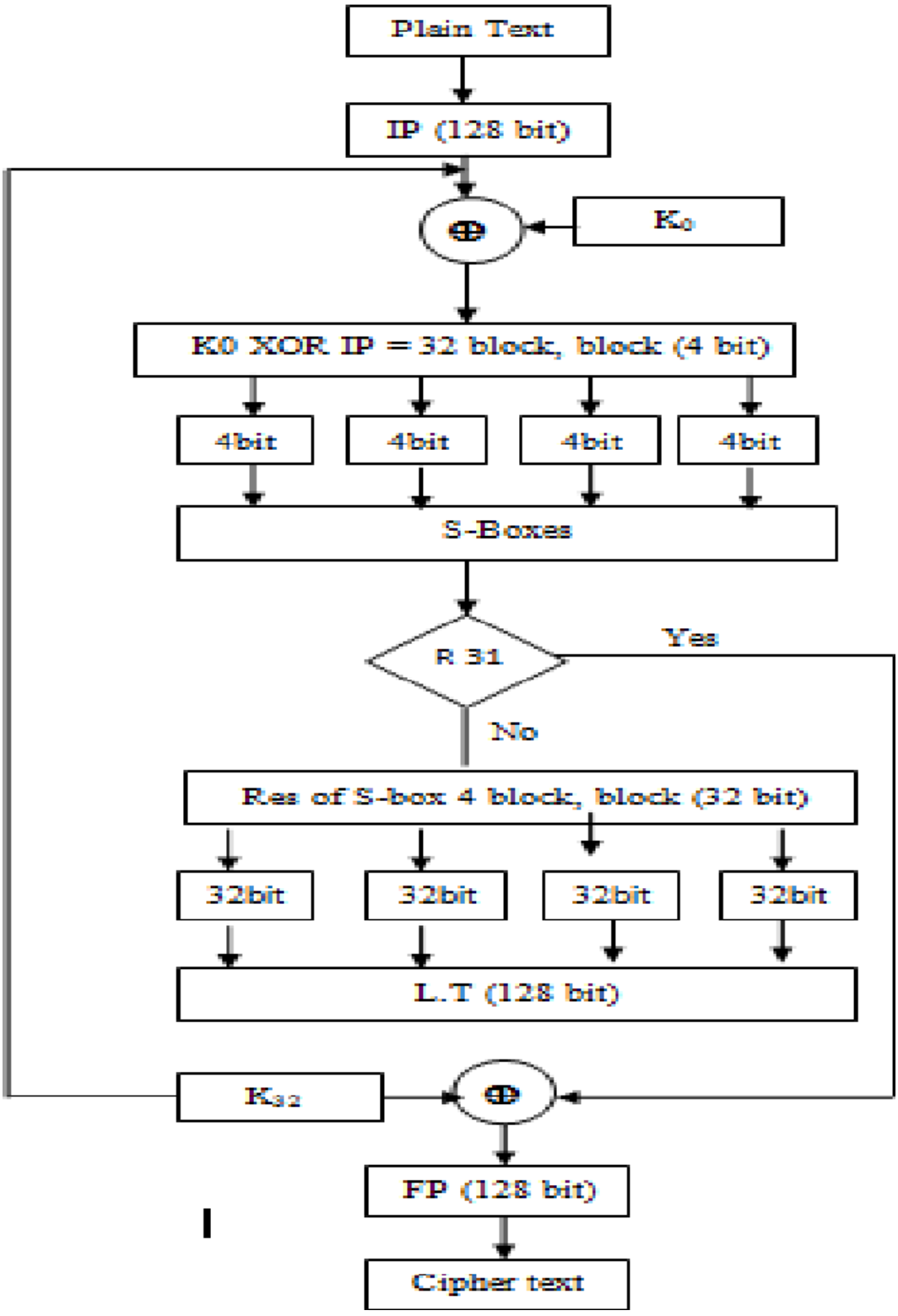
The initial permutation:

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

The Final Permutation:

