Twofish: A 128-Bit Block Cipher

Bruce Schneier* John Kelsey† Doug Whiting‡ David Wagner§ Chris Hall¶
Niels Ferguson ||

15 June 1998

Abstract

Twofish is a 128-bit block cipher that accepts a variable-length key up to 256 bits. The cipher is a 16-round Feistel network with a bijective F function made up of four key-dependent 8-by-8-bit S-boxes, a fixed 4-by-4 maximum distance separable matrix over GF(2^8), a pseudo-Hadamard transform, bitwise rotations, and a carefully designed key schedule. A fully optimized implementation of Twofish encrypts on a Pentium Pro at 17.8 clock cycles per byte, and an 8-bit smart card implementation encrypts at 1660 clock cycles per byte. Twofish can be implemented in hardware in 14000 gates. The design of both the round function and the key schedule permits a wide variety of tradeoffs between speed, software size, key setup time, gate count, and memory. We have extensively cryptanalyzed Twofish; our best attack breaks 5 rounds with $2^{22.5}$ chosen plaintexts and 2^{51} effort.

Keywords: Twofish, cryptography, cryptanalysis, block cipher, AES.

Current web site: `http://www.counterpane.com/twofish.html`

*Counterpane Systems, 101 E Minnehaha Parkway, Minneapolis, MN 55419, USA; `schneier@counterpane.com`.
†Counterpane Systems; `kelsey@counterpane.com`.
‡Hi/fn, Inc., 5973 Avenida Encinas, Suite 110, Carlsbad, CA 92008, USA; `dwhiting@hifn.com`.
§University of California Berkeley, Soda Hall, Berkeley, CA 94720, USA; `daw@cs.berkeley.edu`.
¶Counterpane Systems; `hall@counterpane.com`.
||Counterpane Systems; `niels@counterpane.com`.

Contents

1 Introduction **3**

2 Twofish Design Goals **3**

3 Twofish Building Blocks **4**
3.1 Feistel Networks 4
3.2 S-boxes 5
3.3 MDS Matrices 5
3.4 Pseudo-Hadamard Transforms 5
3.5 Whitening 5
3.6 Key Schedule 5

4 Twofish **5**
4.1 The Function F 7
4.2 The Function g 7
4.3 The Key Schedule 8
4.4 Round Function Overview 12

5 Performance of Twofish **12**
5.1 Performance on Large Microprocessors 12
5.2 Performance on Smart Cards 15
5.3 Performance on Future Microprocessors 16
5.4 Hardware Performance 17

6 Twofish Design Philosophy **18**
6.1 Performance-Driven Design 18
6.2 Conservative Design 19
6.3 Simple Design 20
6.4 S-boxes 21
6.5 The Key Schedule 22

7 The Design of Twofish **23**
7.1 The Round Structure 23
7.2 The Key-dependent S-boxes 24
7.3 MDS Matrix 27
7.4 PHT 29
7.5 Key Addition 29
7.6 Feistel Combining Operation 29
7.7 Use of Different Groups 29
7.8 Diffusion in the Round Function . . . 29
7.9 One-bit Rotation 30
7.10 The Number of Rounds 31
7.11 The Key Schedule 31
7.12 Reed-Solomon Code 36

8 Cryptanalysis of Twofish **36**
8.1 Differential Cryptanalysis 36
8.2 Extensions to Differential Cryptanalysis 41
8.3 Search for the Best Differential Characteristic 41
8.4 Linear Cryptanalysis 44
8.5 Interpolation Attack 45
8.6 Partial Key Guessing Attacks 46
8.7 Related-key Cryptanalysis 46
8.8 A Related-Key Attack on a Twofish Variant 48
8.9 Side-Channel Cryptanalysis and Fault Analysis 50
8.10 Attacking Simplified Twofish 50

9 Trap Doors in Twofish **52**

10 When is a Cipher Insecure? **53**

11 Using Twofish **53**
11.1 Chaining Modes 53
11.2 One-Way Hash Functions 53
11.3 Message Authentication Codes 53
11.4 Pseudo-Random Number Generators . 53
11.5 Larger Keys 54
11.6 Additional Block Sizes 54
11.7 More or Fewer Rounds 54
11.8 Family Key Variant: Twofish-FK . . . 54

12 Historical Remarks **56**

13 Conclusions and Further Work **57**

14 Acknowledgments **58**

A Twofish Test Vectors **65**
A.1 Intermediate Values 65
A.2 Full Encryptions 67

1 Introduction

In 1972 and 1974, the National Bureau of Standards (now the National Institute of Standards and Technology, or NIST) issued the first public request for an encryption standard. The result was DES [NBS77], arguably the most widely used and successful encryption algorithm in the world.

Despite its popularity, DES has been plagued with controversy. Some cryptographers objected to the "closed-door" design process of the algorithm. The debate about whether DES' key is too short for acceptable commercial security has raged for many years [DH79], but recent advances in distributed key search techniques have left no doubt in anyone's mind that its key is simply too short for today's security applications [Wie94, BDR+96]. Triple-DES has emerged as an interim solution in many high-security applications, such as banking, but it is too slow for some uses. More fundamentally, the 64-bit block length shared by DES and most other well-known ciphers opens it up to attacks when large amounts of data are encrypted under the same key.

In response to a growing desire to replace DES, NIST announced the Advanced Encryption Standard (AES) program in 1997 [NIST97a]. NIST solicited comments from the public on the proposed standard, and eventually issued a call for algorithms to satisfy the standard [NIST97b]. The intention is for NIST to make all submissions public and eventually, through a process of public review and comment, choose a new encryption standard to replace DES.

NIST's call requested a block cipher. Block ciphers can be used to design stream ciphers with a variety of synchronization and error extension properties, one-way hash functions, message authentication codes, and pseudo-random number generators. Because of this flexibility, they are the workhorse of modern cryptography.

NIST specified several other design criteria: a longer key length, larger block size, faster speed, and greater flexibility. While no single algorithm can be optimized for all needs, NIST intends AES to become the standard symmetric algorithm of the next decade.

Twofish is our submission to the AES selection process. It meets all the required NIST criteria—128-bit block; 128-, 192-, and 256-bit key; efficient on various platforms; etc.—and some strenuous design requirements, performance as well as cryptographic, of our own.

Twofish can:

- Encrypt data at 285 clock cycles per block on a Pentium Pro, after a 12700 clock-cycle key setup.
- Encrypt data at 860 clock cycles per block on a Pentium Pro, after a 1250 clock-cycle key setup.
- Encrypt data at 26500 clock cycles per block on a 6805 smart card, after a 1750 clock-cycle key setup.

This paper is organized as follows: Section 2 discusses our design goals for Twofish. Section 3 describes the building blocks and general design of the cipher. Section 4 defines the cipher. Section 5 discusses the performance of Twofish. Section 6 talks about the design philosophy that we used. In Section 7 we describe the design process, and why the various choices were made. Section 8 contains our best cryptanalysis of Twofish. In Section 9 we discuss the possibility of trapdoors in the cipher. Section 10 compares Twofish with some other ciphers. Section 11 discusses various modes of using Twofish, including a family-key variant. Section 12 contains historical remarks, and Section 13 our conclusions and directions for future analysis.

2 Twofish Design Goals

Twofish was designed to meet NIST's design criteria for AES [NIST97b]. Specifically, they are:

- A 128-bit symmetric block cipher.
- Key lengths of 128 bits, 192 bits, and 256 bits.
- No weak keys.
- Efficiency, both on the Intel Pentium Pro and other software and hardware platforms.
- Flexible design: e.g., accept additional key lengths; be implementable on a wide variety of platforms and applications; and be suitable for a stream cipher, hash function, and MAC.
- Simple design, both to facilitate ease of analysis and ease of implementation.

Additionally, we imposed the following performance criteria on our design:

- Accept any key length up to 256 bits.

- Encrypt data in less than 500 clock cycles per block on an Intel Pentium, Pentium Pro, and Pentium II, for a fully optimized version of the algorithm.
- Be capable of setting up a 128-bit key (for optimal encryption speed) in less than the time required to encrypt 32 blocks on a Pentium, Pentium Pro, and Pentium II.
- Encrypt data in less than 5000 clock cycles per block on a Pentium, Pentium Pro, and Pentium II with no key setup time.
- Not contain any operations that make it inefficient on other 32-bit microprocessors.
- Not contain any operations that make it inefficient on 8-bit and 16-bit microprocessors.
- Not contain any operations that reduce its efficiency on proposed 64-bit microprocessors; e.g., Merced.
- Not include any elements that make it inefficient in hardware.
- Have a variety of performance tradeoffs with respect to the key schedule.
- Encrypt data in less than less than 10 milliseconds on a commodity 8-bit microprocessor.
- Be implementable on a 8-bit microprocessor with only 64 bytes of RAM.
- Be implementable in hardware using less than 20,000 gates.

Our cryptographic goals were as follows:

- 16-round Twofish (without whitening) should have no chosen-plaintext attack requiring fewer than 2^{80} chosen plaintexts and less than 2^N time, where N is the key length.
- 12-round Twofish (without whitening) should have no related-key attack requiring fewer than 2^{64} chosen plaintexts, and less than $2^{N/2}$ time, where N is the key length.

Finally, we imposed the following flexibility goals:

- Have variants with a variable number of rounds.
- Have a key schedule that can be precomputed for maximum speed, or computed on-the-fly for maximum agility and minimum memory requirements. Additionally, it should be suitable for dedicated hardware applications: e.g., no large tables.
- Be suitable as a stream cipher, one-way hash function, MAC, and pseudo-random number generator, using well-understood construction methods.
- Have a family-key variant to allow for different, non-interoperable, versions of the cipher.

We feel we have met all of these goals in the design of Twofish.

3 Twofish Building Blocks

3.1 Feistel Networks

A *Feistel network* is a general method of transforming any function (usually called the F function) into a permutation. It was invented by Horst Feistel [FNS75] in his design of Lucifer [Fei73], and popularized by DES [NBS77]. It is the basis of most block ciphers published since then, including FEAL [SM88], GOST [GOST89], Khufu and Khafre [Mer91], LOKI [BPS90, BKPS93], CAST-128 [Ada97a], Blowfish [Sch94], and RC5 [Riv95].

The fundamental building block of a Feistel network is the F function: a key-dependent mapping of an input string onto an output string. An F function is always non-linear and possibly non-surjective[1]:

$$F : \{0,1\}^{n/2} \times \{0,1\}^N \mapsto \{0,1\}^{n/2}$$

where n is the block size of the Feistel Network, and F is a function taking $n/2$ bits of the block and N bits of a key as input, and producing an output of length $n/2$ bits. In each round, the "source block" is the input to F, and the output of F is XORed with the "target block," after which these two blocks swap places for the next round. The idea here is to take an F function, which may be a weak encryption algorithm when taken by itself, and repeatedly iterate it to create a strong encryption algorithm.

Two rounds of a Feistel network is called a "cycle" [SK96]. In one cycle, every bit of the text block has been modified once.[2]

[1] A non-surjective F function is one in which not all outputs in the output space can occur.

[2] The notion of a cycle allows Feistel networks to be compared with unbalanced Feistel networks [SK96, ZMI90] such as MacGuffin [BS95] (cryptanalyzed in [RP95a]) and Bear/Lion [AB96b], and with SP-networks (also called uniform transformation structures [Fei73]) such as IDEA, SAFER, and Shark [RDP+96] (see also [YTH96]). Thus, 8-cycle (8-round) IDEA is comparable to 8-cycle (16-round) DES and 8-cycle (32-round) Skipjack.

Twofish is a 16-round Feistel network with a bijective F function.

3.2 S-boxes

An S-box is a table-driven non-linear substitution operation used in most block ciphers. S-boxes vary in both input size and output size, and can be created either randomly or algorithmically. S-boxes were first used in Lucifer, then DES, and afterwards in most encryption algorithms.

Twofish uses four different, bijective, key-dependent, 8-by-8-bit S-boxes. These S-boxes are built using two fixed 8-by-8-bit permutations and key material.

3.3 MDS Matrices

A maximum distance separable (MDS) code over a field is a linear mapping from a field elements to b field elements, producing a composite vector of $a+b$ elements, with the property that the minimum number of non-zero elements in any non-zero vector is at least $b+1$ [MS77]. Put another way, the "distance" (i.e., the number of elements that differ) between any two distinct vectors produced by the MDS mapping is at least $b+1$. It can easily be shown that no mapping can have a larger minimum distance between two distinct vectors, hence the term maximum distance separable. MDS mappings can be represented by an MDS matrix consisting of $a \times b$ elements. Reed-Solomon (RS) error-correcting codes are known to be MDS. A necessary and sufficient condition for an $a \times b$ matrix to be MDS is that all possible square submatrices, obtained by discarding rows or columns, are non-singular.

Serge Vaudenay first proposed MDS matrices as a cipher design element [Vau95]. Shark [RDP+96] and Square [DKR97] use MDS matrices (see also [YMT97]), although we first saw the construction used in the unpublished cipher Manta[3] [Fer96]. Twofish uses a single 4-by-4 MDS matrix over GF(2^8).

3.4 Pseudo-Hadamard Transforms

A pseudo-Hadamard transform (PHT) is a simple mixing operation that runs quickly in software. Given two inputs, a and b, the 32-bit PHT is defined as:

$$a' = a + b \bmod 2^{32}$$

$$b' = a + 2b \bmod 2^{32}$$

SAFER [Mas94] uses 8-bit PHTs extensively for diffusion. Twofish uses a 32-bit PHT to mix the outputs from its two parallel 32-bit g functions. This PHT can be executed in two opcodes on most modern microprocessors, including the Pentium family.

3.5 Whitening

Whitening, the technique of XORing key material before the first round and after the last round, was used by Merkle in Khufu/Khafre, and independently invented by Rivest for DES-X [KR96]. In [KR96], it was shown that whitening substantially increases the difficulty of keysearch attacks against the remainder of the cipher. In our attacks on reduced-round Twofish variants, we discovered that whitening substantially increased the difficulty of attacking the cipher, by hiding from an attacker the specific inputs to the first and last rounds' F functions.

Twofish XORs 128 bits of subkey before the first Feistel round, and another 128 bits after the last Feistel round. These subkeys are calculated in the same manner as the round subkeys, but are not used anywhere else in the cipher.

