Serpent: A Proposal for the Advanced Encryption Standard

Ross Anderson[1] Eli Biham[2] Lars Knudsen[3]

[1] Cambridge University, England; email rja14@cl.cam.ac.uk
[2] Technion, Haifa, Israel; email biham@cs.technion.ac.il
[3] University of Bergen, Norway; email lars.knudsen@ii.uib.no

Abstract. We propose a new block cipher as a candidate for the Advanced Encryption Standard. Its design is highly conservative, yet still allows a very efficient implementation. It uses S-boxes similar to those of DES in a new structure that simultaneously allows a more rapid avalanche, a more efficient bitslice implementation, and an easy analysis that enables us to demonstrate its security against all known types of attack. With a 128-bit block size and a 256-bit key, it is as fast as DES on the market leading Intel Pentium/MMX platforms (and at least as fast on many others); yet we believe it to be more secure than three-key triple-DES.

1 Introduction

For many applications, the Data Encryption Standard algorithm is nearing the end of its useful life. Its 56-bit key is too small, as shown by a recent distributed key search exercise [28]. Although triple-DES can solve the key length problem, the DES algorithm was also designed primarily for hardware encryption, yet the great majority of applications that use it today implement it in software, where it is relatively inefficient.

For these reasons, the US National Institute of Standards and Technology has issued a call for a successor algorithm, to be called the *Advanced Encryption Standard* or *AES*. The essential requirement is that AES should be both faster than triple DES and at least as secure: it should have a 128 bit block length and a 256 bit key length (though keys of 128 and 192 bits must also be supported).

In this paper, we present a candidate for AES. Our design philosophy has been highly conservative; we did not feel it appropriate to use novel and untested ideas in a cipher which, if accepted after a short review period, will be used to protect enormous volumes of financial transactions, health records and government information over a period of decades.

We initially decided to use the S-boxes from DES, which have been studied intensely for many years and whose properties are thus well understood, in a new structure optimized for efficient implementation on modern processors while simultaneously allowing us to apply the extensive analysis already done on DES. The resulting design gave an algorithm (to which we will refer as Serpent-0) that was as fast as DES and yet more secure than three-key triple-DES, provided a 192 or 256 bit key was selected. This design was published at the 5th International Workshop on Fast Software Encryption [10] in order to give the maximum possible time for public review. Since then we have sought to strengthen the algorithm and improve its performance. As a result, we have selected new, stronger, S-boxes and changed the key schedule slightly. We can now show that our design (which we will call Serpent-1, or more briefly, Serpent) resists all known attacks, including those based on both differential [12] and linear [27] techniques, with very generous safety margins.

The Serpent ciphers were inspired by recent ideas for bitslice implementation of ciphers [6]. However, unlike (say) the bitslice 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 bitslicing. This allows the usual modes of operations to be used, so there is no need to change the environment to gain the extra speed.

Serpent achieves its high performance by a design that makes very efficient use of parallelism, and this extends beyond the level of the algorithm itself. For example, in many applications, we wish to compute a MAC and perform CBC encryption simultaneously with different keys; and as we will see below, our design enables this to be done very efficiently on a processor with two 32-bit integer ALUs (such as the popular Intel MMX series) and almost as efficiently on a 64-bit processor (such as the DEC Alpha).

2 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 $\hat{K}_0, \ldots, \hat{K}_{32}$. 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. (We do not propose, for example, the use of 40-bit keys, but if they are required in some applications, then our padding method can cope with them.) There are no other restrictions on the keyspace.

The cipher itself consists of:

- an initial permutation IP;

- 32 rounds, each consisting of a key mixing operation, a pass through 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, which is described in the next section, and to improve its computational efficiency. Both these two permutations and the linear transformation are specified in the appendix; their design principles will be made clear in the next section.

We use the following notation. The initial permutation IP is applied to the plaintext P giving $\hat{B}_0$, 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 $\hat{B}_1$, the output of the second round (round 1) is $\hat{B}_2$, the output of round i is $\hat{B}_{i+1}$, 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 $\hat{B}_{32}$. The final permutation FP is now applied to give the ciphertext C.

Each round function R_i $(i \in \{0, \ldots, 31\}$ 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 $\hat{B}_0 \oplus \hat{K}_0$ 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 $\hat{B}_0 \oplus \hat{K}_0$ and returns the next four bits of the intermediate vector, and so on. The intermediate vector is then transformed using the linear transformation, giving $\hat{B}_1$. Similarly, R_1 uses 32 copies of S_1 in parallel on $\hat{B}_1 \oplus \hat{K}_1$ and transforms their output using the linear transformation, giving $\hat{B}_2$.

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: we apply S_7 on $\hat{B}_{31} \oplus \hat{K}_{31}$, and XOR the result with $\hat{K}_{32}$ 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 S-box design is discussed below.