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

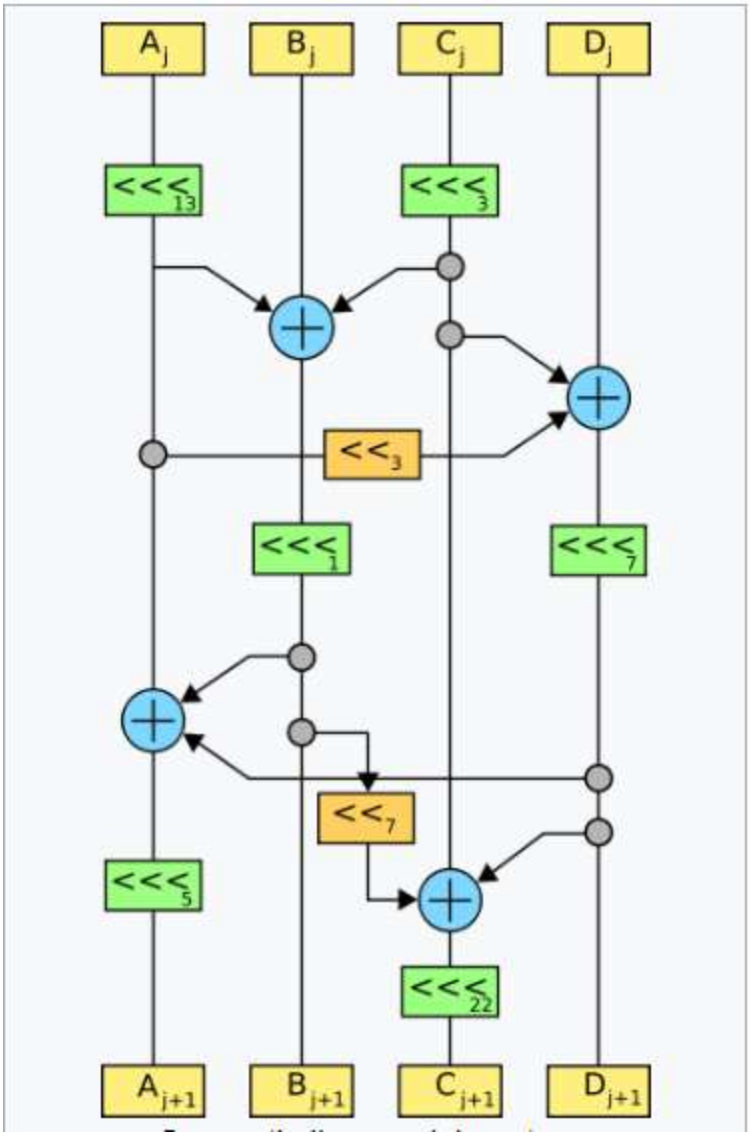
The Flow Graph for Encryption



The three operations for encryption are:

1. Key Mixing: At each round, a 128-bit subkey K_i is exclusive-or with the current intermediate data B_i.
2. S-Boxes: The 128-bit combination of input and key is considered as four 32-bit words. The S-box, which is implemented as a sequence of logical operations is applied to these four words, and the result is four output words. The CPU is thus employed to execute the 32 copies of the S-box simultaneously, resulting with $S_i(B_i \bigoplus K_i)$.
3. Linear Transformation: The 32 bits in each of the output words are linearly mixed, by

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



Decryption:

Decryption is different from encryption in that the inverse of the S-boxes must be used in the reverse order, as well as the inverse linear transformation and reverse order of the subkeys.

Key:

We write the key K as eight 32-bit words w−8,..., w−1 and expand these to an intermediate key (which we call prekey) w0,...,w131 by the following affine recurrence:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11$$

where φ is the fractional part of the golden ratio (√5 + 1)/2 or0x9e3779b9 in hexadecimal. The underlying polynomial $x_8 + x_7 + x_5 + x_3 + 1$ is primitive, which together with the addition of the round index is chosen to ensure an even distribution of key bits throughout the rounds, and to eliminate weak keys and related keys.

The round keys are now calculated from the prekeys using the S-boxes, again in bitslice mode. We use the S-boxes to transform the prekeys w_i into words k_i of round key in the following way:

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &:= S_3(w_0, w_1, w_2, w_3) \\
\{k_4, k_5, k_6, k_7\} &:= S_2(w_4, w_5, w_6, w_7) \\
\{k_8, k_9, k_{10}, k_{11}\} &:= S_1(w_8, w_9, w_{10}, w_{11}) \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &:= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &:= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\dots \\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &:= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &:= S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}$$

Security:

he XSL attack, if effective, would weaken Serpent (though not as much as it would weaken Rijndael, which became AES). However, many cryptanalysts believe that once implementation considerations are taken into account the XSL attack would be more expensive than a brute force attack.

