Cryptography and computer security 2017/2018
The Serpent block cipher
Matej Horvat (matej.horvat@guest.arnes.si)

1. Introduction

Serpent is a block cipher developed in 1998 by Ross Anderson, Eli Biham, and Lars Knudsen for the NIST AES competition. It was one of the finalists but ended up in second place, receiving 59 votes while Rijndael received 86 [RA].

This paper describes the Serpent algorithm, shows examples of how it can be implemented, and discusses its security.

One place where Serpent is used is the widely known TrueCrypt disk encryption program and its derivatives (e.g. VeraCrypt). [VC]

2. High level structure

Serpent uses a substitution-permutation network to encrypt a 128-bit block with a key that is up to 256 bits long. The encryption is done in 32 rounds (regardless of key length) where each round consists of [SP]:

- a key mixing step, during which the round subkey is XOR-ed with the data,
- a substitution, during which bits of data are substituted with other bits as specified in the S-boxes,
- a linear transformation (described later), or, for the last round, an additional key mixing step.

The decryption is the reverse of encryption, with the linear transformation and S-boxes being replaced by their inverses.

There are two ways to implement Serpent: the "basic" way, which is perhaps easier to implement in hardware, but very inefficient in software, and the "bitslice" way, which is designed to be efficient on pipelined 32-bit processors [SP]. We will first describe the parts that are common to both, then describe each one's specifics. We will show code that implements both ways in Pascal (compilable with Turbo Pascal and Free Pascal).

We start with the following type definitions:

```
type
  ty128 = array [0 .. 3] of longint;     {128-bit vector.}
  pty128 = ^ty128;                       {Pointer to 128-bit vector.}
  ty256 = array [0 .. 7] of longint;     {256-bit vector.}
  tySubkeys = array [0 .. 32] of ty128;  {Subkeys.}
  tySBox = array [0 .. 15] of byte;      {One S-box or inverse S-box.}
  ptySBox = ^tySBox;                     {Pointer to an S-box or inverse S-box.}
  tySBoxes = array [0 .. 7] of tySBox;   {Eight S-boxes or inverse S-boxes.}
```

2.1. Key expansion

The algorithm only works with 256-bit keys internally. Keys shorter than 256 bits are expanded to 256 bits by prefixing the key with a "1" bit, then prefixing that with "0" bits so the key is 256 bits long. This way, every short key has a unique long key [SP]. For example:

- "0" becomes "0...010".
- "1" becomes "0...011".
- "00" becomes "0...0100".

In our implementation, we only support 256-bit keys.

2.2. Prekey and subkey generation

The prekey is an array of 132 32-bit words iteratively generated from the 256-bit key with the following assignment (*i* goes from 0 to 131). There are an additional 8 elements (numbered -8 to -1) which are simply the first 8 32-bit words of the 256-bit key. [SP]

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

where ϕ is the fractional part of the golden ratio $(\sqrt{5}+1)/2$ or `0x9e3779b9`

Each subkey (round key) that will be XOR-ed with the data at the start of every encryption round then simply consists of 4 consecutive 32-bit words of the prekey, which are put through S-boxes beforehand [SP].

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

```
procedure PrepareSubkeys (var Key: ty256; var Subkeys: tySubkeys);
{Prepares the subkeys from the given key. Source: page 7 of [SP].}
var
  Prekey: array [-8 .. 131] of longint;
  i: integer;

procedure PrepareSubkey (WhichSubkey: integer; var Subkey: ty128)
  {$IfDef FPC} inline {$EndIf};
{Prepares a subkey (0 to 32; the last being used by the final encryption round
(or first decryption round) only). Source: page 7 of [SP].}
var
  i: integer;
begin
  for i := 0 to 3 do
    Subkey[i] := Prekey[WhichSubkey shl 2 + i];

  Substitute(Subkey, @kSBox[(3 - WhichSubkey) and 7]);
end;

begin
  {Prepare the prekey.}
  for i := -8 to -1 do
    Prekey[i] := Key[i + 8];
  for i := 0 to 131 do
    Prekey[i] := Rotate(
```

```
      Prekey[i - 8] xor
      Prekey[i - 5] xor
      Prekey[i - 3] xor
      Prekey[i - 1] xor
      $9E3779B9 xor
      i,
      11
    );

  {Prepare the subkeys.}
  for i := 0 to 32 do
    PrepareSubkey(i, Subkeys[i]);
end;
```

2.3. S-boxes

Serpent uses 8 4-bit S-boxes. Each S-box has its own inverse S-box. [SP]

When processing a 128-block (a data block or a subkey), the words of the block are put through an S-box in a column-wise manner: if we arrange the words into a binary matrix of 4 rows and 32 columns, then each 4-bit column is substituted with a different column according to the S-box.

```
const
  kSBox: tySBoxes =
    {S-boxes; source: [SP], page 21}
    (
      (3, 8, 15, 1, 10, 6, 5, 11, 14, 13, 4, 2, 7, 0, 9, 12),
      (15, 12, 2, 7, 9, 0, 5, 10, 1, 11, 14, 8, 6, 13, 3, 4),
      ...
      (7, 2, 12, 5, 8, 4, 6, 11, 14, 9, 1, 15, 13, 3, 10, 0),
      (1, 13, 15, 0, 14, 8, 2, 11, 7, 4, 12, 10, 9, 3, 5, 6)
    );
  kInvSBox: tySBoxes =
    {Inverse S-boxes; source: [SP], page 21}
    (
      (13, 3, 11, 0, 10, 6, 5, 12, 1, 14, 4, 7, 15, 9, 8, 2),
      (5, 8, 2, 14, 15, 6, 12, 3, 11, 4, 7, 9, 1, 13, 10, 0),
      ...
      (15, 10, 1, 13, 5, 3, 6, 0, 4, 9, 14, 7, 2, 12, 8, 11),
      (3, 0, 6, 13, 9, 14, 15, 8, 5, 12, 11, 7, 10, 1, 4, 2)
    );

procedure Substitute (var Block: ty128; SBox: ptySBox);
{Applies an S-box or inverse S-box to a 128-bit vector column-wise; the most
significant word is the top row.}
var
  i: integer;
  j: integer;
  S: longint;
  Temp: ty128;
begin
  for i := 0 to 3 do
    Temp[i] := 0;

  for j := 0 to 31 do begin
    {Collect bits from the column.}
    S := 0;
    for i := 0 to 3 do
      S := S or GetBit32(Block[i], j) shl i;
```

```
    {Put them through the S-box.}
    S := SBox^[S];

    {Put them in the result column.}
    for i := 0 to 3 do
      Temp[i] := Temp[i] or GetBit32(S, i) shl j;
  end;

  CopyBlock(@Temp, Block);
end;
```