As with DES, the final permutation is the inverse of the initial permutation. Thus the cipher may be formally described by the following equations:

$$\begin{aligned} \hat{B}_0 &:= IP(P) \\ \hat{B}_{i+1} &:= R_i(\hat{B}_i) \\ C &:= FP(\hat{B}_{32}) \end{aligned}$$

where

$$\begin{aligned} R_i(X) &= L(\hat{\mathcal{S}}_i(X \oplus \hat{K}_i)) \qquad i = 0, \ldots, 30 \\ R_i(X) &= \hat{\mathcal{S}}_i(X \oplus \hat{K}_i) \oplus \hat{K}_{32} \quad i = 31 \end{aligned}$$

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

Although each round of the proposed cipher might seem weaker than a round of DES, this is not the case. For example, the probability of the best six-round characteristic of DES is about 2^{-20}, while for Serpent the corresponding figure is less than 2^{-58}. 16-round Serpent would be as secure as triple-DES, and twice as fast as DES. However, AES may persist for 25 years as a standard and a further 25 years in legacy systems, and will have to withstand advances in both engineering and cryptanalysis during that time. We therefore propose 32 rounds to put the algorithm's security beyond question. This gives us a cipher that is about as fast as DES but very more secure than 3DES.

2.1 The S-boxes

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 $1/2 \pm 1/4$, and a linear relation between one single bit in the input and one single bit in the output has a probability in the range $1/2 \pm 1/8$;
- 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 rth 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[$\cdot$] be an array containing the least significant four bits of each of the 16 ASCII characters in the expression "`sboxesforserpent`". Let sbox[$\cdot$][$\cdot$] be a (32×16)-array containing the 32 rows of the 8 DES S-boxes, where sbox[r][$\cdot$] denotes the rth row. The function swapentries$(\cdot,\cdot)$ is self-explanatory. The following pseudo-code generates the Serpent S-boxes.

```
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
```

In Serpent-0, we used the DES S-boxes in order to inspire a high level of public confidence that we had not inserted any trapdoor in them. A similar assurance for Serpent-1 comes from the fact that the S-boxes have been generated in this simple deterministic manner.

2.2 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.

3 An Efficient Implementation

Much of the motivation for the above design will become clear as we consider how to implement the algorithm efficiently. We do this in bitslice mode. For a full description of a bitslice implementation of DES, see [6]; the basic idea is that just as one can use a 1-bit processor to implement an algorithm such as DES by executing a hardware description of it, using a logical instruction to emulate each gate, so one can also use a 32-bit processor to compute 32 different DES blocks in parallel — in effect, using the CPU as a 32-way SIMD machine.

This is much more efficient than the conventional implementation, in which a 32-bit processor is mostly idle as it computes operations on 6 bits, 4 bits, or even single bits. The bitslice approach was used in the recent successful DES key search [28], in which spare CPU cycles from thousands of machines were volunteered to solve a challenge cryptogram. However the problem with using bitslice techniques for DES encryption (as opposed to keysearch) is that one has to process many blocks in parallel, and although special modes of operation can be designed for this, they are not the modes in common use.

Our cipher has therefore been designed so that all operations can be executed using 32-fold parallelism during the encryption or decryption of a single block. Indeed the bitslice description of the algorithm is much simpler than its conventional description. No initial and final permutations are required, since the initial and final permutations described in the standard implementation above are just those needed to convert the data from and to the bitslice representation. We will now present an equivalent description of the algorithm for bitslice implementation.

The cipher consists simply of 32 rounds. The plaintext becomes the first intermediate data $B_0 = P$, after which the 32 rounds are applied, where each round $i \in \{0, \ldots, 31\}$ consists of three operations:

1. Key Mixing: At each round, a 128-bit subkey K_i is exclusive or'ed 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 (as it would be in hardware) 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 $\mathcal{S}_i(B_i \oplus K_i)$

3. Linear Transformation: The 32 bits in each of the output words are linearly mixed, by

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

where $<<<$ denotes rotation, and $<<$ denotes shift. In the last round, this linear transformation is replaced by an additional key mixing: $B_{32} := S_7(B_{31} \oplus K_{31}) \oplus K_{32}$. Note that at each stage $IP(B_i) = \hat{B}_i$, and $IP(K_i) = \hat{K}_i$.

The first reason for the choice of linear transformation is to maximize the avalanche effect. The S-boxes have the property that a single input bit change will cause two output bits to change; as the difference sets of {0, 1, 3, 5, 7, 13, 22} modulo 32 have no common member (except one), it follows that a single input bit change will cause a maximal number of bit changes after two and more rounds. The effect is that each plaintext bit affects all the data bits after three rounds, as does each round key bit. Even if an opponent chooses some subkeys and works backwards, it is still guaranteed that each key bit affects each data bit over six rounds. (Some historical information on the design of the linear transformation is given in the appendix.)

The second reason is that it is simple, and can be used in a modern processor with a minimum number of pipeline stalls. The third reason is that it was analyzed by programs we developed for investigating block ciphers, and we found bounds on the probabilities of the differential and linear characteristics. These bounds show that this choice suits our needs.

4 The Key Schedule

As with the description of the cipher, we can describe the key schedule in either standard or bitslice mode. We will give the substantive description for the latter case.