In 2000, a paper by Kohno et al. presents a meet-in-the-middle attack against 6 of 32 rounds of Serpent and an amplified boomerang attack against 9 of 32 rounds in Serpent.

A 2001 attack by Eli Biham, Orr Dunkelman and Nathan Keller presents a linear cryptanalysis attack that breaks 10 of 32 rounds of Serpent-128 with 2^{118} known plaintexts and 2^{89} time, and 11 rounds of Serpent-192/256 with 2^{118} known plaintexts and 2^{187} time.

A 2009 paper has noticed that the nonlinear order of Serpent S-boxes were not 3 as was claimed by the designers.

A 2011 attack by Hongjun Wu, Huaxiong Wang and Phuong Ha Nguyen, also using linear cryptanalysis, breaks 11 rounds of Serpent-128 with 2^{116} known plaintexts, $2^{107.5}$ time and 2^{104}memory.

The same paper also describes two attacks which break 12 rounds of Serpent-256. The first requires 2^{118} known plaintexts, $2^{228.8}$ time and 2^{228} memory.

Software Used:

C++, ubuntu terminal

Implementation:

Code:

```
#include <iostream>
#include <cstdlib>
#include <getopt.h>
#include <string.h>

using namespace std;

#define ROUNDS 32
#define BLOCK_LEN 16
#define KEY_LEN 32
#define GOLDEN_RATIO 0x9E3779B9L

#define U32V(v) ((uint32_t)(v) & 0xFFFFFFFFUL)
#define ROTL32(v, n) (U32V((v) << (n)) | ((v) >> (32 - (n))))
#define ROTR32(v, n) ROTL32(v, 32 - (n))
#define HI_NIBBLE(b) (((b) >> 4) & 0x0F)
#define LO_NIBBLE(b) ((b) & 0x0F)

typedef union _serpent_blk_t {
    uint8_t b[BLOCK_LEN];
    uint32_t w[BLOCK_LEN/4];
```

```
    uint64_t q[BLOCK_LEN/2];
} serpent_blk;

typedef uint32_t serpent_subkey_t[4];

typedef struct serpent_key_t {
serpent_subkey_t x[ROUNDS+1];
} serpent_key;

void permute(serpent_blk* out, serpent_blk* in, bool initial) {
    uint8_t carry;

    for (int i = 0; i< BLOCK_LEN / 4; i++) {
        out->w[i] = 0;
    }

    if (initial) {
        for (int i = 0; i< BLOCK_LEN; i++) {
            for (int j = 0; j < BLOCK_LEN / 2; j++) {
                carry = in->w[j % 4] & 1;
                in->w[j % 4] >>= 1;
                out->b[i] = (carry << 7) | (out->b[i] >> 1);
            }
        }
    } else {
        for (int i = 0; i< BLOCK_LEN / 4; i++) {
            for (int j = 0; j < BLOCK_LEN * 2; j++) {
```

```
            carry = in->w[i] & 1;
            in->w[i] >>= 1;
            out->w[j % 4] = (carry << 31) | (out->w[j % 4] >> 1);
          }
        }
    }
}

void subbytes (serpent_blk *blk, uint32_t box_idx, bool encryption)
{
serpent_blktmp_blk, sb;
    uint8_t *sbp;
    uint8_t i, t;

    uint8_t sbox[8][8] =
{ { 0x83, 0x1F, 0x6A, 0xB5, 0xDE, 0x24, 0x07, 0xC9 },
{ 0xCF, 0x72, 0x09, 0xA5, 0xB1, 0x8E, 0xD6, 0x43 },
{ 0x68, 0x97, 0xC3, 0xFA, 0x1D, 0x4E, 0xB0, 0x25 },
{ 0xF0, 0x8B, 0x9C, 0x36, 0x1D, 0x42, 0x7A, 0xE5 },
{ 0xF1, 0x38, 0x0C, 0x6B, 0x52, 0xA4, 0xE9, 0xD7 },
{ 0x5F, 0xB2, 0xA4, 0xC9, 0x30, 0x8E, 0x6D, 0x17 },
{ 0x27, 0x5C, 0x48, 0xB6, 0x9E, 0xF1, 0x3D, 0x0A },
{ 0xD1, 0x0F, 0x8E, 0xB2, 0x47, 0xAC, 0x39, 0x65 }
        };

    uint8_t sbox_inv[8][8] =
{ { 0x3D, 0x0B, 0x6A, 0xC5, 0xE1, 0x74, 0x9F, 0x28 },
```

```