3.6 Key Schedule

The key schedule is the means by which the key bits are turned into round keys that the cipher can use. Twofish needs a lot of key material, and has a complicated key schedule. To facilitate analysis, the key schedule uses the same primitives as the round function.

4 Twofish

Figure 1 shows an overview of the Twofish block cipher. Twofish uses a 16-round Feistel-like structure with additional whitening of the input and output. The only non-Feistel elements are the 1-bit rotates. The rotations can be moved into the F function to create a pure Feistel structure, but this requires an additional rotation of the words just before the output whitening step.

The plaintext is split into four 32-bit words. In the input whitening step, these are XORed with four key words. This is followed by sixteen rounds. In each

[3] Manta is a block cipher with a large block size and an emphasis on long-term security rather than speed. It uses an SP-like network with DES as the S-boxes and MDS matrices for the permutations.

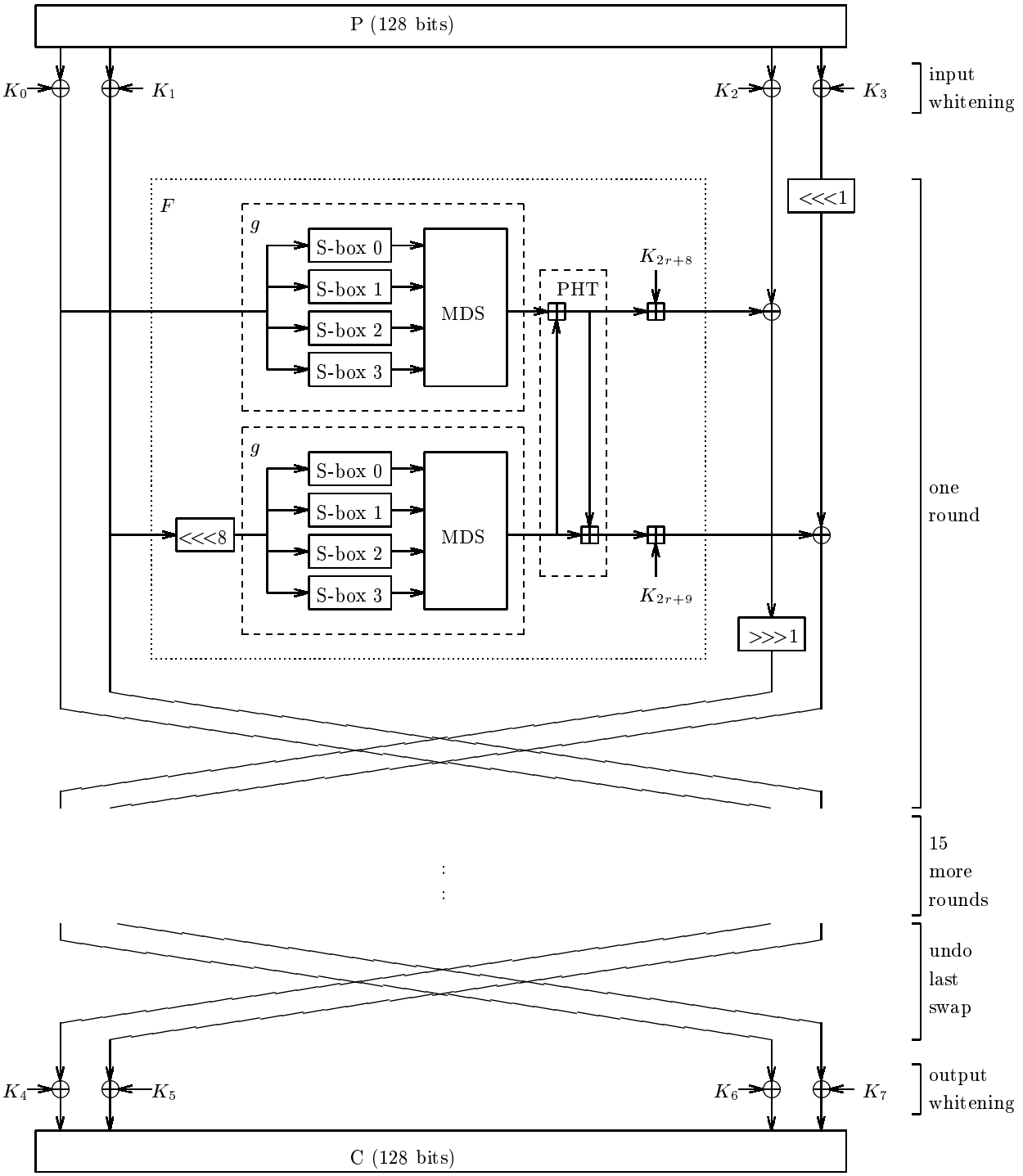

Figure 1: Twofish

round, the two words on the left are used as input to the g functions. (One of them is rotated by 8 bits first.) The g function consists of four byte-wide key-dependent S-boxes, followed by a linear mixing step based on an MDS matrix. The results of the two g functions are combined using a Pseudo-Hadamard Transform (PHT), and two keywords are added. These two results are then XORed into the words on the right (one of which is rotated left by 1 bit first, the other is rotated right afterwards). The left and right halves are then swapped for the next round. After all the rounds, the swap of the last round is reversed, and the four words are XORed with four more key words to produce the ciphertext.

More formally, the 16 bytes of plaintext $p_0, \ldots, p_{15}$ are first split into 4 words $P_0, \ldots, P_3$ of 32 bits each using the little-endian convention.

$$P_i = \sum_{j=0}^{3} p_{(4i+j)} \cdot 2^{8j} \qquad i = 0, \ldots, 3$$

In the input whitening step, these words are XORed with 4 words of the expanded key.

$$R_{0,i} = P_i \oplus K_i \qquad i = 0, \ldots, 3$$

In each of the 16 rounds, the first two words are used as input to the function F, which also takes the round number as input. The third word is XORed with the first output of F and then rotated right by one bit. The fourth word is rotated left by one bit and then XORed with the second output word of F. Finally, the two halves are exchanged. Thus,

$$\begin{aligned}
(F_{r,0}, F_{r,1}) &= F(R_{r,0}, R_{r,1}, r) \\
R_{r+1,0} &= \mathrm{ROR}(R_{r,2} \oplus F_{r,0}, 1) \\
R_{r+1,1} &= \mathrm{ROL}(R_{r,3}, 1) \oplus F_{r,1} \\
R_{r+1,2} &= R_{r,0} \\
R_{r+1,3} &= R_{r,1}
\end{aligned}$$

for $r = 0, \ldots, 15$ and where ROR and ROL are functions that rotate their first argument (a 32-bit word) left or right by the number of bits indicated by their second argument.

The output whitening step undoes the 'swap' of the last round, and XORs the data words with 4 words of the expanded key.

$$C_i = R_{16,(i+2) \bmod 4} \oplus K_{i+4} \qquad i = 0, \ldots, 3$$

The four words of ciphertext are then written as 16 bytes $c_0, \ldots, c_{15}$ using the same little-endian conversion used for the plaintext.

$$c_i = \left\lfloor \frac{C_{\lfloor i/4 \rfloor}}{2^{8(i \bmod 4)}} \right\rfloor \bmod 2^8 \qquad i = 0, \ldots, 15$$

4.1 The Function F

The function F is a key-dependent permutation on 64-bit values. It takes three arguments, two input words R_0 and R_1, and the round number r used to select the appropriate subkeys. R_0 is passed through the g function, which yields T_0. R_1 is rotated left by 8 bits and then passed through the g function to yield T_1. The results T_0 and T_1 are then combined in a PHT and two words of the expanded key are added.

$$\begin{aligned}
T_0 &= g(R_0) \\
T_1 &= g(\mathrm{ROL}(R_1, 8)) \\
F_0 &= (T_0 + T_1 + K_{2r+8}) \bmod 2^{32} \\
F_1 &= (T_0 + 2T_1 + K_{2r+9}) \bmod 2^{32}
\end{aligned}$$

where (F_0, F_1) is the result of F. We also define the function F' for use in our analysis. F' is identical to the F function, except that it does not add any key blocks to the output. (The PHT is still performed.)

4.2 The Function g

The function g forms the heart of Twofish. The input word X is split into four bytes. Each byte is run through its own key-dependent S-box. Each S-box is bijective, takes 8 bits of input, and produces 8 bits of output. The four results are interpreted as a vector of length 4 over GF(2^8), and multiplied by the 4×4 MDS matrix (using the field GF(2^8) for the computations). The resulting vector is interpreted as a 32-bit word which is the result of g.

$$\begin{aligned}
x_i &= \lfloor X/2^{8i} \rfloor \bmod 2^8 \qquad i = 0, \ldots, 3 \\
y_i &= s_i[x_i] \qquad i = 0, \ldots, 3 \\
\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} &= \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \mathrm{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \\
Z &= \sum_{i=0}^{3} z_i \cdot 2^{8i}
\end{aligned}$$

where s_i are the key-dependent S-boxes and Z is the result of g. For this to be well-defined, we need to specify the correspondence between byte values and the field elements of GF(2^8). We represent GF(2^8) as GF(2)[x]/$v(x)$ where $v(x) = x^8 + x^6 + x^5 + x^3 + 1$ is a primitive polynomial of degree 8 over GF(2). The field element $a = \sum_{i=0}^{7} a_i x^i$ with $a_i \in$ GF(2)

is identified with the byte value $\sum_{i=0}^{7} a_i 2^i$. This is in some sense the "natural" mapping; addition in GF(2^8) corresponds to a XOR of the bytes.

The MDS matrix is given by:

$$\text{MDS} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix}$$

where the elements have been written as hexadecimal byte values using the above defined correspondence.

4.3 The Key Schedule

The key schedule has to provide 40 words of expanded key $K_0, \ldots, K_{39}$, and the 4 key-dependent S-boxes used in the g function. Twofish is defined for keys of length $N = 128$, $N = 192$, and $N = 256$. Keys of any length shorter than 256 bits can be used by padding them with zeroes until the next larger defined key length.

We define $k = N/64$. The key M consists of $8k$ bytes $m_0, \ldots, m_{8k-1}$. The bytes are first converted into $2k$ words of 32 bits each

$$M_i = \sum_{j=0}^{3} m_{(4i+j)} \cdot 2^{8j} \qquad i = 0, \ldots, 2k-1$$

and then into two word vectors of length k.

$$\begin{aligned} M_e &= (M_0, M_2, \ldots, M_{2k-2}) \\ M_o &= (M_1, M_3, \ldots, M_{2k-1}) \end{aligned}$$

A third word vector of length k is also derived from the key. This is done by taking the key bytes in groups of 8, interpreting them as a vector over GF(2^8), and multiplying them by a 4×8 matrix derived from an RS code. Each result of 4 bytes is then interpreted as a 32-bit word. These words make up the third vector.

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{RS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

$$S_i = \sum_{j=0}^{3} s_{i,j} \cdot 2^{8j}$$

for $i = 0, \ldots, k-1$, and

$$S = (S_{k-1}, S_{k-2}, \ldots, S_0)$$

Note that S lists the words in "reverse" order. For the RS matrix multiply, GF(2^8) is represented by GF(2)[x]/$w(x)$, where $w(x) = x^8 + x^6 + x^3 + x^2 + 1$ is another primitive polynomial of degree 8 over GF(2). The mapping between byte values and elements of GF(2^8) uses the same definition as used for the MDS matrix multiply. Using this mapping, the RS matrix is given by:

$$\text{RS} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix}$$

The three vectors M_e, M_o, and S form the basis of the key schedule.

4.3.1 Additional Key Lengths

Twofish can accept keys of any byte length up to 256 bits. For key sizes that are not defined above, the key is padded at the end with zero bytes to the next larger length that is defined. For example, an 80-bit key $m_0, \ldots, m_9$ would be extended by setting $m_i = 0$ for $i = 10, \ldots, 15$ and treating it as a 128-bit key.

4.3.2 The Function h

Figure 2 shows an overview of the function h. This is a function that takes two inputs—a 32-bit word X and a list $L = (L_0, \ldots, L_{k-1})$ of 32-bit words of length k—and produces one word of output. This function works in k stages. In each stage, the four bytes are each passed through a fixed S-box, and XORed with a byte derived from the list. Finally, the bytes are once again passed through a fixed S-box, and the four bytes are multiplied by the MDS matrix just as in g. More formally: we split the words into bytes.

$$\begin{aligned} l_{i,j} &= \lfloor L_i / 2^{8j} \rfloor \bmod 2^8 \\ x_j &= \lfloor X / 2^{8j} \rfloor \bmod 2^8 \end{aligned}$$

for $i = 0, \ldots, k-1$ and $j = 0, \ldots, 3$. Then the sequence of substitutions and XORs is applied.

$$y_{k,j} = x_j \qquad j = 0, \ldots, 3$$

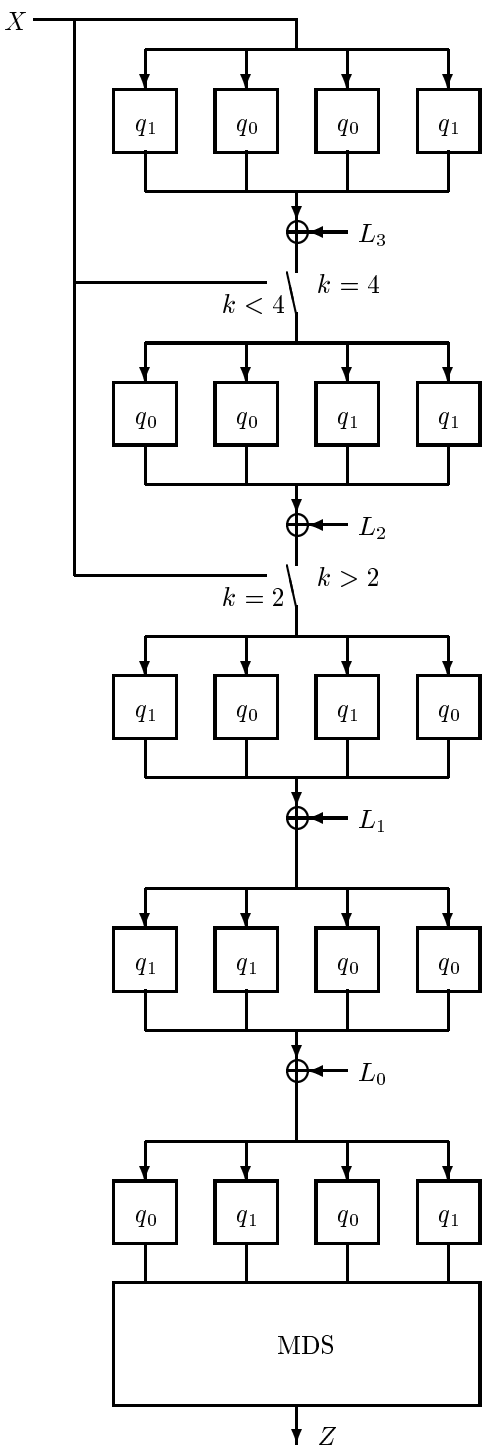

Figure 2: The function h

If $k = 4$ we have

$$\begin{aligned} y_{3,0} &= q_1[y_{4,0}] \oplus l_{3,0} \\ y_{3,1} &= q_0[y_{4,1}] \oplus l_{3,1} \\ y_{3,2} &= q_0[y_{4,2}] \oplus l_{3,2} \\ y_{3,3} &= q_1[y_{4,3}] \oplus l_{3,3} \end{aligned}$$