Our cipher requires 132 32-bit words of key material. We first pad the user supplied key to 256 bits, if necessary, as described in section 2. We then expand it to 33 128-bit subkeys $K_0, \ldots, K_{32}$, in the following way. We write the key K as eight 32-bit words $w_{-8}, \ldots, w_{-1}$ and expand these to an intermediate key (which we call *prekey*) $w_0, \ldots, w_{131}$ 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 $(\sqrt{5}+1)/2$ or `0x9e3779b9` 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:

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

We then renumber the 32-bit values k_j as 128-bit subkeys K_i (for i $\in \{0, \ldots,$ r$\}$) as follows:

$$K_i := \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\} \tag{1}$$

Where we are implementing the algorithm in the form initially described in section 2 above rather than using bitslice operations, we now apply IP to the round key in order to place the key bits in the correct column, i.e., $\hat{K}_i = IP(K_i)$.

5 Security

As mentioned above, the initial version of Serpent used the DES S-boxes, as their differential and linear properties are well understood. Our estimates indicated that the number of known/chosen plaintexts required for either type of attack would be well over 2^{100}. Having investigated how these S-boxes worked in our structure, we realised that it was simple to find S-boxes that would improve this figure to 2^{256}, and our desire to offer the best candidate algorithm led us to change to the S-boxes presented in the appendix. (A secondary consideration was that by using 8 S-boxes rather than 32, we greatly reduce the gate count of a high performance hardware implementation and significantly reduce the code size of a compact implementation for use in low-cost smartcards.)

By a strength of 2^{256}, we mean that a differential or linear attack against any key would take that many texts, assuming that they were available (though they aren't). This figure comes from computing the relevant probabilities over all the keys. There are of course higher probability differentials for fixed keys; indeed for any fixed key. differentials with probability 2^{-120} can be expected, and these could in theory be found by exhaustive search. Such differentials are expected in all the AES candidates, as only the required block size of 128 bits affects their probability. In addition, if a key of 128 or 192 bits is selected, then the theoretical cost of keysearch will be reduced.

The conclusion of our analysis is that there is no indication of any useful shortcut attack; we believe that such an attack would require a new theoretical breakthrough. The expected strength of Serpent with various key lengths is therefore as summarised in the following table. In any case, it should be noted that regardless of the design of a 128 bit block cipher, it is normally prudent to change keys well before 2^{64} blocks have been encrypted, in order to avoid the collision attack of section 5.2 below (which applies equally to all AES candidates). This would easily prevent all known kinds of key recovery attack other than keysearch.

Block Size	Key Size	Workload	Type of attack	Chosen/Known Texts
128	128	2^{128}	Exhaustive Search	1
128	192	2^{192}	Exhaustive Search	2
128	256	2^{256}	Exhaustive Search	2

In our analysis, we use conservative bounds to enable our claims to resist reasonable improvements in the studied attacks. For example, our differential and linear analysis uses 24-round and 28-round characteristics, shorter by 8 and 4 rounds than the cipher, while the best attack on DES uses characteristics that are shorter by only three rounds. Our estimates of the probabilities of the best characteristics are also very conservative; in practice they should be considerably lower. Therefore, our complexity claims are almost certainly much lower than the real values, and one may expect Serpent to be much more secure than we actually claim.

We now list the possible weaknesses and attacks which we had in mind. We designed Serpent with a view to reducing or avoiding any vulnerabilities that might arise from them.

5.1 Dictionary Attacks

As the block size is 128 bits, a dictionary attack will require 2^{128} different plaintexts to allow the attacker to encrypt or decrypt arbitrary messages under an unknown key. This attack applies to any deterministic block cipher with 128-bit blocks regardless of its design.

5.2 Modes of Operation

After encrypting about 2^{64} plaintext blocks in the CBC or CFB mode, one can expect to find two equal ciphertext blocks. This enables an attacker to compute the exclusive-or of the two corresponding plaintext blocks [23]. With progressively more plaintext blocks, plaintext relationships can be discovered with progressively higher probability. In addition, when the algorithm is used in feedforward mode as a hash function, a collision can be found with an effort somewhat more than 2^{64} [30]. This attack applies to any deterministic block cipher with 128-bit blocks regardless of its design.

5.3 Key-Collision Attacks

For key size k, key collision attacks can be used to forge messages with complexity only $2^{k/2}$ [8]. Thus, the complexity of forging messages under 128-bit keys is only 2^{64}, under 192-bit keys it is 2^{96}, and under 256-bit keys it is 2^{128}. This attack applies to any deterministic block cipher, and depends only on its key size, regardless of its design.

5.4 Differential Cryptanalysis

An important fact about Serpent is that any characteristic must have at least one active S-box in each round. At least two active S-boxes are required on average, due to the property that a difference in only one bit in the input causes a difference of at least two bits in the output of each S-box. Therefore, if only one bit differs in the input of some round, then at least two differ in the output, and these two bits affect two distinct S-boxes in the following round, whose output differences affect at least four S-boxes in the following round.

We searched for the best characteristics of this cipher. For this, we made a worst case assumption that all the entries in the difference distribution tables have probability 1/4, except the few entries with a one-bit input difference and one-bit output difference, which are assumed impossible (probability zero). The following results hold independently of the order of the S-boxes used in the cipher, and independently of the choice of the S-boxes, so long as they satisfy these minimal conditions. We searched for the best characteristics with up to seven rounds, and the ones with the highest probabilities are given in Table 1.