{ 0x85, 0xE2, 0x6F, 0x3C, 0x4B, 0x97, 0xD1, 0x0A },
{ 0x9C, 0x4F, 0xEB, 0x21, 0x30, 0xD6, 0x85, 0x7A },
{ 0x90, 0x7A, 0xEB, 0xD6, 0x53, 0x2C, 0x84, 0x1F },
{ 0x05, 0x38, 0x9A, 0xE7, 0xC2, 0x6B, 0xF4, 0x1D },
{ 0xF8, 0x92, 0x14, 0xED, 0x6B, 0x35, 0xC7, 0x0A },
{ 0xAF, 0xD1, 0x35, 0x06, 0x94, 0x7E, 0xC2, 0xB8 },
{ 0x03, 0xD6, 0xE9, 0x8F, 0xC5, 0x7B, 0x1A, 0x24 }
        };

box_idx&= 7;

    if (encryption) {
sbp=(uint8_t*)&sbox[box_idx][0];
    } else {
sbp=(uint8_t*)&sbox_inv[box_idx][0];
    }

    for (i=0; i<16; i+=2) {
        t = sbp[i/2];
sb.b[i+0] = LO_NIBBLE(t);
sb.b[i+1] = HI_NIBBLE(t);
    }

    permute (&tmp_blk, blk, true);

    for (i = 0; i< BLOCK_LEN; i++) {
        t = tmp_blk.b[i];
```

```
tmp_blk.b[i] = (sb.b[HI_NIBBLE(t)] << 4) | sb.b[LO_NIBBLE(t)];
    }

    permute (blk, &tmp_blk, false);
}

void whiten(serpent_blk *dst, serpent_key *key, int idx) {
    for (int i = 0; i< BLOCK_LEN / 4; i++) {
dst->w[i] ^= key->x[idx][i];
    }
}

void linear_trans(serpent_blk* output, bool encryption) {
    uint32_t x0 = output->w[0];
    uint32_t x1 = output->w[1];
    uint32_t x2 = output->w[2];
    uint32_t x3 = output->w[3];

    if (encryption) {
        x2 = ROTL32(x2, 10);
        x0 = ROTR32(x0, 5);
        x2 ^= x3 ^ (x1 << 7);
        x0 ^= x1 ^ x3;
        x3 = ROTR32(x3, 7);
        x1 = ROTR32(x1, 1);
        x3 ^= x2 ^ (x0 << 3);
        x1 ^= x0 ^ x2;
```

```
        x2 = ROTR32(x2,  3);
        x0 = ROTR32(x0, 13);
    } else {
        x0 = ROTL32(x0, 13);
        x2 = ROTL32(x2,  3);
        x1 ^= x0 ^ x2;
        x3 ^= x2 ^ (x0 << 3);
        x1 = ROTL32(x1, 1);
        x3 = ROTL32(x3, 7);
        x0 ^= x1 ^ x3;
        x2 ^= x3 ^ (x1 << 7);
        x0 = ROTL32(x0, 5);
        x2 = ROTR32(x2, 10);
    }

    output->w[0] = x0;
    output->w[1] = x1;
    output->w[2] = x2;
    output->w[3] = x3;
}

uint32_t gen_w (uint32_t *b, uint32_t i) {
    uint32_t ret;
    ret = b[0] ^ b[3] ^ b[5] ^ b[7] ^ GOLDEN_RATIO ^ i;
    return ROTL32(ret, 11);
}
```

```
void key_setup (serpent_key *key, void *input)
{
   union {
      uint8_t b[32];
      uint32_t w[8];
   } s_ws;

   uint32_t i, j;

   // copy key input to local buffer
memcpy (&s_ws.b[0], input, KEY_LEN);

   // expand the key
   for (i=0; i<=ROUNDS; i++) {
      for (j=0; j<4; j++) {
         key->x[i][j] = gen_w (s_ws.w, i*4+j);
memmove (&s_ws.b, &s_ws.b[4], 7*4);
s_ws.w[7] = key->x[i][j];
      }

subbytes((serpent_blk*)&key->x[i], 3 - i, true);
   }
}

size_t hex2bin (void *bin, const char hex[]) {
size_tlen, i;
   int x;
```

```
    uint8_t *p=(uint8_t*)bin;

len = strlen (hex);

    if ((len& 1) != 0) {
        return 0;
    }

    for (i=0; i<len; i++) {
        if (isxdigit((int)hex[i]) == 0) {
            return 0;
        }
    }

    for (i=0; i<len / 2; i++) {
sscanf (&hex[i * 2], "%2x", &x);
        p[i] = (uint8_t)x;
    }
    return len / 2;
}

void dump_hex (uint8_t bin[], int len)
{
    for (int i = 0; i<len; i++) {
printf ("%02x", bin[i]);
    }
}
```

```
void dump_str(uint8_t bin[], int len) {
   for (int i = 0; i<len; i++) {