If $k \geq 3$ we have

$$\begin{aligned} y_{2,0} &= q_1[y_{3,0}] \oplus l_{2,0} \\ y_{2,1} &= q_1[y_{3,1}] \oplus l_{2,1} \\ y_{2,2} &= q_0[y_{3,2}] \oplus l_{2,2} \\ y_{2,3} &= q_0[y_{3,3}] \oplus l_{2,3} \end{aligned}$$

In all cases we have

$$\begin{aligned} y_0 &= q_1[q_0[q_0[y_{2,0}] \oplus l_{1,0}] \oplus l_{0,0}] \\ y_1 &= q_0[q_0[q_1[y_{2,1}] \oplus l_{1,1}] \oplus l_{0,1}] \\ y_2 &= q_1[q_1[q_0[y_{2,2}] \oplus l_{1,2}] \oplus l_{0,2}] \\ y_3 &= q_0[q_1[q_1[y_{2,3}] \oplus l_{1,3}] \oplus l_{0,3}] \end{aligned}$$

Here, q_0 and q_1 are fixed permutations on 8-bit values that we will define shortly. The resulting vector of y_i's is multiplied by the MDS matrix, just as in the g function.

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \cdot & \cdots & \cdot \\ \vdots & \text{MDS} & \vdots \\ \cdot & \cdots & \cdot \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$Z = \sum_{i=0}^{3} z_i \cdot 2^{8i}$$

where Z is the result of h.

4.3.3 The Key-dependent S-boxes

We can now define the S-boxes in the function g by

$$g(X) = h(X, S)$$

That is, for $i = 0, \ldots, 3$, the key-dependent S-box s_i is formed by the mapping from x_i to y_i in the h function, where the list L is equal to the vector S derived from the key.

4.3.4 The Expanded Key Words K_j

The words of the expanded key are defined using the h function.

$$\begin{aligned} \rho &= 2^{24} + 2^{16} + 2^8 + 2^0 \\ A_i &= h(2i\rho, M_e) \\ B_i &= \text{ROL}(h((2i+1)\rho, M_o), 8) \\ K_{2i} &= (A_i + B_i) \bmod 2^{32} \\ K_{2i+1} &= \text{ROL}((A_i + 2B_i) \bmod 2^{32}, 9) \end{aligned}$$

The constant ρ is used here to duplicate bytes; it has the property that for $i = 0, \ldots, 255$, the word $i\rho$ consists of four equal bytes, each with the value i. The function h is applied to words of this type. For A_i the byte values are $2i$, and the second argument of h is M_e. B_i is computed similarly using $2i + 1$ as the byte value and M_o as the second argument, with an extra rotate over 8 bits. The values A_i and B_i are combined in a PHT. One of the results is further rotated by 9 bits. The two results form two words of the expanded key.

4.3.5 The Permutations q_0 and q_1

The permutations q_0 and q_1 are fixed permutations on 8-bit values. They are constructed from four different 4-bit permutations each. For the input value x, we define the corresponding output value y as follows:

$$\begin{aligned} a_0, b_0 &= \lfloor x/16 \rfloor, x \bmod 16 \\ a_1 &= a_0 \oplus b_0 \\ b_1 &= a_0 \oplus \text{ROR}_4(b_0, 1) \oplus 8a_0 \bmod 16 \\ a_2, b_2 &= t_0[a_1], t_1[b_1] \\ a_3 &= a_2 \oplus b_2 \\ b_3 &= a_2 \oplus \text{ROR}_4(b_2, 1) \oplus 8a_2 \bmod 16 \\ a_4, b_4 &= t_2[a_3], t_3[b_3] \\ y &= 16\, b_4 + a_4 \end{aligned}$$

where ROR_4 is a function similar to ROR that rotates 4-bit values. First, the byte is split into two nibbles. These are combined in a bijective mixing step. Each nibble is then passed through its own 4-bit fixed S-box. This is followed by another mixing step and S-box lookup. Finally, the two nibbles are recombined into a byte. For the permutation q_0 the 4-bit S-boxes are given by

$$\begin{aligned} t_0 &= [\ 8\ 1\ 7\ D\ 6\ F\ 3\ 2\ 0\ B\ 5\ 9\ E\ C\ A\ 4\ ] \\ t_1 &= [\ E\ C\ B\ 8\ 1\ 2\ 3\ 5\ F\ 4\ A\ 6\ 7\ 0\ 9\ D\ ] \\ t_2 &= [\ B\ A\ 5\ E\ 6\ D\ 9\ 0\ C\ 8\ F\ 3\ 2\ 4\ 7\ 1\ ] \\ t_3 &= [\ D\ 7\ F\ 4\ 1\ 2\ 6\ E\ 9\ B\ 3\ 0\ 8\ 5\ C\ A\ ] \end{aligned}$$

where each 4-bit S-box is represented by a list of the entries using hexadecimal notation. (The entries for the inputs $0, 1, \ldots, 15$ are listed in order.) Similarly, for q_1 the 4-bit S-boxes are given by

$$\begin{aligned} t_0 &= [\ 2\ 8\ B\ D\ F\ 7\ 6\ E\ 3\ 1\ 9\ 4\ 0\ A\ C\ 5\ ] \\ t_1 &= [\ 1\ E\ 2\ B\ 4\ C\ 3\ 7\ 6\ D\ A\ 5\ F\ 9\ 0\ 8\ ] \\ t_2 &= [\ 4\ C\ 7\ 5\ 1\ 6\ 9\ A\ 0\ E\ D\ 8\ 2\ B\ 3\ F\ ] \\ t_3 &= [\ B\ 9\ 5\ 1\ C\ 3\ D\ E\ 6\ 4\ 7\ F\ 2\ 0\ 8\ A\ ] \end{aligned}$$

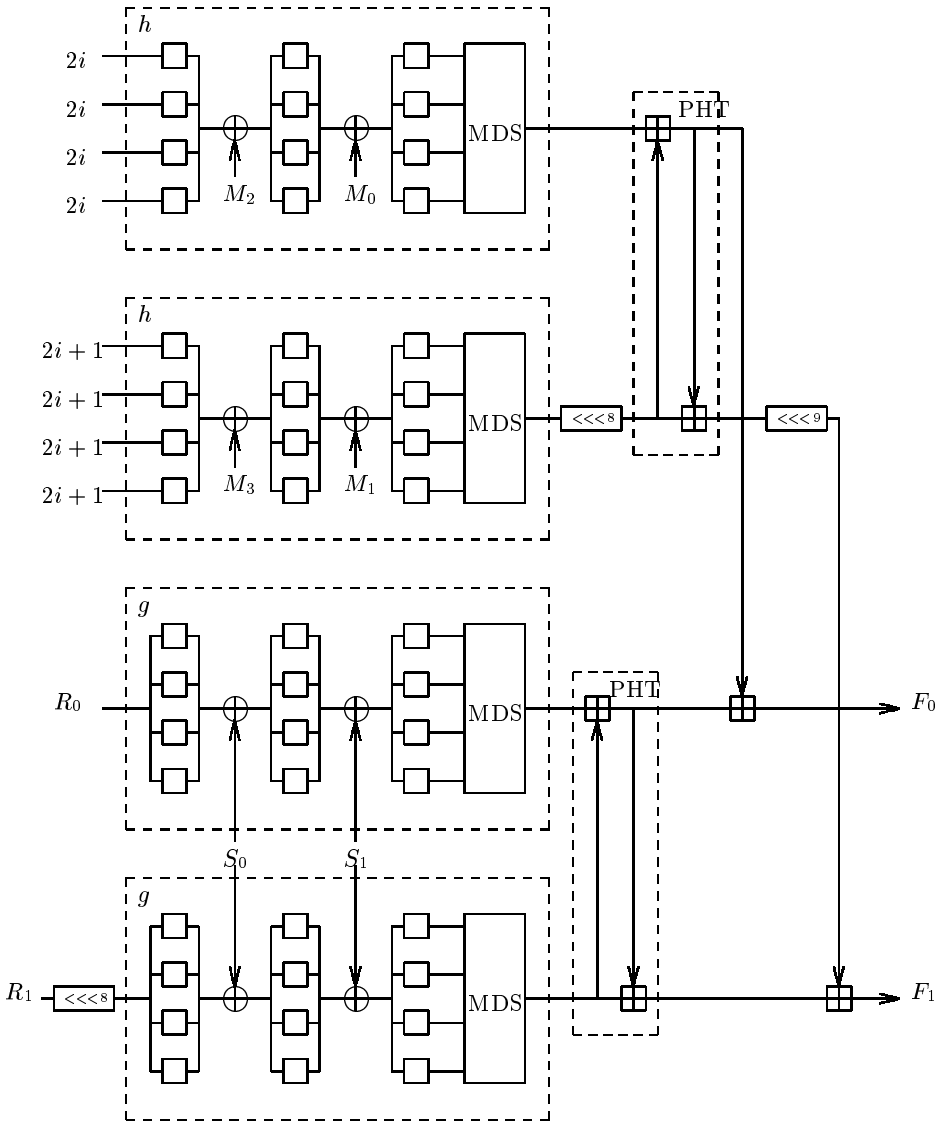

Figure 3: A view of a single round F function (128-bit key)

4.4 Round Function Overview

Figure 3 shows a more detailed view of how the function F is computed each round when the key length is 128 bits. Incorporating the S-box and round-subkey generation makes the Twofish round function look more complicated, but is useful for visualizing exactly how the algorithm works.

5 Performance of Twofish

Twofish has been designed from the start with performance in mind. It is efficient on a variety of platforms: 32-bit CPUs, 8-bit smart cards, and dedicated VLSI hardware. More importantly, though, Twofish has been designed to allow several layers of performance tradeoffs, depending on the relative importance of encrytion speed, key setup, memory use, hardware gate count, and other implementation parameters. The result is a highly flexible algorithm that can be implemented efficiently in a variety of cryptographic applications.

All these options are interoperable; these are simply implementation trade-offs and do not affect the mathematics of Twofish. One end of a communication could use the fastest Pentium II implementation, and the other the cheapest hardware implementation.

5.1 Performance on Large Microprocessors

Table 1 gives Twofish's performance, encryption or decryption, for different key scheduling options and on several modern microprocessors using different languages and compilers. The times for encryption and decryption are usually extremely close, so only the encryption time is given. There is no time required to set up the algorithm except for key setup. The time required to change a key is the same as the time required to setup a key. The approximate total code size (in bytes) of the routines for encryption, decryption, and key setup is also listed, where available.

All timing data is given in clock cycles per block, or clock cycles to set up the complete key. For example, on a Pentium Pro a fully optimized assembly-language version of Twofish can encrypt or decrypt data in 285 clock cycles per block, or 17.8 clock cycles per byte, after a 12700-clock key setup (equivalent to encrypting 45 blocks). On a 200 MHz Pentium Pro microprocessor, this translates to a throughput of just under 90 Mbits/sec.

We have implemented four different keying options. All of our keying options precompute K_i for $i = 0, \ldots, 39$ and use 160 bytes of RAM to store these constants. The differences occur in the way the function g is implemented. There are several other possible keying options, each with slightly different setup/throughput tradeoffs, but the examples listed below are representative of the range of possibilities.

Full Keying This option performs the full key precomputations. Using 4 Kb of table space, each S-box is expanded to a 8-by-32-bit table that combines both the S-box lookup and the multiply by the column of the MDS matrix. Using this option, a computation of g consists of four table lookups, and three XORs. Encryption and decyption speeds are constant regardless of key size.

Partial Keying For applications where few blocks are encrypted with a single key, it may not make sense to build the complete key schedule. The partial keying option precomputes the four S-boxes in 8-by-8 bit tables, and uses four fixed 8-by-32-bit MDS tables to perform the MDS multiply. This reduces the key-schedule table space to 1 Kb. For each byte, the last of the q-box lookups is in fact incorporated into the MDS table, so only k of the q-boxes are incorporated into the 8-by-8-bit S-box table that is built by the key schedule. Encryption and decryption speed are again constant regardless of key size.

Minimal Keying For applications where very few blocks are encrypted with a single key, there is a further possible optimization. Compared to partial keying, one less layer of q-boxes is precomputed into the S-box table, and the remaining q-box is done during the encryption. For the 128-bit key this is particularly efficient as precomputing the S-boxes now consists of copying the table of the appropriate q-box and XORing it with a constant (which can be done word-by-word instead of byte-by-byte). This option uses a 1 Kb table to store the partially precomputed S-boxes. The necessary key bytes from S are of course precomputed as they are needed in every round.