We see that the probability of a 6-round characteristic is bounded by 2^{-58}. Thus, the probability of a 24-round characteristic is bounded by $2^{-4 \cdot 58} = 2^{-232}$. This means that even if an attacker can implement an 8R-attack (which seems unlikely) this will require many more plaintexts than are available.

Notice that if the linear transformation had used only rotates, then every characteristic could have 32 equiprobable rotated variants, with all the data

Rounds	Differential Probability	Linear Probability $(1/2 \pm p)$	p^{-2}
1	2^{-2}	$1/2 \pm 4/16 = 1/2 \pm 2^{-2}$	2^4
2	2^{-6}	$1/2 \pm 2^2(4/16)^3 = 1/2 \pm 2^{-4}$	2^8
3	2^{-14}	$1/2 \pm 2^7(4/16)^8 = 1/2 \pm 2^{-9}$	2^{18}
4	2^{-26}	$1/2 \pm 2^{13}(4/16)^{14} = 1/2 \pm 2^{-15}$	2^{30}
5	2^{-42}	$1/2 \pm 2^{19}(4/16)^{20} = 1/2 \pm 2^{-21}$	2^{42}
6	2^{-58}	$1/2 \pm 2^{26}(4/16)^{27} = 1/2 \pm 2^{-28}$	2^{56}
7	$< 2^{-70}$	$1/2 \pm 2^{32}(4/16)^{33} = 1/2 \pm 2^{-34}$	$> 2^{68}$

Table 1. Bounds on the Probabilities of Differential and Linear Characteristics

words rotated by the same number of bits. This is the reason that we also use shift instructions, which avoid most of these rotated characteristics.

We have bounded the probabilities of characteristics. However, it is both much more important and much more difficult to bound the probabilities of differentials. In order to reduce the probabilities of differentials we have (1) reduced the probabilities of the characteristics, (2) ensured that there are few characteristics with the highest possible probability, and that they cannot be rotated and still remain valid, (3) arranged for characteristics to affect many different bits, so that they cannot easily be unified into differentials.

We conjecture that the probability of the best 28-round differential is not higher than 2^{-120}, and that such a differential would be very hard to find. Note that for any fixed key there expected to be differentials with probability 2^{-120}; such differentials are to be expected in all ciphers with 128-bit block lengths.

5.5 Linear Cryptanalysis

In linear cryptanalysis, it is possible to find one-bit to one-bit relations of the S-boxes. The probability of these relations is bounded by $1/2 \pm 1/8$. Thus, a 28-round linear characteristic with only one active S-box in each round would have probability $1/2 \pm 2^{27}(1/8)^{28} = 1/2 \pm 2^{-57}$, even if the LT is eliminated, and that an attack based on such relations would require about 2^{114} known plaintexts. However, the linear transformation assures that in the round following a round with only one active S-box, at least two are active.

More general attacks can use linear characteristics with more than one active S-box in some of the rounds. In this case the probabilities of the S-boxes are in the range $1/2 \pm 1/4$. As with differential cryptanalysis, we can bound the probability of characteristics. We searched for the best linear characteristic of this cipher under the assumptions that a probability of any entry is not further from 1/2 than 1/4 and that the probability of a characteristic which relates one bit to one bit is not further from 1/2 than 1/8. Note that due to the relation between linear and differential characteristics, the searches are very similar; we actually modified the search program used in the differential case to search for the best linear characteristics with up to seven rounds, and those with the highest probabilities are given in Table 1.

We can see that the probability of a 6-round characteristic is in the range $1/2 \pm 2^{-28}$, from which we can conclude that the probability of a 24-round characteristic is in the range $1/2 \pm 2^{-109}$. The number of plaintexts needed for such an attack is thus at least 2^{218}, which is much higher than the number of available texts.

Based on these figures we believe that the probability of the best 28-round linear differential (or linear hull) is in the range $1/2 \pm 2^{-120}$, so an attack would need at least 2^{240} blocks. This is a very conservative estimate; we believe the real figure is well over 2^{256}. In any case, linear attacks are infeasible.

5.6 Higher Order Differential Cryptanalysis

It is well known that a dth order differential of a function of nonlinear order d is constant, and this can be exploited in higher order differential attacks [7, 22, 26]. The S-boxes all have nonlinear order 3 so one would expect that the nonlinear order of the output bits after r rounds is about 3^r, with the maximum value of 127 reachable after five rounds. Therefore we are convinced that higher order differential attacks are not applicable to Serpent.

5.7 Truncated Differential Cryptanalysis

For some ciphers it is possible and advantageous to predict only the values of parts of the differences after each round. This notion, of truncated differential attacks, was introduced by Knudsen in [22]. However, the method seems best applicable to ciphers where all operations are done on larger blocks of bits. Because of the strong diffusion over many rounds, we believe that truncated differential attacks are not applicable to Serpent.

5.8 Related Keys

As the key schedule uses rotations and S-boxes, and as we XOR the round number into the prekey, it is highly unlikely that keys can be found that allow related key attacks [9, 20, 21]. Moreover, different rounds of Serpent use different S-boxes, so even if related keys were found, related-key attacks would not be applicable.

Serpent has none of the simpler vulnerabilities that can result from exploitable symmetries in the key schedule: there are no weak keys, semi-weak keys, equivalent keys, or complementation properties.