printf ("%c", bin[i]);
   }
}

void encrypt(serpent_blk* blk, serpent_key* key) {
   // repeat 32 rounds (0 - 31)
cout<<"encryption of :- ";
   for (int i = 0; i< ROUNDS; i++) {
whiten(blk, key, i);
subbytes(blk, i, true);
       if (i< ROUNDS - 1) {
           // final round, no linear trans
linear_trans(blk, true);
       }
   }
whiten(blk, key, ROUNDS);
}

void decrypt(serpent_blk* blk, serpent_key* key) {
   // round 31 - 0 (inverse)

cout<<"decryption: ";
whiten(blk, key, ROUNDS);
   for (int i = ROUNDS - 1; i>= 0; i--) {
```

```
subbytes(blk, i, false);
whiten(blk, key, i);
        if (i != 0) {
            // final round (round 0)
linear_trans(blk, false);
        }
    }
}

void generate(uint8_t key[64]) {
    //srand((unsigned int)time(NULL));
    for (int i = 0; i< 64; i++) {
        key[i] = rand() % 256;
    }
}

int main(int argc, char* argv[]) {
    char* message;
          char* key_input;
          int c;
    bool random_key_flag = false, key_flag = false, encrypt_flag = false, decrypt_flag
= false;
serpent_blk blk;
    uint8_t key[64];
serpent_keyskey;

          while (1) {
```

```
            static struct option long_options[] = {
        {"generate", no_argument, 0, 'g'},
        {"encrypt", required_argument, 0, 'e'},
        {"decrypt", required_argument, 0, 'd'},
        {"key", required_argument, 0, 'k'},
        {0, 0, 0, 0}
            };
            int option_index = 0;
            c = getopt_long(argc, argv, "ge:d:k:", long_options, &option_index);
            if (c == -1)
                break;
            if (c == 'g') {
                random_key_flag = true;
            }
            if (c == 'e') {
                message = optarg;
                encrypt_flag = true;
            }
            if (c == 'd') {
                message = optarg;
                decrypt_flag = true;
            }
            if (c == 'k') {
key_flag = true;
                key_input = optarg;
            }
        }
```

```
    if (random_key_flag) {
        generate(key);
dump_hex(key, 32);
        return 0;
    }

memset(key, 0, sizeof(key));
    if (key_flag) {
        if (strlen(key_input) % 64 != 0) {
cout<< "Key length must be 256-bit." <<endl;
            return 0;
        } else {
            hex2bin(key, key_input);
        }
    } else {
        generate(key);
    }

    if (encrypt_flag) {
key_setup(&skey, key);
        for (int i = 0; i<strlen(message); i += 16) {
            for (int j = 0; j < 16; j++) {
                if ((i + j) >= strlen(message)) {
blk.b[j] = 0;
                } else {
blk.b[j] = message[i + j];
```

```
            }
        }
encrypt(&blk, &skey);
dump_hex(blk.b, 16);
    }
cout<<endl;
  }

  if (decrypt_flag) {
key_setup(&skey, key);
    for (int i = 0; i<strlen(message); i += 32) {
        for (int j = 0; j < 32; j += 2) {
            char c[3];
c[0] = message[i + j];
c[1] = message[i + j + 1];
c[2] = '\0';
blk.b[j / 2] = strtol(c, NULL, 16);
        }
decrypt(&blk, &skey);
dump_str(blk.b, 16);
    }
cout<<endl;
  }
  return 0;
}
```