3. A basic implementation of the algorithm

The basic implementation is very slow, but possibly easier to understand and easier to implement in hardware than the optimized ones.

3.1. Initial and final permutations

The basic way requires permuting the 128-bit data block and all subkeys before and after encryption or decryption to give the same results as the bitslice way [SP].

If we imagine the input block (or any subkey) as a matrix of 4 rows and 32 columns, where the first row contains numbers 0 to 31, the second 32 to 63, the third 64 to 95, and the fourth 96 to 127, with these numbers representing bit positions (0 being least significant), then the initial permutation simply transposes the matrix; if we then read the numbers of each row, we get 0, 32, 64, 96, then 1, 33, 65, 97, then 2, 34, 66, 98, and so on.

The final permutation is the inverse of the initial permutation.

```
procedure IP (var Input: ty128; var Output: ty128);
{Does the initial permutation. This is the inverse of the final permutation.
Source: page 19 of [SP].}
var
  i: integer;
begin
  for i := 0 to 127 do
    PutBit128(
      Output,
      i,
      GetBit128(
        Input,
        (i shr 2 + i shl 5) and 127
      )
    );
end;

procedure FP (var Input: ty128; var Output: ty128);
{Does the final permutation. This is the inverse of the initial permutation.
Source: page 19 of [SP].}
var
  i: integer;
begin
  for i := 0 to 127 do
    PutBit128(
      Output,
      i,
      GetBit128(
        Input,
        ((i shl 2) and 127) + i shr 5
```

```
      )
    );
end;
```

3.2. S-boxes

Because the data block is transposed, the S-boxes cannot be applied column-wise on it as they normally would be.

(Subkeys, however, can still be generated the same as before, including the column-wise substitutions; they just have to be transposed with the initial permutation afterwards.)

Instead, we can simply imagine the block as one row of 128 bits, or 32 rows of 4 bits. Each consecutive group of 4 bits is substituted. The easiest way to implement this on a 32-bit processor is to process one word at a time.

```
function SBox32 (Input: longint; Box: ptySBox): longint;
{Returns the results of putting a 32-bit integer into 8 copies of the
specified S-box.}
var
  i: integer;
  Result: longint;
begin
  Result := 0;
  i := 0;
  repeat
    Result := Result or longint(Box^[(Input shr i) and 15]) shl i;
    i := i + 4;
  until i = 32;

  SBox32 := Result;
end;
```

3.3. Encryption

To encrypt a block, we first apply the initial permutation. Then we execute 31 rounds of subkey mixing (XOR-ing), substitution (as just described), and a linear transformation (described next). Finally, we execute one more round, which mixes another subkey instead of the linear transformation, and apply the final permutation.

```
procedure Encrypt (var IOBlock: ty128; var Subkeys: tySubkeys);
{Encrypts a block. Source: page 3 of [SP] for the high level.}
var
  Round: integer;
  Block: ty128;

procedure RoundPrologue;
{Does the operations common to all rounds: XORs the subkey with the block and
puts it through the S-boxes.}
var
  i: integer;
begin
  XORBlockWithSubkey(Block, Subkeys[Round]);

  for i := 0 to 3 do
    Block[i] := SBox32(Block[i], @kSBox[Round and 7]);
end;

begin
  IP(IOBlock, Block);
```

```
  {Rounds 0 do 30.}
  for Round := 0 to 30 do begin
    RoundPrologue;
    LT(Block, @kLT);
  end;

  {Final round.}
  Round := 31;
  RoundPrologue;
  XORBlockWithSubkey(Block, Subkeys[32]);

  FP(Block, IOBlock);
end;
```

3.4. The linear transformation

The linear transformation takes the 128-bit data block and produces a new block. It does not depend on the key. In the basic way, each output bit is obtained by XOR-ing several bits of the input block. Which bits act as inputs for which output bit is specified by a table.

```
type
  tyLT = array [0 .. 127] of array [0 .. 6] of shortint;  {Linear transform.}
  ptyLT = ^tyLT;      {Pointer to a linear or inverse linear transformation.}

const
  kLT: tyLT =
    {Linear transformation; source: [SP], page 19}
    (
      (16, 52, 56, 70, 83, 94, 105),
      (72, 114, 125, -1, -1, -1, -1),
      ...
      (5, 11, 26, 80, 122, 126, -1),
      (32, 86, 99, -1, -1, -1, -1)
    );

procedure LT (var Block: ty128; Table: ptyLT);
{Does the linear or inverse linear transformation according to the specified
table.}
var
  i: integer;
  j: integer;
  Accumulator: Boolean;
  Temp: ty128;
begin
  for i := 0 to 127 do begin
    Accumulator := false;

    j := 0;
    while (j < 7) and (Table^[i][j] <> -1) do begin
      Accumulator := Accumulator <> (GetBit128(Block, Table^[i][j]) <> 0);
      {^ Use <> as XOR since GetBit128 returns 0 or nonzero, not 0 or 1}
      j := j + 1;
    end;

    PutBit128(Temp, i, Ord(Accumulator) {convert to 0 or 1});
  end;

  CopyBlock(@Temp, Block);
end;
```