5.9 Other Attacks

Davies' attack [17, 18] and the improved version of [11] are not applicable, since the S-boxes are invertible, and no duplications of data bits are applied.

As far as we know, neither statistical cryptanalysis [31] nor partitioning cryptanalysis [19] provides a less complex attack than differential or linear cryptanalysis.

Non-linear cryptanalysis has so far only managed to improve the linear attack by small factors [29]. Since a linear attack would involve an impossibly large number of texts, there is no reason to suspect that non-linear techniques will give a useful improvement.

5.10 Timing Attacks

The number of instructions used to encrypt or decrypt does not depend on either the data or the key, and even cache accesses cannot help the attacker as we do not access different locations in memory for different plaintexts or keys. It follows that timing attacks [24] are not applicable.

5.11 Analysis based on Electromagnetic Leakage

Some encryption devices have been found to leak key material electromagnetically, such as through variations in their power consumption [25, 32]. Defending against such attack is primarily the concern of implementers and evaluators; but since most operations in Serpent use all bits in their operands actively, we believe that such attacks will be harder than they would be against comparable devices using many other block ciphers. Much the same should hold for attackers who attempt to exploit compromising electromagnetic radiation.

5.12 Fault Analysis

We have not been concerned to build in any particular protection against attacks based on induced faults [5, 13, 14]. If an attacker can progressively remove the machine instructions by which this cipher is implemented, or progressively destroy selected gates, or progressively modify the bits of the key register, then he can clearly extract the key. An attacker with the ability to modify the implementation detail may have many other options based not just on compromising keys but on subverting protocols, extracting plaintext directly and so on [4]. The mechanisms required to protect against such attacks depend on the device and on the protocols it uses, rather than on the design of any block cipher used [1]. They are thus beyond the scope of this work.

6 Performance in Various Environments

We first implemented this cipher on a 133MHz Pentium/MMX processor. Our 32-round bitslice implementation gave speeds as fast as DES: it encrypted 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 on the same machine and with the same measuring program. With the most obvious optimisation — keeping the subkeys in a fixed array in memory rather than passing them as parameters — the speed increases to 10,281,124 bits per second, or about 1656 clock cycles per 128-bit block. (This version has not been included as it is not consistent with the specified API.) We estimate that on the NIST platform of a 200 MHz Pentium, it will take about the same number of clock cycles (the exact figure will depend on the chip version) and will run at about 14.7 Mbit/s (although this will depend on the test software).

The performance of the cipher on other 32-bit processors in bitslice mode should be similar to the standard implementation of DES, and when coded in assembly language Serpent can be faster than DES. It takes about 1830–1940 instructions (depending on the processor) to encrypt 128 bits versus typically 685 instructions to encrypt 64 bits in DES. The reason our cipher is not 50% slower is that it has been designed to make good use of pipelining, and it does not need so many memory accesses for table lookups.

The instruction count is based on the observation that a gate circuit of the 4x4 (DES-based) S-boxes requires an average of 15.75 gates on the Pentium, about 14.5 on MMX (using only MMX instructions), and even less on the DEC Alpha (the numbers vary due to the different sets of instructions, which are detailed in the appendix). MMX has the additional advantage that it can operate on 64-bit words, or alternatively on two 32-bit words at once (so two encryptions can be done in parallel using the same or different keys, thus enabling simultaneous CBC encryption and MAC computation). It is also implemented with greater parallelism on some recent chips, such as the Pentium II. On the other hand, it does not have rotate operations, so rotates require four instructions (copy, shift left, shift right, and OR).

Interpreted languages will of course give lower performance. Our bitslice Java implementation, for example, performs 10,000 encryptions in 3.3 seconds on a 133 MHz Pentium MMX. This translates to 388 kbit/s, and we expect 583 kbit/s on the NIST 200 MHz machine. Just-in-time compilation should improve these figures dramatically.

It is also worth noting that in many applications we wish to simultaneously encrypt a string and compute a MAC on it, using two different keys. With the market leading Intel/MMX architecture this can be accomplished without difficulty; as noted above, the MMX processor can perform SIMD processing of two 32-bit computations. With a 64-bit processor such as DEC Alpha, it is only slightly more complicated; the S-box computations for both 32-bit computations can be done in parallel and we simply have to implement two different instances of the linear transformation.

When hashing data, the ability to perform a number of encryptions simultaneously may be an advantage, as several streams of data can be hashed separately and then combined at the end of the computation. Even in the absence of such techniques, Serpent can be used in feedforward mode as a hash function, and as key setup takes about one encryption, and 256 bits of message can be hashed at once, the hashing throughput should be roughly equivalent in speed to the encryption or decryption throughput.

For very high speed implementation, one would use dedicated hardware which might have separate logic to pipeline key changes. We estimate that a fully pipelined hardware implementation would have somewhat under 100,000 gates.

This is made up of about 33,000 gates for each of encryption, decryption and key scheduling, plus control logic and buffers. However we know of no applications that would require separate hardware for key scheduling.