Zero Keying The zero keying option does not precompute any of the S-boxes, and thus needs no extra tables. Instead, every entry is computed on

Processor	Language	Keying Option	Code Size	Clocks to Key			Clocks to Encrypt		
				128-bit	192-bit	256-bit	128-bit	192-bit	256-bit
Pentium Pro/II	Assembly	Compiled	8900	12700	15400	18100	285	285	285
Pentium Pro/II	Assembly	Full	8450	7800	10700	13500	315	315	315
Pentium Pro/II	Assembly	Partial	10700	4900	7600	10500	460	460	460
Pentium Pro/II	Assembly	Minimal	13600	2400	5300	8200	720	720	720
Pentium Pro/II	Assembly	Zero	9100	1250	1600	2000	860	1130	1420
Pentium Pro/II	MS C	Full	11200	8000	11200	15700	600	600	600
Pentium Pro/II	MS C	Partial	13200	7100	9700	14100	800	800	800
Pentium Pro/II	MS C	Minimal	16600	3000	7800	12200	1130	1130	1130
Pentium Pro/II	MS C	Zero	10500	2450	3200	4000	1310	1750	2200
Pentium Pro/II	Borland C	Full	14100	10300	13600	18800	640	640	640
Pentium Pro/II	Borland C	Partial	14300	9500	11200	16600	840	840	840
Pentium Pro/II	Borland C	Minimal	17300	4600	10300	15300	1160	1160	1160
Pentium Pro/II	Borland C	Zero	10100	3200	4200	4800	1910	2670	3470
Pentium	Assembly	Compiled	8900	24600	26800	28800	290	290	290
Pentium	Assembly	Full	8200	11300	14100	16000	315	315	315
Pentium	Assembly	Partial	10300	5500	7800	9800	430	430	430
Pentium	Assembly	Minimal	12600	3700	5900	7900	740	740	740
Pentium	Assembly	Zero	8700	1800	2100	2600	1000	1300	1600
Pentium	MS C	Full	11800	11900	15100	21500	630	630	630
Pentium	MS C	Partial	14100	9200	13400	19800	900	900	900
Pentium	MS C	Minimal	17800	3800	11100	16900	1460	1460	1460
Pentium	MS C	Zero	11300	2800	3900	4900	1740	2260	2760
Pentium	Borland C	Full	12700	14200	18100	26100	870	870	870
Pentium	Borland C	Partial	14200	11200	16500	24100	1100	1100	1100
Pentium	Borland C	Minimal	17500	4700	12100	19200	1860	1860	1860
Pentium	Borland C	Zero	11800	3700	4900	6100	2150	2730	3270
UltraSPARC	C	Full		16600	21600	24900	750	750	750
UltraSPARC	C	Partial		8300	13300	19900	930	930	930
UltraSPARC	C	Minimal		3300	11600	16600	1200	1200	1200
UltraSPARC	C	Zero		1700	3300	5000	1450	1680	1870
PowerPC 750	C	Full		12200	17100	22200	590	590	590
PowerPC 750	C	Partial		7800	12200	17300	780	780	780
PowerPC 750	C	Minimal		2900	9100	14200	1280	1280	1280
PowerPC 750	C	Zero		2500	3600	4900	1030	1580	2040
68040	C	Full	16700	53000	63500	96700	3500	3500	3500
68040	C	Partial	18100	36700	47500	78500	4900	4900	4900
68040	C	Minimal	23300	11000	40000	71800	8150	8150	8150
68040	C	Zero	16200	9800	13300	17000	6800	8600	10400

Table 1: Twofish performance with different key lengths and options

the fly. The key setup time consists purely of computing the K_i values and S. For an application that cannot have any key setup time, the time it takes to encrypt one block is the sum of the key setup time and encryption time for the zero keying option.

Compiled In this option, available only in assembly language, the subkey constants are directly embedded into a key-specific copy of the code, saving memory fetches and allowing the use of the Pentium LEA opcode to perform both the PHT and the subkey addition all in a single clock. Some additional setup time is required, as well as about an extra 5000 bytes of memory to hold the "compiled" code, but this option allows the fastest execution time of 285 clocks per block on the Pentium Pro. The setup time for a Pentium more than doubles over the full keying case, because of its smaller cache size, but the Pentium MMX setup time is still comparable to the Pentium Pro setup time. However, almost all the extra time required is consumed merely in copying the code; the table does not reflect the fact that, once a single key has been initialized, future keys can be compiled at a cost of only a few hundred more clocks than full key schedule time.

Language, Compiler, and Processor Choice As with most algorithms, the choices of language and compiler can have a huge impact on performance. It is clear that the Borland C 5.0 compiler chosen as the standard AES reference is not the best optimizing compiler. For example, the Microsoft Visual C++ 4.2 compiler generates Twofish code that is at least 20% faster than Borland on a Pentium computer, with both set to optimize for speed (630 clocks per block for Microsoft Visual C++ 4.2 versus 870 clocks per block for Borland C 5.0); on a Pentium Pro/II, the difference between the compilers is not quite as large (e.g., 600 clocks/block vs. 640 clocks/block), but it is still significant. Part of the difference stems from the inability of the Borland compiler to generate intrinsic rotate instructions, despite documentation claiming that it is possible. This problem alone accounts for nearly half of the speed difference between Borland and Microsoft. The remaining speed difference comes simply from poorer code generation. The Borland compiler is uniformly slower than Microsoft's compiler. The encryption speed in Microsoft C of 40 Pentium clocks per byte (i.e., 630 clocks/block at 16 bytes/block) is over ten percent faster than the best known DES assembly language implementation on the same platform. However, coding the Twofish algorithm in assembly language achieves speeds of 285 clocks, achieving a very significant speedup over any of the C implementations.

To make matters even more complicated, the assembly language that optimizes performance on a Pentium (or Pentium MMX) is drastically different from the assembly language required to maximize speed on a Pentium Pro or Pentium II, even though the final code size and speed achieved on each platform are almost identical. For example, the Pentium Pro/II CPUs can only perform one memory read per clock cycle, while the Pentium and Pentium MMX can perform two. However, the Pentium Pro/II can peform two ALU operations per clock in addition to memory accesses, while the Pentium can process only a total of two ALU operations or memory accesses per clock. These (and other) significant architectural differences result in the fact that running the optimal Pentium Twofish encryption code on a Pentium Pro results in a slowdown of nearly 2:1, and vice versa! Fortunately, it is relatively simple to detect the CPU type at run-time and select which version of the assembly code to use. Another anomaly is that the key schedule setup time is considerably faster (43%) on the Pentium MMX CPU than on the Pentium, not because the key schedule uses any MMX instructions, but simply because of the larger cache size of the MMX chip.

Empirically, there also seems to be some anomalous behavior of the C compilers. In almost all cases, the encryption and decryption routines in C achieved speeds within a few percent of each other. However, there were cases in the table where the two speeds differed by considerably more than ten percent (we used the larger number in the table), which is very odd because the "inner loop" C code used is virtually identical. We also noted several cases where compiler switches which seemed unrelated to performance optimization sometimes caused very large changes in timings.

It should be noted that performance numbers for Pentium II processors are almost identical in all cases to those for a Pentium Pro, which is not surprising since Intel claims they have the same core. The Pentium and Pentium MMX achieve almost identical speeds for encryption and decryption, although, as noted above, the MMX key setup times are faster, due mainly to the larger cache size.

The bottom line is that, when comparing the relative performance of different algorithms, using the same language and compiler for all implementations helps to make the comparison meaningful, but it does not guarantee a valid measure of the relative speeds. We have listed many different software per-

formance metrics across platforms and languages to facilitate speed comparisons between Twofish and other algorithms. Our belief is that, on any given platform (e.g., Pentium Pro), the assembly language performance numbers are the best numbers to use to gauge absolute performance, since they are unaffected by the vagaries and limitations of the compiler (e.g., inability to produce rotate opcodes). High-level languages (e.g., C, Java) are also important because of the ease of porting to different platforms, but once an algorithm becomes standardized, it will ultimately be coded in assembly for the most popular platforms.

Code and Data Size As shown in Table 1, the code size for a fully optimized Twofish on a Pentium Pro ranges from about 8450 bytes in assembler to 14100 bytes in Borland C. In assembler, the encryption and decryption routines are each about 2250 bytes in size; the remaining 4000 bytes of code are in the key scheduling routines, but about 2200 bytes of that total can be discarded if only 128-bit keys are needed. In Borland C, the encryption and decryption routines are each about 4500 bytes in length, and the key schedule routine is slightly less than 5000 bytes. Note that each routine fits easily within the code cache of a Pentium or a Pentium Pro. These sizes are larger than Blowfish but very similar to a fully optimized assembly language version of DES. Note that, with the exception of the zero keying option, the code sizes in the table are for fully unrolled implementations; in either C or assembler, it is possible to achieve significantly smaller code sizes using loops with round counters, at a cost in performance.

In addition to the code, there are about 4600 bytes of fixed tables for the MDS matrix, q_0, and q_1 required for key setup, and each key requires about 4300 bytes of key-dependent data tables for the full keying option. During encryption or decryption, these tables fit easily in the Pentium data cache. The other keying options use less data and table space, as discussed above.

Total Encryption Times Any performance measures that do not take key setup into account are only valid for asymptotically large amounts of text. For shorter messages, performance is the sum of key setup and encryption. For very short messages, the key setup time can overwhelm the encryption speed.

Table 2 gives Twofish's performance on the Pentium Pro (assembly-language version), both 128-bit key setup and encryption, for a variety of message lengths. This table assumes the best of our implementations for the particular length of text.

5.2 Performance on Smart Cards

Twofish has been implemented on a 6805 CPU, which is a typical smart card processor, with several different space–time tradeoff options. Our different options result in the following numbers:

RAM (bytes)	Code and Table Size	Clocks per Block	Time per block @ 4MHz
60	2200	26500	6.6 msec
60	2150	32900	8.2 msec
60	2000	35000	8.7 msec
60	1760	37100	9.3 msec

The code size includes both encryption and decryption.[4] The block encryption and decryption times are almost identical. If only encryption is required, minor improvements in code size and speed can be obtained. The only key schedule precomputation time required in this implementation is the Reed-Solomon mapping used to generate the S-box key material S from the key M, which requires slightly over 1750 clocks per key. This setup time could be cut considerably at the cost of two additional 512-byte ROM tables. It should also be observed that the lack of a second index register on the 6805 has a significant impact on the code size and performance, so a different CPU with multiple index registers (e.g., 6502) might be a better fit for Twofish.

Size and speed estimates for larger key sizes are very straightforward, given the 128-bit implementation. The extra code size is fairly negligible—less than 100 extra bytes for a 192-bit key, and less than 200 bytes for a 256-bit key. The encryption time per block increases by less than 2600 clocks per block (for any of the code size/speed tradeoffs above) for 192-bit keys, and by about 5200 clocks per block for 256-bit keys. Similarly, the key schedule precomputation increases to 2550 clocks for 192-bit keys, and to 3400 clocks for 256-bit keys.

The 60 bytes of RAM for the smart card implementation include 16 bytes for the plaintext/ciphertext block and 16 bytes for the key. If the 16 key bytes and the extra 8 bytes of the Reed-Solomon results (S) are retained in memory between encryption operations, the remaining 36 bytes of RAM are available for other functions, and there is zero startup

[4] For comparison purposes: DES on a 6805 takes about 2K code, 23 bytes of RAM, and 20000 clock cycles per block.

Plaintext (bytes)	Keying Option	Clocks to Key	Clocks to Encrypt	Total Clocks per Byte
16	Zero	1250	860	131.9
32	Zero	1250	1720	92.8
64	Zero	1250	4690	73.3
128	Zero	1250	6880	63.5
256	Partial	4900	7360	47.9
512	Full	7800	10080	34.9
1K	Full	7800	20160	27.3
2K	Full	7800	40320	23.5
4K	Compiled	12700	72960	20.9
8K	Compiled	12700	145920	19.4
16K	Compiled	12700	291840	18.6
32K	Compiled	12700	583680	18.2
64K	Compiled	12700	1167360	18.0
1M	Compiled	12700	18677760	17.8

Table 2: Best speed to encrypt a message with a new 128-bit key on a Pentium Pro

time for the next encryption operation with the same key. Note that larger key sizes also require more RAM to store the larger keys and the larger Reed-Solomon results. In some applications it might be viable to store all key material in non-volatile memory, reducing the RAM requirements of the implementation significantly.

Observe that it is possible to save further ROM space by computing q_0 and q_1 lookups using the underlying 4-bit construction, as specified in Section 4.3.5. Such a scheme would replace 512 bytes of ROM table with 64 bytes of ROM and a small subroutine to compute the full 8-bit q_0 and q_1, saving perhaps 350 bytes of ROM; unfortunately, encryption speed would decrease by a factor of ten or more. Thus, this technique is only of interest in smart card applications for which ROM size is extremely critical but performance is not. Nonetheless, such an approach illustrates the implementation flexibility afforded by Twofish.

5.3 Performance on Future Microprocessors

Given the ever-advancing capabilities of CPUs, it is worthwhile to make some observations about how the Twofish algorithm will run on future processors, including Intel's Merced. Not many details are known about Merced, other than that it includes an Explicitly Parallel Instruction Computing (EPIC) architecture, as well the ability to run existing Pentium code. EPIC is related to VLIW architectures that allow many parallel opcodes to be executed at once, while the Pentium allows only two opcodes in parallel, and the Pentium Pro/Pentium II may process up to three opcodes per clock. However, access to memory tables is limited in most VLIW implementations to only a few parallel operations, and we expect similar restrictions to hold for Merced. For example, an existing Philips VLIW CPU can process up to five opcodes in parallel, but only two of the opcodes can read from memory.

Since Twofish relies on 8-bit non-linear S-boxes, it is clear that table access is an integral part of the algorithm. Thus, Twofish might not be able to take advantage of all the parallel execution units available on a VLIW processor. However, there is still plenty of parallelism in Twofish that can be well-utilized in an optimized VLIW software implementation. Equally important, the alternative of not using large S-boxes, while it may allow greater parallelism, also naturally involves less non-linearity and thus generally requires more rounds. For example, Serpent [BAK98], based on "inline" computation of 4-bit S-boxes, may experience a relatively larger speedup than Twofish on a VLIW CPU, but Serpent also requires 32 rounds, and is considerably slower to start with.

It should also be noted that, as with most encryption algorithms, the primitive operations used in Twofish could very easily be added to a CPU instruction set to improve software performance significantly. Future mainstream CPUs may include such support for the new AES standard. However, it is also worthwhile to remember that DES has been

a standard for more than twenty years, and no popular CPU has added instruction set support for it, even though DES software performance would benefit greatly from such features.