3.5. Decryption

Decryption is the inverse of encryption. First, we apply the initial permutation (to undo the final permutation), then undo each encryption round by doing the same operations as for encryption, except in reverse, and using the inverse S-boxes and inverse linear transformation. Finally, we apply the final permutation.

```
procedure Decrypt (var IOBlock: ty128; var Subkeys: tySubkeys);
{Decrypts a block.}
var
  Round: integer;
  Block: ty128;

procedure RoundEpilogue;
{Does what is common to all decryption rounds: puts the block through the
inverse S-boxes, then XORs it with the current round subkey.}
var
  i: integer;
begin
  for i := 0 to 3 do
    Block[i] := SBox32(Block[i], @kInvSBox[Round and 7]);

  XORBlockWithSubkey(Block, Subkeys[Round]);
end;

begin
  IP(IOBlock, Block);

  {Undo the final round.}
  XORBlockWithSubkey(Block, Subkeys[32]);
  Round := 31;
  RoundEpilogue;

  {Undo rounds 30 to 0.}
  for Round := 30 downto 0 do begin
    LT(Block, @kInvLT);
    RoundEpilogue;
  end;

  FP(Block, IOBlock);
end;
```

3.6. Inverse linear transformation

The inverse linear transformation is performed with the same algorithm as the linear transformation, except with a different table.

```
const
  kInvLT: tyLT =
    {Inverse linear transformation; source: [SP], page 20}
    (
      (53, 55, 72, -1, -1, -1, -1),
      (1, 5, 20, 90, -1, -1, -1),
      ...
      (11, 98, -1, -1, -1, -1, -1),
      (4, 27, 86, 97, 113, 115, 127)
    );
```

4. A bitslice implementation of the algorithm

In this implementation, the initial and final permutations are not needed because the substitutions are done in a column-wise manner, as when generating the subkeys.

4.1. The linear transformation

The linear transformation is a sequence of shifts, XORs, and left rotations. Its counterpart in the basic implementation is what happens if this form is explicitly written out in equations for each bit.

```
    {The linear transformation.}
    Block[0] := Rotate(Block[0], 13);
    Block[2] := Rotate(Block[2], 3);
    Block[1] := Block[1] xor Block[0] xor Block[2];
    Block[3] := Block[3] xor Block[2] xor Block[0] shl 3;
    Block[1] := Rotate(Block[1], 1);
    Block[3] := Rotate(Block[3], 7);
    Block[0] := Block[0] xor Block[1] xor Block[3];
    Block[2] := Block[2] xor Block[3] xor Block[1] shl 7;
    Block[0] := Rotate(Block[0], 5);
    Block[2] := Rotate(Block[2], 22);
```

4.2. The inverse linear transformation

```
    {The inverse linear transformation. The same steps as in the linear
    transformation are done in reverse and we subtract the rotation amounts
    from 32 so it goes the other way.}
    Block[2] := Rotate(Block[2], 32 - 22);
    Block[0] := Rotate(Block[0], 32 - 5);
    Block[2] := Block[2] xor Block[3] xor Block[1] shl 7;
    Block[0] := Block[0] xor Block[1] xor Block[3];
    Block[3] := Rotate(Block[3], 32 - 7);
    Block[1] := Rotate(Block[1], 32 - 1);
    Block[3] := Block[3] xor Block[2] xor Block[0] shl 3;
    Block[1] := Block[1] xor Block[0] xor Block[2];
    Block[2] := Rotate(Block[2], 32 - 3);
    Block[0] := Rotate(Block[0], 32 - 13);
```