Unless hardware keysearch chips are required to recover 40-bit (or other nonstandard length) keys, the application likely to require the greatest key agility would be ATM/B-ISDN. Here, if successive cells are encrypted with different keys, this could mean a key change after every 3 blocks encrypted. However, Serpent's key change overhead is much the same as a single block encryption, and so separate S-boxes for key scheduling are unlikely to be an economic use of silicon. Thus a high performance implementation might take 67,000 gates. However, as Serpent consists of four repetitions of the same structure of 8 S-boxes, it would in most high-speed applications be adequate to pipeline only eight rounds at a time, leading to a gate count of approximately 18,000.

It is also worth remarking that if chip makers wish to support high speed implementations, then it may not be necessary to add a hardware encryption circuit to the CPU. It would be sufficient to add what we call the 'BITSLICE instruction': this works as a generalised multiplexer in that it executes an arbitrary boolean function on four registers under the control of a truth table encoded in a (64-bit) fifth register. We estimate that the cost of implementing this on an n-bit processor will be only about $100n$ gates, and it would have many uses other than cryptography (image processing is a particular candidate). If supported, one BITSLICE instruction would replace many of the instructions in each round; Serpent would become much faster, and require much smaller code.

A compact hardware implementation of the cipher would iteratively apply one round at a time, and as the S-boxes in each round are different, one could use a trick similar to the BITSLICE instruction: get a description of the S-boxes as a parameter in some register, and compute the S-boxes according to this description. This trick also reduces the number of gates required for the hardware implementation of the cipher, to an estimated 4,500.

Another kind of hybrid software-hardware implementation is possible in applications such as satellite TV and indeed analog pay-TV scrambling in general. Here, a keystream generator provides pseudorandom values which specify cut points in luminance and colour signals, which are then cut and rotated using a dedicated hardware descrambler which contains A/D, FIFO and D/A logic. The keystream can be provided by Serpent externally, or by means of a small (4,500 gate) hardware circuit, or by a RISC core such as the ARM embedded in the descrambler. The advantage of having intelligence there is that authentication protocols can be executed with the customer smartcard, which prevents key material being available in the clear to attackers — a known vulnerability of current systems.

With the move to HDTV and to digital video broadcasting in general, decryption functions can be combined with decompression and copyright management functions on a single chip or in a tamper-resistant PCMCIA assembly for upgrading already fielded equipment. Here, if the decoding is performed in an ASIC, a 4,500-gate implementation of Serpent would take up negligible space;

if the decoding is done using an embedded 32-bit RISC or DSP chip (such as an ARM) then Serpent can provide more than adequate performance. The same holds in applications such as secure telephony: here the majority of the processing effort relates to speech compression, and the 32-bit processors that are appropriate for this task can encrypt and decrypt using Serpent at the relevant bit rates (2.4 through 64 kbit/sec) using less than 1% of their clock cycles. Such low CPU utilisation is also a goal of videoconferencing equipment makers, and here too we expect that Serpent will use only a tiny fraction of the relevant CPU (e.g., less than 1% of a Philips' 100MHz TriMedia to encrypt or decrypt at 256 kbit/sec).

On processors with a smaller word size, the instruction count is larger but in many applications one optimises for code size rather than speed. The block cipher applications that use 8-bit processors, such as smartcards, toll tags and prepayment utility meters, typically encrypt or decrypt only one or two blocks during a transaction lasting a second or more; but the cost of the processors, and thus memory size, is a critical factor [3]. The most compact implementation of Serpent appears to be a bitslice implementation (thus avoiding the initial and final permutations) but using table lookups for each S-box (thus avoiding the code size of the Boolean expression of the S-box).

An Ada implementation that uses this strategy indicates a code size of just under 1K and a computational cost of 34,000 clock cycles. Thus on a 3.5 MHz 6805, we expect a throughput of about 100 encryptions per second or 12.8 kbit/s. A full bitslice implementation would occupy more memory (2K) but should take 11,000 clock cycles and thus deliver 40.7 kbit/s. These figures are more than adequate for the applications in question.

However, we expect that the great majority of implementations will use software on general purpose computers such as Pentium MMX and its compatible successors. We therefore designed Serpent to operate as efficiently as possible in this environment subject to the constraint of a highly conservative design in which no radical and untested cryptologic mechanisms are used.

7 Conclusion

We have presented a cipher which we have engineered to satisfy the AES requirements. It is as fast as DES, and we believe it to be more secure than three-key triple DES. We 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.

A patent application has been filed, but if this cipher is adopted as the Advanced Encryption Standard we shall grant a worldwide royalty-free license for conforming implementations.

Finally, information on Serpent, including not just this submission but also the initial version that appeared at 'Fast Software Encryption' and implementations in other languages such as Ada, are linked to the authors' home pages:

```
http://www.cl.cam.ac.uk/~rja14/
http://www.cs.technion.ac.il/~biham/
http://www.ii.uib.no/~larsr/
```

Acknowledgments

We are extremely grateful to Frank Stajano for coding reference implementations in C and Python which enabled us to debug our initial version of Serpent, for running many of the tests independently, and for proofreading this submission. We are also grateful to Intel Corporation for supporting the second author during a visit to Cambridge in September 1997 while much of this work was done; to Markus Kuhn, for the Ada implementation and further proofreading; to the Cryptix group [16] for the initial Java implementation; to Craig Clapp for a discussion on the costs of implementing the key schedule in hardware; and to Gideon Yuval for suggesting the name of the cipher (see Amos 5.19).