5.4 Hardware Performance

No actual logic design has been implemented for Twofish, but estimates in terms of gates for each building block have been made. As in software, there are many possible space–time tradeoffs in hardware implementations of Twofish. Thus, it is not meaningful to give just one figure for the speed and size attributes of Twofish in hardware. Instead, we will try to outline several of the options and give estimate for speed and gate count of several different architectures.

For example, the round subkeys could be precomputed and stored in a RAM, or they could be computed on the fly. If computed on the fly, the h function logic could be time-multiplexed between subkeys and the round function to save size at a cost in speed, or the logic could be duplicated, adding gates but perhaps running twice as fast. If the subkeys were precomputed, the h function logic would be used during a key setup phase to compute the subkeys, saving gates but adding a startup time roughly equal to one block encryption time. Similarly, a single h function logic block could be time-multiplexed between computing T_0 and T_1, halving throughput but saving even more gates.

As another example of the possible tradeoffs, the S-boxes could be precomputed and stored in on-chip RAMs, allowing faster operation because there is no need to ripple through several layers of key material XORs and q permutations. The addition of such RAMs (e.g., eight 256-byte RAMs) would perhaps double or triple the size of the logic, and it would also impose a significant startup time on key change to initialize the RAMs. Despite these disadvantages, such an architecture might raise the throughput by a factor of two or more (particularly for the larger key sizes), so for high-performance systems with infrequent re-keying, this option may be attractive.

The construction method specified in Section 4.3.5 for building the 8-bit permutations q_0 and q_1 from four 4-bit permutations was selected mainly to minimize gate count in many hardware implementations of Twofish. These permutations can be built either directly in logic gates or as full 256-byte ROMs in hardware, but such a ROM is usually several times larger than the direct logic implementation. Since each full h block in hardware (see Figure 2) involves six q_0 blocks and six q_1 blocks (for $N = 128$), the gate savings mount fairly quickly. The circuit delays in building q_0 or q_1 using logic gates are typically at least as fast as those using ROMs, although this metric is certainly somewhat dependent on the particular silicon technology and circuit library available.

It should also be noted that the Twofish round structure can be very nicely pipelined to break up the overall function into smaller and much faster blocks (e.g., q's, key XORs, MDS, PHT, subkey addition, Feistel XOR). None of these operations individually is slow, but trying to run all of them in a single clock cycle does affect the cycle time. In ECB mode, counter mode, or an interleaved chaining mode, the throughput can be dramatically increased by pipelining the Twofish round structure. As a very simple example of two-level pipelining, we could compute the q_i's, key XORs, and MDS multiply for one block during "even" clocks, while the PHT, subkey addition, and Feistel XOR would be computed on the "odd" clocks; a second block is processed in parallel on the alternate clock cycles. Using careful balancing of circuit delays between the two clock phases, this approach allows us to cut the logic delay in half, thus running the clock at twice the speed of an unpipelined approach. Such an approach does not require duplicating the entire Twofish round function logic, but merely the insertion of one extra layer of clocked storage elements (128 bits). Thus, for a very modest increase in gate count, throughput can be doubled, assuming that the application can use one of these cipher modes. It is clear that this general approach can be applied with more levels of pipelining to get higher throughput, although diminishing returns are achieved past a certain point. For even higher levels of performance, multiple independent engines can be used to achieve linear speedups at a linear cost in gates. We see no problem meeting NSA's requirement to "be able to encrypt data at a minimum of 1 Gb/s, pipelined if necessary, in existing technology" [McD97].

Table 3 gives hardware size and speed estimates for the case of 128-bit keys. Depending on the architecture, the logic will grow somewhat in size for larger keys, and the clock speed (or startup time) may increase, but it is believed that a 128-bit AES scheme will be acceptable in the market long enough that most vendors will choose to implement that recommended key length. These estimates are all based on existing 0.35 micron CMOS technology. All the examples in the table are actually quite small in today's technology, except the final (highest performance non-pipelined) instance, but even that is very

Gate count	h blocks	Clocks per Block	Pipeline Levels	Clock Speed	Throughput (Mbits/sec)	Startup clocks	Comments
14000	1	64	1	40 MHz	80	4	Subkeys on the fly
19000	1	32	1	40 MHz	160	40	
23000	2	16	1	40 MHz	320	20	
26000	2	32	2	80 MHz	640	20	
28000	2	48	3	120 MHz	960	20	
30000	2	64	4	150 MHz	1200	20	
80000	2	16	1	80 MHz	640	300	S-box RAMs

Table 3: Hardware tradeoffs (128-bit key)

doable today and will become fairly inexpensive as the next generation silicon technology (0.25 micron) becomes the industry norm.

6 Twofish Design Philosophy

In the design of Twofish, we tried to stress the following principles:

Performance When comparing different options, compare them on the basis of relative performance.

Conservativeness Do not design close to the edge. In other words, leave a margin for error and provide more security than is provably required. Also, try to design against attacks that are not yet known.

Simplicity Do not include ad hoc design elements without a clear reason or function. Try to design a cipher whose details can be easily kept in one's head.

These principles were applied not only to the overall design of Twofish, but to the design of the S-boxes and the key schedule.

6.1 Performance-Driven Design

The goal of performance-driven design is to build and evaluate ciphers on the basis of performance [SW97]. The early post-DES cipher designs would often compete on the number of rounds in the cipher. The original FEAL paper [SM88], for example, discussed the benefits of a stronger round function and fewer rounds. Other cipher designs of the period—REDOC II [CW91], LOKI [BPS90] and LOKI 93 [BKPS93], IDEA [LM91, LMM91]—only considered performance as an afterthought. Khufu/Khafre [Mer91] was the first published algorithm that explicitly used operations that were efficient on 32-bit microprocessors; SEAL [RC94, RC97] is a more recent example. RC2 [Riv97, KRRR98] was designed for 16-bit microprocessors, SOBER [Ros98] for 8-bit ones. Other, more recent designs, do not seem to take performance into account at all. Two 1997 designs, SPEED [Zhe97][5] and Zhu-Guo [ZG97], are significantly slower than alternatives that existed years previous.

Arbitrary metrics, such as the number of rounds, are not good measures of performance. What is important is the cipher's speed: the number of clock cycles per byte encrypted. When ciphers are analyzed according to this property, the results can be surprising [SW97]. RC5 might have twice the number of rounds of DES,[6] but since its round function is more than twice as fast as DES', RC5 is faster than DES on most microprocessors.

Even when cryptographers made efforts to use efficient 32-bit operations, they often lacked a full appreciation of low-level software optimization principles associated with high-performance CPUs. Thus, many algorithms are not as efficient as they could be. Minor modifications in the design of Blowfish [Sch94], SEAL [RC94, RC97], and RC4 [Sch96] could improve performance without affecting security [SW97] (or, alternatively, increase the algorithms' complexity without affecting performance). In designing Twofish, we tried to evaluate all design decisions in terms of performance.

[5] Speed has been cryptanalyzed in [HKSW98, HKR+98].

[6] Here we use the term "round" in the traditional sense: as it was defined by DES [NBS77] and has been used to describe Feistel-network ciphers ever since. The RC5 documentation [Riv95] uses the term "round" differently: one RC5-defined round equals two Feistel rounds.

Since NIST's platform of choice was the Intel Pentium Pro [NIST97b], we concentrated on that platform. However, we did not ignore performance on other 32-bit CPUs, as well as 8-bit and 16-bit CPUs. If there is any lesson from the past twenty years of microprocessors, it is that the high end gets better and the low end never goes away. Yesterday's top-of-the-line CPUs are currently in smart cards. Today's CPUs will eventually be in smart cards, while the 8-bit microprocessors will move to devices even smaller. The only thing we did not consider in our performance metrics are bitslice implementations [Bih97, SAM97, NM97], since these can only be used in very specialized applications and often require unrealistic implementations: e.g., 32 simultaneous ECB encryptions, or 32 interleaved IVs.[7]

6.1.1 Performance-driven Tradeoffs

During our design, we constantly evaluated the relative performance of different modifications to our round function. Twofish's round function encrypts at about 20 clock cycles; 16 rounds translates to about 320 clock cycles per block encrypted. When we contemplated a change to the round function, we evaluated it in terms of increasing or decreasing the number of rounds to keep performance constant.

Three examples:

- We could have added a data-dependent rotation to the output of the two MDS matrices in each round. This would add 10 clock cycles to the round function on the Pentium (2 on the Pentium Pro). To keep the performance constant, we would have to reduce the number of rounds to 11. The question to ask is: Are 11 rounds of the modified cipher more or less secure than 16 rounds of the unmodified cipher?
- We could have removed the one-bit rotation. This would have saved clocks equivalent to one Twofish round. Are 17 rounds of this new round function more or less secure than 16 rounds of the old one?
- We could have defined the key-dependent S-boxes using the whole key, instead of half of it. This would have doubled key setup time on high-end machines, and halved encryption speed on memory-poor implementations (where the S-boxes could not be precomputed). On memory-poor machines, we would have to cut the number of rounds in half to be able to afford this. Are 8 rounds of this improved cipher better than 16 rounds of the current design?

This analysis is necessarily dependent on the microprocessor architecture the algorithm is being compared on. While we focused on the Intel Pentium architecture, we also tried to keep 8-bit smart card and hardware implementations in mind. For example, we considered using a 8-by-8 MDS matrix over GF(2^4) to ensure a finer-grained diffusion, instead of a 4-by-4 MDS matrix over GF(2^8); the former would have been no slower on a Pentium but at least twice as slow on a low-memory smart card.

6.2 Conservative Design

There has been considerable research in designing ciphers to be resistant to known attacks [Nyb91, Nyb93, OCo94a, OCo94b, OCo94c, Knu94a, Knu94b, Nyb94, DGV94b, Nyb95, NK95, Mat96, Nyb96], such as differential [BS93], linear [Mat94], and related-key cryptanalysis [Bih94, KSW96, KSW97]. This research has culminated in strong cipher designs—CAST-128 [Ada97a] and MISTY [Mat97] are probably the most noteworthy—as well as some excellent cryptanalytic theory.

However, it is dangerous to rely solely on theory when designing ciphers. Ciphers provably secure against differential cryptanalysis have been attacked with higher-order differentials [Lai94, Knu95b] or the interpolation attack [JK97]: KN-cipher [NK95] was attacked in [JK97, SMK98], Kiefer [Kie96] in [JK97], and a version of CAST in [MSK98a]. The CAST cipher cryptanalyzed in [MSK98a] is not CAST-128, but it does illustrate that while the CAST design procedure [AT93, HT94] can create ciphers resistant to differential and linear cryptanalysis, it does not create ciphers resistant to whatever form of cryptanalysis comes next. SNAKE [LC97], another cipher provably secure against differential and linear cryptanalysis, was successfully broken using the interpolation attack [MSK98b]. When designing a cipher, it is prudent to assume that new attacks will be developed in order to break it.

We took a slightly different approach in our design. Instead of trying to optimize Twofish against known attacks, we tried to make Twofish strong against both known and unknown attacks. While it is impossible to optimize a cipher design for resisting

[7] One AES submission, Serpent [BAK98], uses ideas from bitslice implementations to create a cipher that is very efficient on 32-bit processors while sacrificing performance on 8-bit microprocessors.

attacks that are unknown, conservative design and over-engineering can instill some confidence.

Many elements of Twofish reflect this philosophy. We used well-studied design elements throughout the algorithm. We started with a Feistel network, probably the most studied block-cipher structure, instead of something newer like an unbalanced Feistel network [SK96, ZMI90] or a generalized Feistel network [Nyb96].

We did not implement multiplication mod $2^{16} + 1$ (as in IDEA or MMB [DGV93]) or data-dependent rotations (as in RC5[8] or Akelarre [AGMP96][9]) for non-linearity. The most novel design elements we used—MDS matrices and PHTs—are only intended for diffusion (and are used in Square [DKR97] and SAFER, respectively).

We used key-dependent S-boxes, because they offer adequate protection against known statistical attacks and are likely to offer protection to any unknown similar attacks. We defined Twofish at 16 rounds, even though our analysis cannot break anywhere near that number. We added one-bit rotations to prevent potential attacks that relied solely on the byte structure. We designed a very thorough key schedule to prevent related-key and weak-key attacks.

6.3 Simple Design

A guiding design principle behind Twofish is that the round function should be simple enough for us to keep in our heads. Anecdotal evidence from algorithms like FEAL [SM88], CAST, and Blowfish indicates that complicated round functions are not always better than simple ones. Also, complicated round functions are harder to analyze and rely on more ad-hoc arguments for security (e.g., REDOC-II [CW91]).

However, with enough rounds, even bad round functions can be made to be secure.[10] Even a simple round function like TEA's [WN95] or RC5's seems secure after 32 rounds [BK98]. In Twofish, we tried to create a simple round function and then iterate it more than enough times for security.

6.3.1 Reusing Primitives

One of the ways to simplify a design is to reuse the same primitives in multiple parts of a cipher. Cryptographic design does not lend itself to the adage of not putting all your eggs in one basket. Since any particular "basket" has the potential of breaking the entire cipher, it makes more sense to use as few baskets as possible—and to scrutinize those baskets intensely.

To that end, we used essentially the same construction (8-by-8-bit key-dependent S-boxes consisting of alternating fixed permutations and subkey XORs followed by an MDS matrix followed by a PHT) in both the key schedule and the round function. The differences were in the key material used (the round function's g function uses a list of key-derived words processed by an RS code; the key schedule's h function uses individual key bytes directly) and the rotations. The rotations represent a performance-driven design tradeoff: putting the additional rotations into F would have unacceptably slowed down the cipher performance on high-end machines. The use of the RS code to derive the key material for g adds substantial resistance to related-key attacks.

While many algorithms reuse the encryption operation in their key schedule (e.g., Blowfish, Panama [DC98], RC4, CRISP [Lee96], YTH [YTH96]), and several alternative DES key schedules reuse the DES operation [Knu94b, BB96], we are unaware of any that reuse the same primitives in exactly this manner.[11] We feel that doing so greatly simplifies the analysis of Twofish, since the same kinds of analysis can apply to the cipher in two different ways.

6.3.2 Reversibility

While it is essential that any block cipher be reversible, so that ciphertext can be decrypted back into plaintext, it is not necessary that the identical function be used for encryption and decryption. Some block ciphers are reversible with changes only in the key schedule (e.g., DES, IDEA, Blowfish), while others require different algorithms for encryption and decryption (e.g., SAFER, Serpent, Square).