4.3. S-box optimizations

With the linear transformation and its inverse optimized for 32-bit processors and the initial and final permutations removed, the bottleneck of the algorithm become the S-boxes, which are slow as they use loops and only work on a few bits at a time. The paper [SuS] presents a way to optimize S-boxes and their inverses to not require any loops or branches, and in particular to be suitable for use on x86 processors, which only have 2-operand instructions and few registers. If we implement the S-boxes this way, we can get rid of tables for the S-boxes and get:

```
procedure Substitute (var Block: ty128; SBox: integer);
{Applies an S-box to a 128-bit vector. Source: pages 10 to 13 of the "Speeding
up Serpent" paper.}
var
  X0: longint;
  X1: longint;
  X2: longint;
  X3: longint;
  T: longint;  {Temporary register.}
begin
```

```
X0 := Block[0];
X1 := Block[1];
X2 := Block[2];
X3 := Block[3];

case SBox of
  0:
    begin
      X3 := X3 xor X0;
      T := X1;
      X1 := X1 and X3;
      T := T xor X2;
      X1 := X1 xor X0;
      X0 := X0 or X3 xor T;
      T := T xor X3;
      X3 := X3 xor X2;
      Block[2] := X2 or X1 xor T;
      T := not T or X1;
      X1 := X1 xor X3 xor T;
      X3 := X3 or X0;
      Block[0] := X1 xor X3;
      Block[1] := T xor X3;
      Block[3] := X0;
    end;
  1:
    begin
      X0 := not X0;
      X2 := not X2;
      T := X0;
      X0 := X0 and X1;
      X2 := X2 xor X0;
      X0 := X0 or X3;
      X3 := X3 xor X2;
      X1 := X1 xor X0;
      X0 := X0 xor T;
      T := T or X1;
      X1 := X1 xor X3;
      X2 := (X2 or X0) and T;
      X0 := X0 xor X1;
      Block[3] := X1 and X2 xor X0;
      Block[1] := X0 and X2 xor T;
      Block[0] := X2;
      Block[2] := X3;
    end;
  2:
    begin
      T := X0;
      X0 := X0 and X2 xor X3;
      X2 := X2 xor X1 xor X0;
      X3 := X3 or T xor X1;
      T := T xor X2;
      X1 := X3;
      X3 := X3 or T xor X0;
      X0 := X0 and X1;
      T := T xor X0;
      Block[2] := X1 xor X3 xor T;
      Block[3] := not T;
      Block[0] := X2;
      Block[1] := X3;
    end;
  3:
    begin
```

```
    T := X0;
    X0 := X0 or X3;
    X3 := X3 xor X1;
    X1 := X1 and T;
    T := T xor X2;
    X2 := X2 xor X3;
    X3 := X3 and X0;
    T := T or X1;
    X3 := X3 xor T;
    X0 := X0 xor X1;
    T := T and X0;
    X1 := X1 xor X3;
    Block[3] := T xor X2;
    X1 := X1 or X0 xor X2;
    X0 := X0 xor X3;
    Block[1] := X1;
    Block[0] := X1 or X3 xor X0;
    Block[2] := X3;
  end;
4:
  begin
    X1 := X1 xor X3;
    X3 := not X3;
    X2 := X2 xor X3;
    X3 := X3 xor X0;
    T := X1;
    X1 := X1 and X3 xor X2;
    T := T xor X3;
    X0 := X0 xor T;
    X2 := X2 and T xor X0;
    X0 := X0 and X1;
    X3 := X3 xor X0;
    T := T or X1 xor X0;
    X0 := X0 or X3 xor X2;
    X2 := X2 and X3;
    Block[2] := not X0;
    Block[1] := T xor X2;
    Block[0] := X1;
    Block[3] := X3;
  end;
5:
  begin
    X0 := X0 xor X1;
    X1 := X1 xor X3;
    X3 := not X3;
    T := X1;
    X1 := X1 and X0;
    X2 := X2 xor X3;
    X1 := X1 xor X2;
    X2 := X2 or T;
    T := T xor X3;
    X3 := X3 and X1 xor X0;
    T := T xor X1 xor X2;
    X2 := X2 xor X0;
    X0 := X0 and X3;
    X2 := not X2;
    Block[2] := X0 xor T;
    T := T or X3;
    Block[3] := X2 xor T;
    Block[0] := X1;
    Block[1] := X3;
  end;
```

```
    6:
      begin
        X2 := not X2;
        T := X3;
        X3 := X3 and X0;
        X0 := X0 xor T;
        X3 := X3 xor X2;
        X2 := X2 or T;
        X1 := X1 xor X3;
        X2 := X2 xor X0;
        X0 := X0 or X1;
        X2 := X2 xor X1;
        T := T xor X0;
        X0 := X0 or X3 xor X2;
        T := T xor X3 xor X0;
        X3 := not X3;
        Block[3] := X2 and T xor X3;
        Block[0] := X0;
        Block[1] := X1;
        Block[2] := T;
      end;
    7:
      begin
        T := X1;
        X1 := X1 or X2 xor X3;
        T := T xor X2;
        X2 := X2 xor X1;
        X3 := (X3 or T) and X0;
        T := T xor X2;
        Block[1] := X3 xor X1;
        X1 := X1 or T xor X0;
        X0 := X0 or T xor X2;
        X1 := X1 xor T;
        X2 := X2 xor X1;
        Block[2] := X1 and X0 xor T;
        X2 := not X2 or X0;
        Block[0] := T xor X2;
        Block[3] := X0;
      end;
  end;
end;

procedure InverseSubstitute (var Block: ty128; SBox: integer);
{Applies an inverse S-box to a 128-bit vector. Source: pages 10 to 13 of the
"Speeding up Serpent" paper.}
var
  X0: longint;
  X1: longint;
  X2: longint;
  X3: longint;
  T: longint;  {Temporary register.}
begin
  X0 := Block[0];
  X1 := Block[1];
  X2 := Block[2];
  X3 := Block[3];

  case SBox of
    0:
      begin
        X2 := not X2;
        T := X1;
```

```
    X1 := X1 or X0;
    T := not T;
    X1 := X1 xor X2;
    X2 := X2 or T;
    X1 := X1 xor X3;
    X0 := X0 xor T;
    X2 := X2 xor X0;
    X0 := X0 and X3;
    T := T xor X0;
    X0 := X0 or X1 xor X2;
    X3 := X3 xor T;
    X2 := X2 xor X1;
    X3 := X3 xor X0 xor X1;
    X2 := X2 and X3;
    Block[1] := T xor X2;
    Block[0] := X0;
    Block[2] := X1;
    Block[3] := X3;
  end;
1:
  begin
    T := X1;
    X1 := X1 xor X3;
    X3 := X3 and X1;
    T := T xor X2;
    X3 := X3 xor X0;
    X0 := X0 or X1;
    X2 := X2 xor X3;
    X0 := X0 xor T or X2;
    X1 := X1 xor X3;
    X0 := X0 xor X1;
    X1 := X1 or X3 xor X0;
    T := not T xor X1;
    X1 := X1 or X0 xor X0 or T;
    Block[2] := X3 xor X1;
    Block[0] := T;
    Block[1] := X0;
    Block[3] := X2;
  end;
2:
  begin
    X2 := X2 xor X3;
    X3 := X3 xor X0;
    T := X3;
    X3 := X3 and X2 xor X1;
    X1 := X1 or X2 xor T;
    T := T and X3;
    X2 := X2 xor X3;
    T := T and X0 xor X2;
    X2 := X2 and X1 or X0;
    X3 := not X3;
    Block[2] := X2 xor X3;
    X0 := (X0 xor X3) and X1;
    Block[3] := X3 xor T xor X0;
    Block[0] := X1;
    Block[1] := T;
  end;
3:
  begin
    T := X2;
    X2 := X2 xor X1;
    X0 := X0 xor X2;
```

```
    T := T and X2 xor X0;
    X0 := X0 and X1;
    X1 := X1 xor X3;
    X3 := X3 or T;
    X2 := X2 xor X3;
    X0 := X0 xor X3;
    X1 := X1 xor T;
    X3 := X3 and X2 xor X1;
    X1 := X1 xor X0 or X2;
    X0 := X0 xor X3;
    X1 := X1 xor T;
    Block[3] := X0 xor X1;
    Block[0] := X2;
    Block[1] := X1;
    Block[2] := X3;
  end;
4:
  begin
    T := X2;
    X2 := X2 and X3 xor X1;
    X1 := (X1 or X3) and X0;
    T := T xor X2 xor X1;
    X1 := X1 and X2;
    X0 := not X0;
    X3 := X3 xor T;
    X1 := X1 xor X3;
    X3 := X3 and X0 xor X2;
    X0 := X0 xor X1;
    X2 := X2 and X0;
    X3 := X3 xor X0;
    X2 := X2 xor T or X3;
    Block[1] := X3 xor X0;
    Block[2] := X2 xor X1;
    Block[0] := X0;
    Block[3] := T;
  end;
5:
  begin
    X1 := not X1;
    T := X3;
    X2 := X2 xor X1;
    X3 := X3 or X0 xor X2;
    X2 := (X2 or X1) and X0;
    T := T xor X3;
    X2 := X2 xor T;
    T := T or X0 xor X1;
    X1 := X1 and X2 xor X3;
    T := T xor X2;
    X3 := X3 and T;
    T := T xor X1;
    X3 := X3 xor T;
    Block[1] := not T;
    Block[2] := X3 xor X0;
    Block[0] := X1;
    Block[3] := X2;
  end;
6:
  begin
    X0 := X0 xor X2;
    T := X2;
    X2 := X2 and X0;
    T := T xor X3;
```

```
        X2 := not X2;
        X3 := X3 xor X1;
        X2 := X2 xor X3;
        T := T or X0;
        X0 := X0 xor X2;
        X3 := X3 xor T;
        T := T xor X1;
        X1 := X1 and X3 xor X0;
        X0 := X0 xor X3 or X2;
        Block[3] := X3 xor X1;
        Block[2] := T xor X0;
        Block[0] := X1;
        Block[1] := X2;
      end;
    7:
      begin
        T := X2;
        X2 := X2 xor X0;
        X0 := X0 and X3;
        T := T or X3;
        X2 := not X2;
        X3 := X3 xor X1;
        X1 := X1 or X0;
        X0 := X0 xor X2;
        X2 := X2 and T;
        X3 := X3 and T;
        X1 := X1 xor X2;
        X2 := X2 xor X0;
        X0 := X0 or X2;
        T := T xor X1;
        X0 := X0 xor X3;
        X3 := X3 xor T;
        T := T or X0;
        Block[0] := X3 xor X2;
        Block[3] := T xor X2;
        Block[1] := X0;
        Block[2] := X1;
      end;
  end;
end;
```