Result:

To compile: g++ -o out serpent.cpp

For key generation: ./out -- generate

Output:
8265500565a4955b050b7ff2f46edf6e3744749331980c05a5ae57ac80f4e202

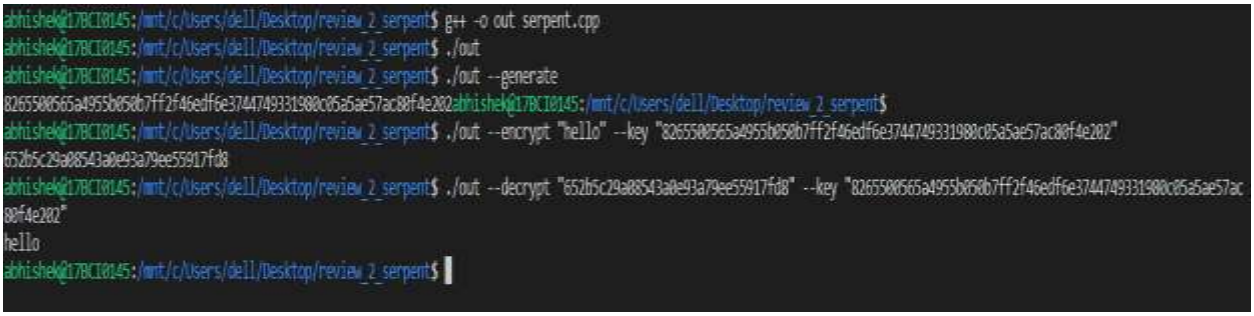
For encryption: ./out --encrypt “hello” --key “8265500565a4955b050b7ff2f46edf6e3744749331980c05a5ae57ac80f4e202”

Output: 652b5c29a08543a0e93a79ee55917fd8

For Decryption: ./out --decrypt “652b5c29a08543a0e93a79ee55917fd8” --key “8265500565a4955b050b7ff2f46edf6e3744749331980c05a5ae57ac80f4e202”

Output: hello

Snippet:



Conclusion: -

We have presented a cipher which is as fast as DES, and we believe it to be more secure than three-keytriple DES. It is believe that it would still be as secure as three-key triple DES if its number of rounds were reduced by half. Its security is partially based on the reuse of the thoroughly studied components of DES, and even although the final version of the cipher no longer reuses the DES S-boxes, we can still draw on the wide literature of block cipher cryptanalysis. Our design strategy should also give a high level of confidence that we have not inserted any trapdoors. The algorithm's performance comes from allowing an efficient bitslice implementation on a range of processors, including the market leading Intel/MMX and compatible chips.

References:

1. DG Abraham, GM Dolan, GP Double, JV Stevens, \Transaction Security System", in IBM Systems Journal v 30 no 2 (1991) pp 206-229

2. RJ Anderson, \UEPS | a Second Generation Electronic Wallet" in Computer Security | ESORICS 92, Springer LNCS vol 648 pp 411-418

3. RJ Anderson, SJ Bezuidenhoudt, \On the Reliability of Electronic Payment Systems", in IEEE Transactions on Software Engineering v 22 no 5 (May 1996) pp 294-301

4. RJ Anderson, MG Kuhn, \Tamper Resistance | a Cautionary Note", in The Second USENIX Workshop on Electronic Commerce Proceedings (Nov 1996) pp 1-11 5. RJ Anderson, MG Kuhn, \Low Cost Attacks on Tamper Resistant Devices", in Security Protocols | Proceedings of the 5th International Workshop, Springer LNCS v 1361 pp 125-136

6. E Biham, \A Fast New DES Implementation in Software", in Fast Software Encryption | 4th International Workshop, FSE '97, Springer LNCS v 1267 pp 260-2717. E Biham, \Higher Order Di_erential Cryptanalysis", unpublished paper, 19948. E Biham, How to Forge DES-Encrypted Messages in 228 Steps, Technical ReportCS884, Technion, August 1996.