The Twofish encryption and decryption round functions are slightly different, but are built from the

[8] RC5's security is almost wholly based on data-dependent rotations. Although initial cryptanalysis was promising [KY95] (see also [Sel98]), subsequent research [KM97, BK98] suggests that there is considerably more to learn about the security properties of data-dependent rotations.

[9] Akelarre was severely broken in [FS97, KR97].

[10] Student cryptography projects bear this observation out. At 16 rounds, the typical student cipher fares rather badly against a standard suite of statistical tests. At 32 rounds, it looks better. At 128 rounds, even the worst designs look very good.

[11] The closest idea is an alternate DES key schedule that uses the DES round function, both the 32-bit block input and 48-bit key input, to create round subkeys [Ada97b].

same blocks. That is, it is simple to build a hardware or software module that does both encryption and decryption without duplicating much functionality, but the exact same module cannot both encrypt and decrypt.

Note that having the cipher work essentially the same way in both directions is a nice feature in terms of analysis, since it lets analysts consider chosen-plaintext and chosen-ciphertext attacks at once, rather than considering them as separate attacks with potentially radically different levels of difficulty [Cop98].

6.4 S-boxes

The security of a cipher can be very sensitive to the particulars of its S-boxes: size, number, values, usage. Ciphers invented before the public discovery of differential cryptanalysis sometimes used arbitrary sources for their S-box entries.

Randomly constructed known S-boxes are unlikely to be secure. Khafre uses S-boxes taken from the RAND tables [RAND55], and it is vulnerable to differential cryptanalysis [BS92]. NewDES [Sco85],[12] with S-boxes derived from the Declaration of Independence [Jeff+76], could be made much stronger with good S-boxes. DES variants with random fixed S-boxes are very likely to be weak [BS93, Mat95], and CMEA was weakened extensively because of a poor S-box choice [WSK97].

Some cipher designers responded to this threat by carefully crafting S-boxes to resist known attacks—DES [Cop94], s^nDES [KPL93, Knu93c, KLPL95], CAST [MA96, Ada97a]—while others relied on random key-dependent S-boxes for security—Khufu, Blowfish, WAKE [Whe94].[13] The best existing attack on Khufu breaks 16 rounds [GC94], while the best attack on Blowfish breaks only four [Rij97]. Serpent [BAK98] reused the DES S-boxes.

GOST [GOST89] navigated a middle course: each application has different fixed S-boxes, turning them into an application-specific family key.

6.4.1 Large S-boxes

S-boxes vary in size, from GOST's 4-by-4-bit S-boxes to Tiger's 8-by-64-bit S-boxes [AB96b]. Large S-boxes are generally assumed to be more secure than smaller ones—a view we share—but at the price of increased storage requirements; DES' eight 6-by-4-bit S-boxes require 256 bytes of storage, while Blowfish's four 8-by-32-bit S-boxes require 4 kilobytes. Certainly input size matters more than output size; an 8-by-64-bit S-box can be stored in 2 kilobytes, while a 16-by-16-bit S-box requires 128 kilobytes. (Note that there is a limit to the advantages of making S-boxes bigger. S-boxes with small input size and very large output size tend to have very good linear approximations; S-boxes with sufficiently large outputs relative to input size are *guaranteed* to have at least one perfect linear approximation [Bih95].)

Twofish used the same solution as Square: mid-sized S-boxes (8-by-8-bit) used to construct a large S-box (8-by-32-bit).

6.4.2 Algorithmic S-boxes

S-boxes can either be large tables, like DES, Khufu/Khafre, and YLCY [YLCY98], or derived algebraically, like FEAL, LOKI-89/LOKI-91 (and LOKI97 [Bro98]), IDEA, and SAFER. The advantage of the former is that there is no mathematical structure that can potentially be used for cryptanalysis. The advantage of the latter is that the S-boxes are more compact, and can be more easily implemented in applications where the ROM or RAM for large tables is not available.

Algebraic S-boxes can result in S-boxes that are vulnerable to differential cryptanalysis: [Mur90] against FEAL, and [Knu93a, Knu93b] against LOKI. Higher-order differential cryptanalysis is especially powerful against algorithms with simple algebraic S-boxes [Knu95b, JK97, SMK98]. Both tabular and algebraic techniques, however, can be used to generate S-boxes with given cryptographic properties, simply by testing the results of the generation algorithm.

In Twofish we tried to do both: we chose to build our 8-by-8-bit S-boxes algorithmically out of random 4-by-4-bit S-boxes. However, we chose the 4-by-4-bit S-boxes randomly and then extensively tested the resulting 8-by-8-bit S-boxes against the cryptographic properties we required. This idea is similar to the one used in CS-cipher [SV98].

[12] Despite the algorithm name, NewDES is neither a DES variant nor a new algorithm based on DES.

[13] The WAKE design has several variants [Cla97, Cla98]; neither the basic algorithm nor its variants have been extensively cryptanalyzed.

6.4.3 Key-dependent S-boxes

S-boxes are either fixed for all keys or key dependent. It is our belief that ciphers with key-dependent S-boxes are, in general, more secure than fixed S-boxes.

There are two different philosophies regarding key-dependent S-boxes. In some ciphers, the S-box is constructed specifically to ensure that no two entries are identical—Khufu and WAKE—while others simply create the S-box randomly and hope for the best: REDOC II [CW91] and Blowfish [Sch94]. The latter results in a simpler key schedule, but may result in weaknesses (e.g., a weakness in reduced-round variants of Blowfish [Vau96a]). Another strategy is to generate key-dependent S-boxes from a known secure S-box and a series of strict mathematical rules: e.g., Biham-DES [BB94].

Most key-dependent S-boxes are created by some process completely orthogonal to the underlying cipher. SEAL, for example, uses SHA [NIST93] to create its key-dependent S-boxes. Blowfish uses repeated iterations of itself. The results are S-boxes that are effectively random, but the cost is an enormous performance penalty in key-setup time.[14] An alternative is to build the S-boxes using fairly simple operations from the key. This results in a much faster key setup, but unless the creation algorithm is extensively cryptanalyzed together with the encryption algorithm, unwanted synergies could lead to attacks on the resulting cipher.

To avoid differential (as well as high-order differential, linear, and related-key) attacks, we made the small S-boxes key dependent. It is our belief that while random key-dependent S-boxes can offer acceptable security if used correctly, the benefits of a surjective S-box are worth the additional complexities that constructing them entails. So, to avoid attacks based on non-surjective round functions [BB95, RP95b, RPD97, CWSK98], we made the 8-by-8-bit S-boxes, as well as the 8-by-32-bit S-boxes, bijective.

However, there is really no such thing as a key-dependent S-box. Twofish uses a complex multistage series of S-boxes and round subkeys that are often precomputed as key-dependent S-boxes for efficiency purposes. (For example, see Figure 3 on page 11.) We often used this conceptualization when carrying out our own cryptanalysis against Twofish.

6.5 The Key Schedule

An algorithm's key schedule is the mechanism that distributes key material to the different parts of the cipher that need it, expanding the key material in the process. This is necessary for three reasons:

- There are fewer key bits provided as input to the cipher than are needed by the cipher.
- The key bits used in each round must be unique to the round, in order to avoid "slide" attacks [Wag95b].
- The cipher must be secure against an attacker with partial knowledge or control over some key bits.

When key schedules are poorly designed, they often lead to strange properties of the cipher: large classes of equivalent keys, self-inverse keys, etc. These properties can often aid an attacker in a real-world attack. For example, the DES weak (self-inverse) keys have been exploited in many attacks on larger cryptographic mechanisms built from DES [Knu95a], and the S-1 [Anon95] cipher was broken due to a bad key-schedule design [Wag95a]. Even worse, they can make attacks on the cipher easier, and some attacks on the cipher will be focused directly at the key schedule, such as related-key differential attacks [KSW96, KSW97]. These attacks can be especially devastating when the cipher is used in a hash function construction.

Key schedules can be divided into several broad categories [CDN98]. In some key schedules, knowledge of a round subkey uniquely specifies bits of other round subkeys. In some ciphers the bits are just reused, as in DES, IDEA, and LOKI, and in others some manipulation of the round subkeys is required to determine the other round subkeys: e.g., CAST and SAFER. Other key schedules are designed so that knowledge of one round subkey does not directly specify bits of other round subkeys. Either the round subkey itself is used to generate the other round subkeys in some cryptographically secure manner, as in RC5 and CS-Cipher, or a one-way function is used to generate the sound subkeys (sometimes the block cipher itself): e.g., Blowfish, Serpent [BAK98], ICE [Kwa97], and Shark.

Some simple design principles guided our development of the key schedule for Twofish:

- **Design the Key Schedule for the Cipher.** This is not simply a cryptographic PRNG grafted onto the cipher; the Twofish

[14] For example, setting up a single Blowfish key takes as much time as encrypting 520 blocks, or 4160 bytes, of data.

key schedule is instead an integral part of the whole cipher design.

- **Reuse the Same Primitives.** The Twofish key schedule's subkey generation mechanism, h, is built from the same primitives as the Twofish round function. This allowed us to apply much of the same analysis to both the round function and the subkey generation. This also makes for a relatively simple picture of the cipher and key schedule together. It is reasonable to consider one round's operations, and the derivation of its subkeys, at the same time.

- **Use All Key Bytes the Same Way.** All key material goes through h (or g, which is the same function). That is, the only way a key bit can affect the cipher is after it defines a key-dependent S-box. This allows us to analyze the properties of the key schedule in terms of the properties of the byte permutations.

- **Make It Hard to Attack Both S-box and Subkey Generation.** The key material used to derive the key-dependent S-boxes in g is derived from the key using an RS code having properties similar to those of the MDS matrix. Deriving the key material in this way maximizes the difficulties of an attacker trying to mount any kind of related-key attack on the cipher, by giving him conflicting requirements between controlling the S-box keys and controlling the subkeys.

6.5.1 Performance Issues

For large messages, performance of the key schedule is minor compared to performance of the encryption and decryption functions. For smaller messages, key setup can overwhelm encryption speed. In the design of Twofish, we tried to balance these two items. Our performance criteria included:

- The key schedule must be precomputable for maximal efficiency. This involves trying to minimize the amount of storage required to keep the precomputed key material.

- The key schedule must work "on the fly," deriving each block of subkey material as it is needed, with as little required memory as possible.

- The key schedule must be reasonably efficient for hardware implementations.

- The key schedule must have minimal latency for changing keys.[15]

If performance were not an issue, it would make sense to simply use a one-way hash function to expand the key into the subkeys and S-box entries, as is done in Khufu, Blowfish, and SEAL. However, the AES efficiency requirements make such an approach unacceptable. This led to a much simpler key schedule with much more complicated analysis.

The key schedule design of some other ciphers has led to various undesirable properties. These properties, such as the existence of equivalent keys; DES-style weak, semi-weak, and quasi-weak keys; and DES-style complementation properties, do not necessarily make the cipher weak. However, they tend to make it harder to use the cipher securely. With our key schedule, we can make convincing arguments that none of these properties exists.

7 The Design of Twofish

7.1 The Round Structure

Twofish was designed as a Feistel network, because it is one of the most studied block cipher building blocks. Additionally, use of a Feistel network means that the F function need only be calculated in one direction.[16] This means that we were able to use operations in our F function that are inefficient in the other direction, and make do with tables and constants for one direction only. Contrast this with an SP-network, which must execute its encryption function in both the forward and backward directions.

[15] In its comments on the AES criteria, the NSA suggested that "a goal should be that two blocks could be enciphered with different keys in virtually the same time as two blocks could be enciphered with the same key" [McD97]. The cynical reader would immediately conclude that the NSA is concerned with the efficiency of their brute-force keysearch machines. However, there are implementations where key agility is a valid concern. Key-stretching techniques can always be used to frustrate brute-force attacks [QDD86, KSHW98]. A better defense, of course, is to always use keys too large to make a brute-force search practicable, and to generate them randomly.

[16] In fact, the F function can be non-surjective, as it is in DES or Blowfish.

7.2 The Key-dependent S-boxes

A fundamental component of Twofish is the set of four key-dependent S-boxes. These must have several properties:

- The four different S-boxes need to actually be different, in terms of best differential and linear characteristics and other kinds of analysis.
- Few or no keys may cause the S-boxes used to be "weak," in the sense of having high-probability differential or linear characteristics, or in the sense of having a very simple algebraic representation.
- There should be few or no pairs of keys that define the same S-boxes. That is, changing even one bit of the key used to define an S-box should always lead to a different S-box. In fact, these pairs of keys should lead to extremely different S-boxes.

7.2.1 The Fixed Permutations q_0 and q_1

The construction method for building q_0 and q_1 from 4-bit permutations (specified in Section 4.3.5) was chosen because it decreases hardware and memory costs for some implementations, as discussed previously, without adding any apparent weaknesses to the cipher. It is helpful to recall that these individual fixed-byte permutations are used only to construct the key-dependent S-boxes, which, in turn, are used only within the h and g functions. In particular, the individual characteristics of q_0 and q_1 are not terribly relevant (except perhaps in some related-key attacks), because Twofish always uses at least three of these permutations in series, with at least two XORs with key material bytes. Consideration was initially given to using random full 8-bit permutations for q_0 and q_1, as well as algebraically derived permutations (e.g., multiplicative inverses over GF(2^8)) that have slightly better individual permutation characteristics, but no substantial improvement was found when composite keyed S-boxes were constructed and compared to the q_0 and q_1 used in Twofish.

The q_0 and q_1 permutations were chosen by random search, constructing random 4-bit permutations t_0, t_1, t_2, and t_3 for each. Using the notation of Matsui [Mat96], we define

$$\mathrm{DP}_{\max}(q) = \max_{a \neq 0, b} \Pr_X[q(X \oplus a) \oplus q(X) = b]$$

and

$$\mathrm{LP}_{\max}(q) = \max_{a,b \neq 0} \left(2 \Pr_X[X \cdot a = q(X) \cdot b] - 1\right)^2$$

where q is the mapping $\mathrm{DP}_{\max}$ and $\mathrm{LP}_{\max}$ are being computed for, the probabilities are taken over a uniformly distributed X, and the operator $\cdot$ computes the overall parity of the bitwise-and of its two operands. Only fixed permutations with $\mathrm{DP}_{\max} \leq 10/256$, $\mathrm{LP}_{\max} \leq 1/16$, and fewer than three fixed points were accepted as potential candidates. These criteria alone rejected over 99.8 percent of all randomly chosen permutations of the given construction. Pairs of permutations meeting these criteria were then evaluated as potential (q_0, q_1) pairs, computing various metrics when combined with key material into Twofish's S-box structure, as described below.