In Pascal (and other high-level languages), we are not actually limited to 2-operand expressions, so that aspect of [SuS] is not helpful here (though we did eliminate some assignments to hopefully reduce the number of loads and stores). We have to trust the compiler to generate efficient code.

5. Security

In their original proposal [SC] as well as the specification for Serpent [SP], the authors say that Serpent is secure because it is based on the following principles:

- It uses 32 rounds, which is at least twice as many as needed to defeat all attacks that were known at the time.
- It is based on a substitution-permutation network; those are well-understood and therefore easier to analyze.
- The S-boxes were generated by a deterministic algorithm which optimized them for maximum resistance against linear and differential cryptanalysis.
- There are no weak keys (except too short ones).

5.1. Known attacks

In 2000, Kohno, Kelsey, and Schneier presented the following attacks [KKS2000]:

Rounds	Key size	Data required	Time	Space	Technique
6	256	512 known plaintexts	2^247	2^246	Meet in the middle
6	all	2^83 chosen plaintexts	2^90	2^40	Differential
6	all	2^71 chosen plaintexts	2^103	2^75	Differential
6	192, 256	2^41 chosen plaintexts	2^163	2^45	Differential
7	256	2^122 chosen plaintexts	2^248	2^126	Differential
8	192, 256	2^128 plain/cipher pairs	2^163	2^133	Boomerang
8	192, 256	2^110 chosen plaintexts	2^175	2^115	Amplified boomerang
9	256	2^110 chosen plaintexts	2^252	2^212	Amplified boomerang

At approximately the same time, the same authors presented another amplified boomerang attack (a form of differential cryptanalysis), which was published slightly later [KKS2001]:

Rounds	Key size	Data required	Time	Space	Technique
8	256	2^114 chosen plaintexts	2^179	2^119	Amplified boomerang

In 2001, Biham, Dunkelman, and Keller presented the following attacks [BDK2001]. Note that time is sometimes measured in memory accesses (not encryptions or decryptions) and memory is sometimes measured in bytes instead of blocks (1 block is 2^4 bytes). The rectangle attack is an improvement of the amplified boomerang attack.

Rounds	Key size	Data required	Time	Space	Technique
7	all	2^84 chosen plaintexts	2^85 MA	2^52	Differential
8	256	2^84 chosen plaintexts	2^213 MA	2^84	Differential
10	256	2^126.8 chosen plaintexts	2^207.4	2^131.8 B	Rectangle
10	256	2^126.8 chosen plaintexts	2^205	2^196 B	Rectangle

In 2002, the same authors published the following attacks using linear cryptanalysis [BDK2002]:

Rounds	Key size	Data required	Time	Space	Technique
10	all	2^118 plain/cipher pairs	2^89 MA	2^45	Linear
10	192, 256	2^106 plain/cipher pairs	2^185	2^96	Linear
11	192, 256	2^118 plain/cipher pairs	2^187	2^193	Linear

In 2003, the same authors published the following attacks using differential-linear cryptanalysis [BDK2003]. The paper also contains a summary of all attacks known at that time.