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 260–271
7. E Biham, "Higher Order Differential Cryptanalysis", unpublished paper, 1994
8. E Biham, *How to Forge DES-Encrypted Messages in* 2^{28} *Steps*, Technical Report CS884, Technion, August 1996
9. E Biham, "New Types of Cryptanalytic Attacks Using Related Keys", in *Journal of Cryptology* v 7 (1994) no 4 pp 229–246

10. E Biham, RJ Anderson, LR Knudsen, "Serpent: A New Block Cipher Proposal", in *Fast Software Encryption — FSE 98*, Springer LNCS vol 1372 pp 222–238
11. E Biham, A Biryukov, "An Improvement of Davies' Attack on DES", in *Journal of Cryptology* v 10 no 3 (Summer 97) pp 195–205
12. E Biham, A Shamir, *'Differential Cryptanalysis of the Data Encryption Standard'* (Springer 1993)
13. E Biham, A Shamir, "Differential Fault Analysis of Secret Key Cryptosystems", in *Advances in Cryptology — Crypto 97*, Springer LNCS v 1294 pp 513–525
14. D Boneh, RA DeMillo, RJ Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", in *Advances in Cryptology — Eurocrypt 97*, Springer LNCS v 1233 pp 37–51
15. CSK Clapp, "Joint Hardware / Software Design of a Fast Stream Cipher", in *Fast Software Encryption — FSE 98*, Springer LNCS vol 1372 pp 75–92
16. "Cryptix AES Kit", `http://www.t-and-g.fl.net.au/java/cryptix/aes/`
17. DW Davies, *'Investigation of a Potential Weakness in the DES Algorithm'*, private communication (1987)
18. D Davies, S Murphy, "Pairs and Triplets of DES S Boxes", in *Journal of Cryptology* v 8 no 1 (1995) pp 1–25
19. C Harpes, JL Massey, "Partitioning Cryptanalysis", in *Fast Software Encryption — 4th International Workshop, FSE '97*, Springer LNCS v 1267 pp 13–27
20. J Kelsey, B Schneier, D Wagner, "Key-Schedule Cryptanalysis of IDEA, GDES, GOST, SAFER and Triple-DES", in *Advances in Cryptology — Crypto 96*, Springer LNCS v 1109 pp 237–251
21. LR Knudsen, "Cryptanalysis of LOKI91", in *Advances in Cryptology — Auscrypt '92* Springer LNCS v 718 pp 196–208
22. LR Knudsen, "Truncated and Higher-Order Differentials", in *Fast Software Encryption — 2nd International Workshop, FSE '94*, Springer LNCS v 1008 pp 196–211
23. L.R. Knudsen, *Block Ciphers – Analysis, Design and Applications*, Ph.D. Thesis, Århus University, Denmark, 1994.
24. PC Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", in *Advances in Cryptology — Crypto 96*, Springer LNCS v 1109 pp 104–113
25. PC Kocher, "Differential Power Analysis", available online at `http://www.cryptography.com/dpa/`
26. XJ Lai, *'Higher Order Derivative and Differential Cryptanalysis'*, private communication, September 30, 1993.
27. M Matsui, "Linear Cryptanalysis Method for DES Cipher", in *Advances in Cryptology — Eurocrypt 93*, Springer LNCS v 765 pp 386–397
28. RSA Data Security Inc., `www.rsa.com`
29. T Shimoyama, "Quadratic relation of S-box and Application to the Linear Cryptanalysis of DES", presented at the rump session of Fast Software Encryption 98.
30. PC van Oorschot, MJ Wiener, "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms", in *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (ACM, Nov 94) pp 210–218
31. S Vaudenay, "An Experiment on DES Statistical Cryptanalysis", in *3rd ACM Conference on Computer and Communications Security, March 14-16, 96, New Delhi, India; proceedings published by ACM* pp 139–147

32. P Wright, *'Spycatcher — The Candid Autobiography of a Senior Intelligence Officer'*, William Heinemann Australia, 1987, ISBN 0-85561-098-0

A Appendix

A.1 The Initial Permutation *IP*:

Here, having the value v (say, 32) at position p (say, 1) means that the output bit at position p (1) comes from the input bit at position v (32).

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

A.2 The Final Permutation *FP*:

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

A.3 The Linear Transformation:

For each output bit of this transformation, we describe the list of input bits whose parity becomes the output bit. The bits are listed from 0 to 127. (Thus, for example, output bit 1 is the exclusive-or of input bits 72, 114 and 125.) In each row we describe four output bits which together make up the input to a single S-box in the next round. It should be noted that this table is IP(LT(FP(x))) where LT is the linear transformation as specified in section 3.

```
{16 52 56  70  83  94 105} {72 114 125} { 2  9 15  30  76  84 126} {36  90 103}
{20 56 60  74  87  98 109} { 1  76 118} { 2  6 13  19  34  80  88} {40  94 107}
{24 60 64  78  91 102 113} { 5  80 122} { 6 10 17  23  38  84  92} {44  98 111}
{28 64 68  82  95 106 117} { 9  84 126} {10 14 21  27  42  88  96} {48 102 115}
{32 68 72  86  99 110 121} { 2  13  88} {14 18 25  31  46  92 100} {52 106 119}
{36 72 76  90 103 114 125} { 6  17  92} {18 22 29  35  50  96 104} {56 110 123}
{ 1 40 76  80  94 107 118} {10  21  96} {22 26 33  39  54 100 108} {60 114 127}
{ 5 44 80  84  98 111 122} {14  25 100} {26 30 37  43  58 104 112} { 3 118    }
{ 9 48 84  88 102 115 126} {18  29 104} {30 34 41  47  62 108 116} { 7 122    }
{ 2 13 52  88  92 106 119} {22  33 108} {34 38 45  51  66 112 120} {11 126    }
```

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

A.4 The Inverse Linear Transformation:

Here we provide the inverse of the above linear transformation, which is used when implementing a non-bitslice version of decryption.

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

```
{ 13  15  32} {50 61  89  93 108} {62 73 103} {50 61 77  79  91  96 119}
{ 17  19  36} {54 65  93  97 112} {66 77 107} {54 65 81  83  95 100 123}
{ 21  23  40} {58 69  97 101 116} {70 81 111} {58 69 85  87  99 104 127}
{ 25  27  44} {62 73 101 105 120} {74 85 115} { 3 62 73  89  91 103 108}
{ 29  31  48} {66 77 105 109 124} {78 89 119} { 7 66 77  93  95 107 112}
{ 33  35  52} { 0 70  81 109 113} {82 93 123} {11 70 81  97  99 111 116}
{ 37  39  56} { 4 74  85 113 117} {86 97 127} {15 74 85 101 103 115 120}
{ 41  43  60} { 8 78  89 117 121} {     3  90} {19 78 89 105 107 119 124}
{ 45  47  64} {12 82  93 121 125} {     7  94} { 0 23 82  93 109 111 123}
{ 49  51  68} { 1 16  86  97 125} {    11  98} { 4 27 86  97 113 115 127}
```

A.5 S-Boxes

Here are the S-boxes S_0 through S_7:

```
S0:   3  8 15  1 10  6  5 11 14 13  4  2  7  0  9 12
S1:  15 12  2  7  9  0  5 10  1 11 14  8  6 13  3  4
S2:   8  6  7  9  3 12 10 15 13  1 14  4  0 11  5  2
S3:   0 15 11  8 12  9  6  3 13  1  2  4 10  7  5 14
S4:   1 15  8  3 12  0 11  6  2  5  4 10  9 14  7 13
S5:  15  5  2 11  4 10  9 12  0  3 14  8 13  6  7  1
S6:   7  2 12  5  8  4  6 11 14  9  1 15 13  3 10  0
S7:   1 13 15  0 14  8  2 11  7  4 12 10  9  3  5  6
```

Here are the inverse S-boxes for use in decryption:

```
InvS0:  13  3 11  0 10  6  5 12  1 14  4  7 15  9  8  2
InvS1:   5  8  2 14 15  6 12  3 11  4  7  9  1 13 10  0
InvS2:  12  9 15  4 11 14  1  2  0  3  6 13  5  8 10  7
InvS3:   0  9 10  7 11 14  6 13  3  5 12  2  4  8 15  1
InvS4:   5  0  8  3 10  9  7 14  2 12 11  6  4 15 13  1
InvS5:   8 15  2  9  4  1 13 14 11  6  5  3  7 12 10  0
InvS6:  15 10  1 13  5  3  6  0  4  9 14  7  2 12  8 11
InvS7:   3  0  6 13  9 14 15  8  5 12 11  7 10  1  4  2
```

A.6 Lists of Relevant Instructions on Various Processors

The relevant instructions on the following processors are:

Pentium:	AND, OR, XOR, NOT, rotate
MMX:	AND, OR, XOR, NOT, ANDN, only shifts
Alpha:	AND, OR, XOR, NOT, ANDN, ORN, XORN, only shifts

where the ANDN operation on x and y is $x \wedge (\neg y)$, the ORN operation is $x \vee (\neg y)$, and the XORN operation is $x \oplus (\neg y)$ (or equivalently $\neg(x \oplus y)$).

On MMX a rotate takes four instructions, while on an Alpha it takes three. On Pentium and MMX it might be necessary to copy some of the registers before use, as instructions have only two arguments; but some instructions can refer directly to memory. The Alpha instructions have 3 arguments (src1, src2 and destination), but cannot refer directly to memory.

A.7 Historical Remarks

Here we describe some design history. In our 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.

As noted above, the first published version of Serpent reused the S-boxes from DES. After this was presented at Fast Software Encryption 98 [10], 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 on the last few rounds of the cipher', as we put it in [10]. 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.

Finally, we considered making available cognate algorithms with the same structure as Serpent but with block sizes of (say) 64, 256 and 512 bits. In the end we decided not include these in our submission for a number of reasons, of which by far the most important was that we used the available time to concentrate on the main algorithm. We did not have the resources to test variants with other block lengths with the thoroughness that would have been appropriate. We do not however foresee any great difficulty in developing such variants should they be required at a later date.