The actual q_0 and q_1 chosen were one of several pairs with virtually identical statistics that were found with only a few tens of hours of searching on Pentium class computers. Each permutation has $\mathrm{DP}_{\max} = 10/256$ and $\mathrm{LP}_{\max} = 1/16$; q_0 has one fixed point, while q_1 has two fixed points.

7.2.2 The S-boxes

Each S-box is defined with two, three, or four bytes of key material, depending on the Twofish key size. This is done as follows for 128-bit Twofish keys:

$$\begin{aligned}
s_0(x) &= q_1[q_0[q_0[x] \oplus s_{0,0}] \oplus s_{1,0}] \\
s_1(x) &= q_0[q_0[q_1[x] \oplus s_{0,1}] \oplus s_{1,1}] \\
s_2(x) &= q_1[q_1[q_0[x] \oplus s_{0,2}] \oplus s_{1,2}] \\
s_3(x) &= q_0[q_1[q_1[x] \oplus s_{0,3}] \oplus s_{1,3}]
\end{aligned}$$

where the $s_{i,j}$ are the bytes derived from the key bytes using the RS matrix. Note that with equal key bytes, no pair of these S-boxes is equal. When all $s_{i,j} = 0$, $s_0(x) = q_1[s_1(q_1^{-1}[x])]$. There are other similar relationships between S-boxes. We have not been able to find any weaknesses resulting from this, so long as q_0 and q_1 have no high-probability differential characteristics.

In some sense this construction is similar to a rotor machine citeDK85, with two different types of rotor (q_0 and q_1). The first rotor is fixed, and the fixed offset between two rotors specified by a key byte. We did not find any useful cryptanalysis from this parallel, but someone else might.

For the 128-bit key, we have experimentally verified that each $N/8$-bit key used to define a byte permutation results in a distinct permutation. For example, in the case of a 128-bit key, the S-box s_0 uses 16 bits of key material. Each of the 2^{16} s_0 permutations

defined is distinct, as is also the case for s_1, s_2, and s_3. We have not yet exhaustively tested longer key length, but we conjecture that all S-boxes generated by our construction are distinct. We also conjecture that this would be the case for almost all choices of q_0 and q_1 meeting the basic criteria discussed above.

7.2.3 Exhaustive and Statistical Analysis

Given the fixed permutations q_0 and q_1, and the definitions for how to construct s_0, s_1, s_2, and s_3 from them, we have performed extensive testing of the characteristics of these key-dependent S-boxes. In the 128-bit key case, all testing has been performed exhaustively, which is feasible because each S-box uses only 16 bits of key material. In many cases, however, only statistical (i.e., Monte Carlo) testing has been possible for the larger key sizes. In this section, we present and discuss these results. It is our hope to complete exhaustive testing over time for the larger key sizes where feasible, but the probability distributions obtained for the statistical tests give us a fairly high degree of confidence that no surprises are in store.

Table 4 shows the $\mathrm{DP}_{\max}$ distribution for the various key sizes. A detailed explanation of the format of this table will also help in understanding the other tables in this section. An asterisk (*) next to a key size in the tables discussed in this section indicates that the entries in that row are a statistical distribution using Monte Carlo sampling of key bits, not an exhaustive test. For example, in Table 4, only the 256-bit key row uses statistical sampling; the 128-bit and 192-bit cases involve an exhaustive test. Clearly, the maximum value from a statistical sample is not a guaranteed maximum. Note that, for 128-bit keys, each S-box has a $\mathrm{DP}_{\max}$ value no larger than 18/256. The remaining columns give the distribution of observed $\mathrm{DP}_{\max}$ values. Each entry is expressed as the negative base-2 logarithm of the fraction of S-boxes with the given value (or value range), with blank entries indicating that no value in the range was found. For example, in the $N = 128$ case only 1 of every $2^{12.0}$ key-dependent S-boxes has $\mathrm{DP}_{\max} = 18/256$, while over half (1 in $2^{0.9}$) have $\mathrm{DP}_{\max} = 12/256$. These statistics are taken over all four S-boxes (s_0, s_1, s_2, s_3), so a total of 4×2^{16} (i.e., 256K) S-boxes were evaluated for the 128-bit key case. Each Monte Carlo sampling involves at least 2^{16} S-boxes, but in many cases the number is considerably larger.

Table 5 shows the distribution of $\mathrm{LP}_{\max}$ for the various key sizes. Observe that the vast majority of all Twofish S-boxes have $\mathrm{LP}_{\max} < (88/256)^2$, although there is a small fraction of S-boxes with larger values. For 128-bit keys, no Twofish S-box has an $\mathrm{LP}_{\max}$ value greater than $(100/256)^2$, while the maximum value is somewhat higher for larger key sizes. Monte Carlo statistics are given for the larger two key sizes, since the computational load for computing $\mathrm{LP}_{\max}$ is roughly a factor of fifteen higher than for $\mathrm{DP}_{\max}$.

The distribution of the number of permutation fixed points for various key sizes is given in Table 6. A fixed point of an S-box s_i is a value x for which $x = s_i(x)$. This metric does not necessarily have any direct cryptographic signficance. However, it is a useful way to verify that the S-boxes are behaving similarly to random S-boxes, since it is possible to compute the theoretical distribution of fixed points for random S-boxes. The probabilities for random S-boxes is given in the last row. (The probability of n fixed points is approximately $e^{-1}/n!$.) The Twofish S-box distributions of fixed points over all keys match theory fairly well.

The metrics discussed so far give us a fair level of confidence that the Twofish method of constructing key-dependent S-boxes does a reasonable job of approximating a random set of S-boxes, when viewed individually. Another possible concern is how different the various Twofish S-boxes are from each other, for a given key length. This issue is of particular interest in dealing with related-key attacks, but it also has an important bearing on the ability of an attacker to model the S-boxes. Despite the fact that the Twofish S-boxes are key-dependent, if the entire set (or large subsets) of S-boxes, taken as a black box, are very closely related, the benefit of key-dependency is severely weakened. As a pathological example, consider a fixed 8-bit permutation $q[x]$ which has very good $\mathrm{DP}_{\max}$ and $\mathrm{LP}_{\max}$ values, but which is used as a key-dependent family of S-boxes simply by defining $s_k(x) = q[x] \oplus k$. It is true that each s_k permutation also has good individual metrics, but the class of s permutations is so closely related that conventional differential and linear cryptanalysis techniques can probably be effectively applied without knowing the key k. The Twofish S-box structure has been carefully designed with this issue in mind.

In one class of related-key attacks, an attacker attempts to modify the key bytes in such a way as to minimize the differences in the round subkey values A_i, B_i. Since the Twofish S-box structure is used in computing A_i and B_i, with M_e and M_o as key material, respectively, a measure of the differential

Key Size	Max Value	$-\log_2(\Pr(\mathrm{DP_{max}} = x/256))$								
		$x=8$	$x=10$	$x=12$	$x=14$	$x=16$	$x=18$	$x=20$	$x=22$	$x=24$
128 bits	18/256	15.4	1.3	0.9	4.1	8.0	12.0			
192 bits	24/256	15.2	1.3	0.9	4.1	8.0	12.2	16.5	20.8	25.0
256 bits*	22/256	15.1	1.3	0.9	4.1	8.0	12.2	16.7	22.0	

Table 4: $\mathrm{DP_{max}}$ over all keys

Key Size	Max Value	$-\log_2(\Pr(\mathrm{LP_{max}} = (x/256)^2))$							
		$x=$	56..63	64..71	72..79	80..87	88..95	96..103	104..111
128 bits	$(100/256)^2$		9.3	1.0	1.2	4.2	8.0	12.4	
192 bits*	$(104/256)^2$		9.3	1.0	1.2	4.2	8.2	12.2	17.0
256 bits*	$(108/256)^2$		9.4	1.0	1.2	4.2	8.1	12.5	17.4

Table 5: $\mathrm{LP_{max}}$ over all keys.

Key Size	Max Value	$-\log_2(\Pr(\#\text{ fixed points} = x))$										
		$x=0$	$x=1$	$x=2$	$x=3$	$x=4$	$x=5$	$x=6$	$x=7$	$x=8$	$x=9$	$x=10$
128 bits	8	1.4	1.4	2.4	4.1	6.0	8.2	11.1	14.1	17.0		
192 bits	10	1.4	1.4	2.4	4.0	6.0	8.4	10.9	13.8	16.8	19.8	23.4
256 bits*	10	1.4	1.4	2.4	4.0	6.0	8.4	10.9	13.7	16.8	21.0	21.0
random		1.4	1.4	2.4	4.0	6.0	8.3	10.9	13.7	16.7	19.9	23.2

Table 6: Number of fixed points over all keys

characteristics of A_i (or B_i) across keys will help us understand both how different the S-boxes are from each other and how likely such an attack is to succeed.

To this end, let us first look at how many consecutive values of A_i with a fixed XOR difference can be generated for two different keys. Without loss of generality, we consider only A_i, and in particular we consider a change in only the key material M_e that affects one of the four S-boxes. Let y_i be the output sequence for one S-box used in generating the sequence A_i, and the sequence for another key be y'_i. Consider the difference sequence $y^*_i = y_i \oplus y'_i$. For example, in the 128-bit key case, with 16 bits of key material per S-box, there are about 2^{31} pairs of distinct keys for each S-box, so there would be 2^{31} such difference sequences, each of length 20. What is the probability of having a "run" of n consecutive equal values in the sequence? If n can be made to approach 20, then a related-key attack might be able to control the entire sequence of A_i values, and, even worse, our belief in the independence of the key-dependent S-boxes must be called seriously into question. Note that an attacker has exactly 16 bits of freedom for a single S-box in the 128-bit key case, so intuitively it seems unlikely that he should be able to force a given difference sequence that is very long.

Table 7 shows the distribution of run lengths of the same XOR difference y^*_i for consecutive i. For random byte sequences, we would theoretically expect that $\Pr(\text{XOR run length} = n)$ should be roughly $2^{-8(n-1)}$, which matches quite nicely with the behavior observed in the table. It can be seen that the probability of injecting a constant difference into more than five consecutive subkey entries is extremely low, which is reassuring.

Table 8 shows the results of measuring this same property in a different way. Instead of requiring a run of n consecutive identical differences, this metric is similar to a "mini"-$\mathrm{DP_{max}}$, computed over only the 20 input values used in the subkey generation (e.g., $0, 2, 4, \ldots, 38$). The quantity measured is the maximum number of equal differences out of the 20 values generated, across key pairs. In other words, while it may be difficult to generate a large run, is it possible to generate equal values in a large fraction of the set of differences? The results are again very encouraging, showing that it is extremely difficult to force more than five or six differences to be identical. This also shows that it is not possible to influence only a few A_i, as that would require many zero differences.

Key Size	Max Value	$-\log_2(\Pr(\text{XOR run length} = n))$				
		$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
128 bits	5	0.01	7.98	15.95	23.89	30.00
192 bits*	4	0.01	7.98	15.93	23.69	
256 bits*	5	0.01	7.98	15.93	24.02	30.29

Table 7: Subkey XOR difference run lengths

Key Size	Max Value	$-\log_2(\Pr(\text{max \# equal XOR differences} = n))$						
		$x=1$	$x=2$	$x=3$	$x=4$	$x=4$	$x=6$	$x=7$
128 bits	7	1.1	0.9	5.9	11.6	17.6	23.9	31.0
192 bits*	7	1.1	0.9	5.9	11.7	17.8	23.9	29.0
256 bits*	7	1.1	0.9	5.9	11.7	17.7	23.8	29.3

Table 8: "Mini"-$\text{DP}_{\max}$ subkey distribution

For every metric discussed here, a similar distribution using randomly generated 8-bit permutations has been generated for purposes of comparison. For example, $\text{DP}_{\max}$ was computed for a set of 2^{16} randomly generated permutations, and the resulting distribution of $\text{DP}_{\max}$ values compared to that of the Twofish key-dependent S-boxes. For each metric, the probability distributions looked virtually identical to those obtained for the Twofish set of key-dependent S-boxes, except for small fluctuations on the tail ends of the distribution, as should be expected. This similarity is comforting, as is the fact that the probability distributions for each metric look quite similar across key sizes. These results help confirm our belief that, from a statistical standpoint, the Twofish S-box sets behave largely like a randomly chosen set of permutations.

7.3 MDS Matrix

The four bytes output from the four S-boxes are multiplied by a 4-by-4 MDS matrix over GF(2^8). This matrix multiply is the principal diffusion mechanism in Twofish. The MDS property here guarantees that the number of changed input bytes plus the number of changed output bytes is at least five. In other words, any change in a single input byte is guaranteed to change all four output bytes, any change in any two input bytes is guaranteed to change at least three output bytes, etc. More than 2^{127} such MDS matrices exist, but the Twofish MDS matrix is also carefully chosen with the property of preserving the number of bytes changed even after the rotation in the round function.

The MDS matrix used in Twofish has fixed coefficients. Initially, some thought was given to making the matrix itself key-dependent, but such a scheme would require the verification that the key-dependent values in fact formed an MDS matrix, adding a non-trivial computational load to the key selection and key scheduling process. However, it should be noted that there are many acceptable MDS matrices, even with the extended properties discussed below.

For software implementation on a modern microprocessor, the MDS matrix multiply is normally implemented using four lookup tables, each consisting of 256 32-bit words, so the particular coefficients used in the matrix do not affect performance. However, for smart cards and in hardware, "simple" coefficients, as in Square [DKR97], can make implementations cheaper and faster. Unlike the MDS matrix used in Square, Twofish does not use the inverse matrix for decryption because of its Feistel structure, nor is there a requirement that the matrix be a circulant matrix.