Rounds	Key size	Data required	Time	Space	Technique
10	all	2^105.2 chosen plaintexts	2^123.2	2^40	Differential-linear
11	192, 256	2^125.3 chosen plaintexts	2^172.4	2^30	Differential-linear
11	192, 256	2^125.3 chosen plaintexts	2^139.2	2^60	Differential-linear

In 2009, Singh, Alexander, and Burman noticed that in some of the S-boxes, the nonlinear order of the output bits as a function of the input bits is only 2, not 3 as claimed by the designers [SAB2009].

In 2011, Nguyen, Wu, and Wang presented the following attacks using multidimensional linear cryptanalysis [NWW2011]:

Rounds	Key size	Data required	Time	Space	Technique
11	128	2^116 known plaintexts	2^107.5	2^104	Multidim. linear

```
11      128       2^118 known plaintexts     2^109.5  2^100  Multidim. linear
12      256       2^118 known plaintexts     2^228.8  2^228  Multidim. linear
12      256       2^116 known plaintexts     2^237.5  2^121  Multidim. linear
```

5.2. Timing side channel attacks

While this was probably not known at the time, as the code path that the Serpent encryption and decryption procedures take is always identical and not dependent on the key, and so are the data accesses made to lookup tables (if they are used), Serpent should be immune to timing side channel attacks.

5.3. Power analysis side channel attacks

The paper [PA] describes a power analysis attack on Serpent implemented on an 8-bit smart card. It claims that a 256-bit key can be found in less than 4 ms on average.

5.4. Comparison to AES

There exist attacks on AES which break all rounds in less time (about 2^2) than a brute force search [TW2015]. For Serpent, no attack exists that would break more than 12 of 32 rounds.

AES is also very vulnerable to timing side channel attacks; in very specific scenarios, a full key recovery has been demonstrated with 7 or less blocks of plaintext or ciphertext [AGM2016]. Hardware implementations, of course, do not suffer from this. There are no known timing side channel attacks on Serpent, and, as noted above, they are unlikely to be successful.

6. Speed and space requirements

Of the AES finalists, Serpent was claimed to be the fastest in hardware [SC], however Rijndael was faster in software and chosen for this reason.

Space-wise, the bitslice version of Serpent does not require much ROM. The only lookup tables are the S-boxes, and even if we use the optimized versions from [SuS], the size is not too large.

The way we implemented it temporarily stores the whole 560-byte prekey (132 words + 8 extra words, of 32 bits) and the input key (32 bytes, redundant after the prekey has been generated) in RAM and then precomputes all subkeys, which take up 528 bytes (33 times 32 bytes); a total of 1120 bytes, not counting the actual 16-byte data block and other temporary variables. We did it this way for simplicity. However, even this can be reduced if we only hold in memory the few words needed to generate required parts of the prekey and the subkeys "on the fly".

6.1. Comparison of the presented implementations

We have presented three implementations:

- A basic implementation that needs the initial and final permutations, performs substitutions "horizontally", and performs linear transformations by XOR-ing individual bits. This is version A.
- A bitslice implementation that does not need the initial and final permutations, performs substitutions "vertically" (column-wise), and performs linear transformations by shifting, rotating, and XOR-ing 32 bits at a time. This is version B.

- An optimized variation of version B with S-boxes implemented as specified in [SuS]. This is version C.

To test correctness and measure speed, the following code was used:

```
procedure Test;
{Tests encryption and decryption.}
type
  tyTestVector = record
    Key: ty256;
    Plaintext: ty128;
    Ciphertext: ty128;
  end;
const
  {$IfDef FPC}
    {Test vectors}
    {$I Vectors.inc}
  {$EndIf}
  {Test vector for speed test}
  kInput: ty128 = (0, 0, 0, 0);
  kKey: ty256 = ($80, 0, 0, 0, 0, 0, 0, 0);
  kExpectedOutput: ty128 = (
    longint($12AA23A2),
    longint($0E3C4688),
    longint($BD8EE32B),
    longint($C0165682)
  );
  {Number of iterations for speed test}
  kNumIterations = {$IfDef kDebug} 1 {$Else} 100000 {$EndIf};
var
  {$IfDef FPC}
    Start: TTimeStamp;
  {$EndIf}
  IOBlock: ty128;
  Encrypted: ty128;
  Key: ty256;
  Subkeys: tySubkeys;
  i: integer;
  j: longint;
  Correct: Boolean;

procedure StartTimer;
begin
  {$IfDef FPC}
    Start := DateTimeToTimeStamp(Now);
  {$EndIf}
end;

procedure StopTimer;
begin
  {$IfDef FPC}
    WriteLn(DateTimeToTimeStamp(Now).Time - Start.Time, ' ms');
  {$Else}
    WriteLn;
  {$EndIf}
end;

function BlocksDiffer (var A: ty128; B: pty128): Boolean;
{Checks whether block A matches the expected block B.}
var
  i: integer;
begin
```

```
  BlocksDiffer := false;
  for i := 0 to 3 do
    if A[i] <> B^[i] then begin
      BlocksDiffer := true;
      Exit;
    end;
end;

begin
  {$IfDef FPC}
    {Correctness test}

    Correct := true;
    for i := 0 to 1283 do begin
      Move(kTestVectors[i].Key, Key, 32);
      PrepareSubkeys(Key, Subkeys);

      CopyBlock(@kTestVectors[i].Plaintext, IOBlock);
      Encrypt(IOBlock, Subkeys);
      if BlocksDiffer(IOBlock, @kTestVectors[i].Ciphertext) then begin
        WriteLn('Encryption failed for test vector ', i);
        PrintBlock('Encrypted:', IOBlock);
        PrintBlock('Expected: ', kTestVectors[i].Ciphertext);
        Correct := false;
      end;

      CopyBlock(@kTestVectors[i].Ciphertext, IOBlock);
      Decrypt(IOBlock, Subkeys);
      if BlocksDiffer(IOBlock, @kTestVectors[i].Plaintext) then begin
        WriteLn('Decryption failed for test vector ', i);
        PrintBlock('Decrypted:', IOBlock);
        PrintBlock('Expected: ', kTestVectors[i].Plaintext);
        Correct := false;
      end;
    end;
    if Correct then
      WriteLn('All test vectors passed');
  {$EndIf}

  {Speed test - also a correctness test for Turbo Pascal as the test vector
  file is too large for it}
  WriteLn('Measuring speed:');

  Move(kKey, Key, 32);
  PrepareSubkeys(Key, Subkeys);

  Write('Encrypting (', kNumIterations, ' iterations)... ');
  StartTimer;
  for j := kNumIterations - 1 downto 0 do begin
    CopyBlock(@kInput, IOBlock);
    Encrypt(IOBlock, Subkeys);
  end;
  StopTimer;
  {$IfNDef FPC}
    PrintBlock('Encrypted:', IOBlock);
    PrintBlock('Expected: ', kExpectedOutput);
  {$EndIf}

  Write('Decrypting (', kNumIterations, ' iterations)... ');
  CopyBlock(@IOBlock, Encrypted);
  StartTimer;
  for j := kNumIterations - 1 downto 0 do begin
```

```
    CopyBlock(@Encrypted, IOBlock);
    Decrypt(IOBlock, Subkeys);
  end;
  StopTimer;
  {$IfNDef FPC}
    PrintBlock('Decrypted:', IOBlock);
    PrintBlock('Expected: ', kInput);
  {$EndIf}
end;
```

The test vectors were obtained from [TV]. To be used in the program, they were first permuted and converted to Pascal syntax with the program that is listed in Appendix B.

All code was compiled and tested with Free Pascal 3.0.4 with options `-O3 -Si` on an AMD FX-9590 processor. Each version was run 5 times, encrypting and decrypting a 128-bit block with a 256-bit key 100000 times (1600000 bytes; 1.6 Mb or about 1.5 MB), and the measurements were averaged. The following table shows the average encryption and decryption time (in milliseconds) for each version:

	Encryption	Decryption
A	16215.8	15100.6
B	1835.6	1765.8
C	71.0	71.2

The following table shows the speed in Mb/s:

	Encryption	Decryption
A	0.098669	0.105956
B	0.871650	0.906105
C	22.535210	22.471910

Considering that the fastest version at the time of Serpent's submission to the AES competition achieved over 45 Mb/s on a 200 MHz Pentium, this is not very good. However, we used Pascal and did not make any processor-specific optimizations; with C or assembly, the speed would be much higher.

6.2. TrueCrypt implementation and comparison to AES

The following table shows encryption and decryption speed (in Mb/s) of Serpent as implemented in TrueCrypt, compared to Rijndael (AES), which TrueCrypt can also use. The measurement was done by running TrueCrypt's built-in benchmark on the same processor, with parallelization and AES hardware acceleration disabled.

	Encryption	Decryption
AES	1120.0	1240.0
Serpent	529.6	504.8

TrueCrypt is written in C, implements the S-boxes as in [SuS], and unrolls most loops.

6.3. Hardware implementation and comparison to AES

The paper [HW] compares hardware implementations of Serpent and AES. Serpent was found to be slightly slower (1.96 Gb/s vs. 2.26 Gb/s), however the implementation of AES only supported 128-bit keys and therefore used 10 rounds. If we assume that the required time increases linearly with the number of rounds, a 256-bit implementation of AES with 14 rounds would have a speed of about 1.61 Gb/s, which is slower than Serpent.

7. Conclusion

Serpent does not seem to have many drawbacks, although for an efficient software implementation, one has to resort to novel ways of optimizing the S-boxes, and the result is still barely half as fast as AES. However, it seems to be at least as secure as AES.

References

[SP] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard, 1998. https://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf

[SC] Ross Anderson, Eli Biham, and Lars Knudsen. The case for Serpent, 1998. https://www.cl.cam.ac.uk/~rja14/Papers/serpentcase.pdf

[RA] Ross Anderson. https://www.cl.cam.ac.uk/~rja14/serpent.html

[VC] VeraCrypt: Encryption Algorithms. https://www.veracrypt.fr/en/Encryption%20Algorithms.html

[SuS] Dag Arne Osvik. Speeding up Serpent, 2000. http://www.ii.uib.no/~osvik/pub/aes3.pdf

[TV] Eli Biham. https://www.cs.technion.ac.il/~biham/Reports/Serpent/Serpent-256-128.verified.test-vectors

[KKS2000] Tadayoshi Kohno, John Kelsey, and Bruce Schneier. Preliminary Cryptanalysis of Reduced-Round Serpent, 2000. https://www.schneier.com/academic/archives/2000/04/preliminary_cryptana.html

[KKS2001] Tadayoshi Kohno, John Kelsey, and Bruce Schneier. Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent, 2001. https://www.schneier.com/academic/paperfiles/paper-boomerang.pdf

[BDK2001] Eli Biham, Orr Dunkelman, and Nathan Keller. The Rectangle Attack - Rectangling the Serpent, 2001. https://www.iacr.org/archive/eurocrypt2001/20450338.pdf

[BDK2002] Eli Biham, Orr Dunkelman, and Nathan Keller. Linear Cryptanalysis of Reduced Round Serpent, 2002. https://link.springer.com/content/pdf/10.1007%2F3-540-45473-X_2.pdf

[BDK2003] Eli Biham, Orr Dunkelman, and Nathan Keller. Differential-Linear Cryptanalysis of Serpent, 2003. https://link.springer.com/content/pdf/10.1007%2F978-3-540-39887-5_2.pdf

[SAB2009] Bhupendra Singh, Lexy Alexander, and Sanjay Burman. On Algebraic Relations of Serpent S-Boxes, 2009. https://eprint.iacr.org/2009/038.pdf

[NWW2011] Phuong Ha Nguyen, Hongjun Wu, and Huaxiong Wang. Improving the Algorithm 2 in Multidimensional Linear Cryptanalysis, 2011. http://www3.ntu.edu.sg/home/wuhj/research/publications/2011_ACISP_MLC.pdf

[TW2015] Biaoshuai Tao and Hongjun Wu. Improving the Biclique Cryptanalysis of AES,

2015. https://link.springer.com/chapter/10.1007/978-3-319-19962-7_3

[AGM2016] Ashokkumar C., Ravi Prakash Giri, and Bernard Menezes. Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks, 2016. https://ieeexplore.ieee.org/document/7467359/

[HW] A. K. Lutz, J. Treichler, F. K. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner. 2Gbit/s Hardware Realizations of RIJNDAEL and SERPENT: A Comparative Analysis, 2003. https://link.springer.com/content/pdf/10.1007%2F3-540-36400-5_12.pdf

[PA] Kevin J. Compton, Brian Timm, and Joel VanLaven. A Simple Power Analysis Attack on the Serpent Key Schedule, 2009. https://eprint.iacr.org/2009/473.pdf

A. Utility functions

The Pascal functions presented so far call a few utility functions. Here they are for the sake of completeness. Note that concatenating all the parts of the program in the same order as they were presented is not enough to reconstruct the whole program; Pascal's program structure rules must be respected.

```
function GetBit128 (var X: ty128; Index: integer): longint;
{Gets a bit from a 128-bit vector. Returns 0 or nonzero.}
begin
  GetBit128 := X[Index shr 5] and (longint(1) shl (Index and 31));
end;

procedure PutBit128 (var X: ty128; Index: integer; Value: longint);
{Writes a bit into a 128-bit test vector.}
var
  Mask: longint;
begin
  Mask := longint(1) shl (Index and 31);
  Index := Index shr 5;  {This is now the index of the bit in the longint.}
  if Value = 0 then
    X[Index] := X[Index] and not Mask
  else
    X[Index] := X[Index] or Mask;
end;

procedure PrintHex (Input: longint);
{Prints a 32-bit integer in hexadecimal.}
const
  kNumerals: array [0 .. 15] of char =
    (
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
    );
var
  i: integer;
begin
  i := 32 - 4;
  repeat
    Write(kNumerals[(Input shr i) and 15]);
    i := i - 4;
  until i < 0;
end;

procedure PrintBlock (Description: string; Input: ty128);
{Prints a 128-bit vector in hexadecimal.}
```

```
var
  i: integer;
begin
  Write(Description, ' ');
  for i := 3 downto 0 do begin
    PrintHex(Input[i]);
    if i <> 0 then
      Write('|');
  end;
  WriteLn;
end;

function GetBit32 (X: longint; Index: integer): longint;
  {$IfDef FPC} inline; {$EndIf}
{Gets a bit from a 32-bit integer. Returns 0 or 1.}
begin
  GetBit32 := (X shr Index) and 1;
end;

function Rotate (Input: longint; Amount: byte): longint;
  {$IfDef FPC} inline; {$EndIf}
{Does a left rotation.}
begin
  Rotate := Input shl Amount or Input shr (32 - Amount);
end;

procedure CopyBlock (Src: pty128; var Dst: ty128);
  {$IfDef FPC} inline; {$EndIf}
{Copies a block. The source is specified with a pointer so that constants can
be used.}
begin
  Move(Src^, Dst, 16);
end;

procedure XORBlockWithSubkey (var Block: ty128; var Subkey: ty128);
{XORs a block with a subkey.}
begin
  Block[0] := Block[0] xor Subkey[0];
  Block[1] := Block[1] xor Subkey[1];
  Block[2] := Block[2] xor Subkey[2];
  Block[3] := Block[3] xor Subkey[3];
end;
```

B. Test vector conversion program

The following Python program reads test vectors (from [TV]) and produces a file that can be included in the Pascal program:

```
def Permute(S):
  """
  Takes a string of hexadecimal digits and reverses the order of the bytes in
  it, then reverses the order of the 32-bit words in it, then transforms it
  into Pascal syntax.
  """

  # Reverse the order of the bytes.
  S2 = ""
  i = len(S) - 2
  while i >= 0:
    S2 += S[i : i + 2]
    i -= 2
```

```
  # Reverse the order of the 32-bit words.
  S = ""
  i = len(S2) - 8
  while i >= 0:
    S += S2[i : i + 8]
    i -= 8

  # Convert to Pascal syntax.
  S2 = "("
  for i in range(0, len(S), 8):
    S2 += "longint($" + S[i : i + 8] + ")"
    if i != len(S) - 8:
      S2 += ", "
  S2 += ")"

  return S2

Output = open("Vectors.inc", "w")
Output.write(
  "kTestVectors: array [0 .. 1283] of tyTestVector =\n" + \
  "\t(\n"
)

Counter = 0
InKey = False
for Line in open("Vectors.txt", "r"):
  if "Set" in Line:
    Id = Line.strip()
  elif "key=" in Line:  # First half of the key
    Key = Line[31 :].strip()
    InKey = True
  elif InKey:  # Second half of the key
    Key = Permute(Key + Line[31 :].strip())
    InKey = False
  elif "plain=" in Line:
    Plaintext = Permute(Line[31 :].strip())
  elif "cipher=" in Line:
    Ciphertext = Permute(Line[31 :].strip())
  elif "encrypted=" in Line or "decrypted=" in Line:
    Output.write(
      "\t\t{" + str(Counter) + ": " + Id + "}\n" + \
      "\t\t(\n" + \
      "\t\t\tKey: " + Key + ";\n" + \
      "\t\t\tPlaintext: " + Plaintext + ";\n" + \
      "\t\t\tCiphertext: " + Ciphertext + ";\n" + \
      "\t\t)" + ("" if Counter == 1283 else ",") + "\n"
    )
    Counter += 1

Output.write("\t);")
Output.close()
```