However, because of the rotation applied after the XOR within the round function, it is desirable to select the MDS matrix carefully to preserve the diffusion properties even after the rotation. For both encryption and decryption, a right rotation by one bit occurs after the XOR. This direction of rotation is chosen to preserve the MDS property with respect to the PHT addition $a+2b$, since the rotation undoes the shift applied to b as part of the PHT. It is true that the most significant bit of b is still "lost" in this half of the PHT, but the MDS properties for three of the bytes are still fully guaranteed with respect to b, and they are met with probability 254/255 for the fourth byte.

The effect of the rotation on the unshifted PHT additions also needs to be addressed. A single byte input change to the MDS matrix will change all four output bytes, which affect the round function output, but after rotation there is no such guarantee. If a byte value difference of 1 is one output from the matrix multiply, performing a 32-bit rotate on the result will shift in one bit from the next byte in the 32-bit word and shift out the only non-zero bit. The MDS matrix coefficients in Twofish are carefully selected so that, if a byte difference 1 is output from the matrix multiply with a single byte input change, the next highest byte is guaranteed to have its least significant bit changed too. Thus, if the rotation shifts out the only original flipped bit, it will also shift in a flipped bit from the next byte.

The construction used to preserve this property after rotation is actually very simple. The idea is to choose a small set of non-zero elements of GF(2^8) with the property that, for each pair x, y in the set, if $x * a = 1$, then $y * a$ (i.e., y/x) has the least significant bit set. Observe that this property is reflexive; i.e., if $x = y$, then $y/x = 1$, so the property holds. It is intuitively obvious (and empirically verified) that the probability that two random field elements satisfy this property is roughly one half, so it should not be difficult to find such sets of elements. Also, since over 75% of all 4-by-4 matrices over GF(2^8) are MDS, the hope of finding such a matrix sounds reasonable.

A computer search was executed over all primitive polynomials of degree eight, looking for sets of three "simple" elements x,y,z with the above property. Simple in this context means that $x * a$ for arbitrary a can be easily computed using at most a few shifts and XORs. Several dozen sets of suitable values were found, each of which allowed several MDS matrixes with the three values. The primitive polynomial $v(x) = x^8 + x^6 + x^5 + x^3 + x^0$ was selected, together with the field elements 1, EF, and 5B (using hexadecial notation and the field element to byte value correspondence of Section 4). The element EF is actually $\beta^{-2} + \beta^{-1} + 1$, where β is a root of $v(x)$, and 5B is $\beta^{-2} + 1$, so multiplication by these elements consists of two LFSR right shifts mod $v(x)$, plus a few byte XORs.

Another constraint imposed on the Twofish MDS matrix is that no row (or column) of the matrix be a rotation of another row (or column) of the matrix. This property guarantees that all single-byte input differences result in unique output differences, even when rotations over 8 bits are applied to the output, as is done in generating the round subkeys. This constraint did not seem to limit the pool of candidate matrices significantly. In fact, the Twofish MDS matrix actually exhibits a much stronger property: all 1020 MDS output differences for single-byte input changes are distinct from all others, even under bitwise rotations by each of the rotation values in the range 6..26. The subkey generation routine takes advantage of this property to help thwart related-key differential attacks. For all single byte input changes, the rotation of the output 32-bit word B by eight bits has an output difference that is guaranteed to be unique from output differences in the unrotated A quantity.

There are many MDS matrices containing only the three elements 1, EF, and 5B. The particular Twofish matrix was also chosen to maximize the minimum binary Hamming weight of the output differences over all single-byte input differences. This Twofish MDS matrix guarantees that any single-byte input change will produce an output Hamming difference of at least eight bits, in addition to the property that all four output bytes will be affected. In fact, as shown in the table below, only 7 of the 1020 possible single-byte input differences result in output differences with Hamming weight difference eight; the remaining 1013 differences result in higher Hamming weights. Also, only one of the seven outputs with Hamming weight 8 has its most significant bit set, meaning that at least eight bits will be affected even in the PHT term $T_0 + 2T_1$ for 1019 of the 1020 possible single-byte input differences. Input differences in two bytes are guaranteed to affect three output bytes, and they can result in an output Hamming difference of only three bits, with probability 0.000018; the probability that the output difference Hamming weight is less than eight bits for two byte input differences is only 0.00125, which is less than the corresponding probability (0.0035) for a totally random binomial distribution.

Hamming Weight	Number of occurrences	Number with MSB Set
8	7/1020	1
9	23/1020	4
10	42/1020	15

There are many other MDS matrices, using either the same or another set of simple field elements, that can guarantee the same minimum output Hamming difference, and the particular matrix chosen is representative of the class. No higher minimum Hamming weight was found for any matrices with such simple elements.

It is fairly obvious that this matrix multiply can be implemented at high speed with minimal cost, both in hardware and in firmware on a smart card.

It should also be noted that a very different type of construction could be used to preserve MDS properties on rotation. Use of an 8-by-8 MDS matrix over GF(2^4) will guarantee eight output 4-bit nibble changes for every input nibble change. Because the changes now are nibble-based, a one-bit rotation may shift the only non-zero bit of a nibble out of a byte, but the other nibble remains entirely contained in the byte. In fact, it can easily be seen that this construction preserves the MDS property nicely even for multi-bit rotations. Unfortunately, 8-by-8 MDS matrices over GF(2^4) are nowhere nearly as plentiful as 4-by-4 matrices over GF(2^8), so very little freedom is available to pick simple coefficients. The best construction for such a matrix seems to be an extended RS-(15,7) code over GF(2^4), which requires considerably more gates in hardware and more tables in smart card firmware than the Twofish matrix. Because of this additional cost, we decided not to use an 8-by-8 MDS matrix over GF(2^4).

The actual transformation defined by the MDS matrix is purely linear over GF(2). That is, each of the output bits is the XOR of a subset of the input bits.

7.4 PHT

The PHT operation, including the addition of the round subkeys, was chosen to facilitate very fast operation on the Pentium CPU family using the LEA (load effective address) opcodes. The LEA opcodes allow the addition of one register to a shifted (by 1,2,4,8) version of another register, along with a 32-bit constant, all in a single clock cycle, with the result placed in an arbitrary Pentium register. For best performance, a version of the encryption and decryption code can be "compiled" for a given key, with the round subkeys inserted as constant values in LEA opcodes in the instruction stream. This approach requires a full instantiation in memory of the code for each key in use, but it provides a speedup for bulk encryption.

Instead of using 4 key-dependent S-boxes, a 4-by-4 MDS matrix, and the PHT, we could have used 8 key-dependent S-boxes and an 8-by-8 MDS matrix over GF(2^8). Such a construction would be easier to analyse and would have nicer properties, but it is much slower in virtually all implementations and would not be worth it.

7.5 Key Addition

As noted in the previous section, the round subkeys are combined with the PHT output via addition to enable optimal performance on the Pentium CPU family. From a cryptographic standpoint, an XOR operation could have been used, but it would reduce the best Pentium software performance for bulk encryption. It should be noted that using addition instead of XOR does impose a minor gate count and speed penalty in hardware, but this additional overhead was considered to be well worth the extra performance in software. On a smart card, using addition instead of XOR has virtually no impact on code size or speed.

7.6 Feistel Combining Operation

Twofish uses XOR to combine the output of F with the target block. This is done primarily for simplicity; XOR is the most efficient operation in both hardware and software. We chose not to use addition (used in MD4 [Riv91], MD5 [Riv92], RIPEMD-160 [DBP96] and SHA [NIST93]), or a more complicated combining function like Latin squares (used in DESV [CDN95]). We did not implement dynamic swapping [KKT94] or any additional complexity.

7.7 Use of Different Groups

By design, the general ordering of operations in Twofish alternates as follows: 8-by-8 S-box, MDS matrix, addition, and XOR. The underlying algebraic operations thus alternate between non-linear table lookup, a GF(2)-linear combination of the bits by the MDS matrix, integer addition mod 2^{32}, and GF(2) addition (XOR). Within the S-boxes, several levels of alternating XOR and 8-by-8 permutations are applied. The goal of this ordering is to help destroy any hope of using a single algebraic structure as the basis of an attack. No two consecutive operations use the same structure, except for the PHT and the key addition that are designed to be merged for faster implementations.

7.8 Diffusion in the Round Function

There are two major mechanisms for providing diffusion in the round function. The first is the MDS matrix multiply, which ensures that each output byte depends on all input bytes. The two outputs of the g functions (T_0 and T_1) are then combined using a PHT so that both of them will affect both 32-bit

Feistel XOR quantities. The half of the PHT involving the quantity $T_0 + 2T_1$ will lose the most significant bit of T_1 due to the multiply by two. This bit could be regained using some extra operations, but the software performance would be significantly decreased, with very little apparent cryptographic benefit. In general, the most significant byte of this PHT output will still have a non-zero output difference with probability 254/255 over all 1-byte input differences.

We cannot guarantee that a single byte input change to the F function will change 7 or 8 of the output bytes of F. The reason is that the carries in the addition of the PHT can remove certain byte differences. For example, an addition with constant might turn a difference of 00000180_{16} into 00000080_{16}. The chances of this happening depend on the distance between the two bits that influence each other. A large reduction in the number of changed bytes is very unlikely.

7.9 One-bit Rotation

Within each round, both of the 32-bit words that are XORed with the round function results are also rotated by a single bit. One word is rotated before the XOR, and one after the XOR. This structure provides symmetry for decryption in the sense that the same software pipeline structure can be applied in either direction. By rotating a single bit per round, each 32-bit quantity is used as an input to the round function once in each of the eight possible bit alignments within the byte.

These rotations in the Twofish round functions were included specifically to help break up the byte-aligned nature of the S-box and MDS matrix operations, which we feared might otherwise permit attacks using statistics based on strict byte alignment. For example, Square uses an 8-by-8-bit permutation and an MDS matrix in a fashion fairly similar to Twofish, but without any rotations. An early draft of the Square paper proposed a very simple and powerful attack, based solely on such byte statistics, that forced the authors to increase the number of rounds from six to eight. Also, an attack against SAFER is based on the cipher's reliance on a byte structure [Knu95c].

Choosing a rotation by an odd number of bits ensures that each of the four 32-bit words are used as input to the g function in each of the eight possible bit positions within a byte. Rotating by only one bit position helps optimize performance on the Pentium CPU (which, unlike the Pentium Pro, has a one-clock penalty for multi-bit rotations) and on smart card CPUs (which generally do not have multi-bit rotate opcodes). Limiting rotations to single-bit also helps minimize hardware costs, since the wiring overhead of fixed multi-bit rotations is not negligible.

There are three downsides to the rotations in Twofish. First, there is a minor performance impact (less than 7% on the Pentium) in software due to the extra rotate opcodes. Second, the rotations make the cipher non-symmetric in that the encryption and decryption algorithms are slightly different, thus requiring distinct code for encryption and decryption. It is only the rotations that separate Twofish from having a "reversible" Feistel structure. Third, the rotates make it harder to analyze the cipher for security against differential and linear attacks. In particular, they make the simple technique of merely counting active S-boxes quite a bit more complicated. On the other hand, it is much harder for the attacker to analyze the cipher. For instance, it is much harder to find iterative characteristics, since the bits do not line up. The rotates also make it harder to use the same high-probability characteristic several times as the bits get rotated out of place. On the whole, the advantages were considered to outweigh the disadvantages.

It is possible to convert Twofish to a pure Feistel structure by incorporating round-dependent rotation amounts in F, and adding some fixed rotations just before the output whitening. This might be a useful view of the cipher for analysis purposes, but we do not expect any implementation to use such a structure.

The rotations were also very carefully selected to work together with the PHT and the MDS matrix to preserve the MDS difference properties for single byte input differences to g. In particular, for both encryption and decryption, the one-bit right rotation occurs after the Feistel XOR with the PHT output. The MDS matrix was chosen to guarantee that a 32-bit word rotate right by one bit will preserve the fact that all four bytes of the 32-bit word are changed for all single input byte differences. Thus, placing the right rotation after the XOR preserves this property. However, during decryption, the rotate right is done after the XOR with the Feistel quantity involving $T_0 + 2T_1$. Note that, in this case, the rotate right puts the $2T_1$ quantity back on its byte boundary, except that the most significant bit has been lost. Therefore, given a single input byte difference that affects only T_1, the least significant three bytes of the Feistel XOR output are guaranteed to change

after the rotation, and the most significant byte will change with probability 254/255.

The fact that the rotate left by one bit occurs before the Feistel XORs during encryption guarantees that the same relative ordering (i.e., rotate right after XOR) occurs during decryption, preserving the difference-property for both directions. Also, performing one rotation before and one after the Feistel XOR imposes a symmetry between encryption and decryption that helps guarantee very similar software performance for both operations on the Pentium CPU family, which has only one ALU pipeline capable of performing rotations.

7.10 The Number of Rounds

Sixteen rounds corresponds to 8 cycles, which seems to be the norm for block ciphers. DES, IDEA, and Skipjack all have 8 cycles. Twofish was defined to have 16 rounds primarily out of pessimism. Although our best non-related-key attack only breaks 5 rounds of the cipher, we cannot be sure that undiscovered cryptanalysis techniques do not exist that can do better. Hence, we consider 16 rounds to be a good balance between our natural skepticism and our desire to optimize performance.

Even so, we took pains to ensure that the Twofish key schedule works with a variable number of rounds. It is easy to define Twofish variants with more or fewer rounds.

7.11 The Key Schedule

To understand the design of the key schedule, it is necessary to consider how key material is used in Twofish:

The Whitening Subkeys 128 bits of key material are XORed into the plaintext block before encryption, and another 128 bits after encryption. Since the rest of the encryption is a permutation, this can be seen as selecting among as many as 2^{256} different (but closely related) 128-bit to 128-bit permutations. This key material has the effect of making many cryptanalytic attacks a little more difficult, at a very low cost. Note that nearly all the added strength against cryptanalytic attack is added by the XOR of subkeys into the input to the first and last rounds' F functions.

The Round Subkeys 64 bits of key material are combined into the output of each round's F function using addition modulo 2^{32}. The F function without the round subkey addition is a permutation on 64-bit values; the round subkey selects among one of 2^{64} closely related permutations in each round. These subkeys must be slightly different per round, to prevent a slide attack, as will be discussed below.

The Key-dependent S-boxes To create the S-boxes, the key is mapped down to a block of data half its size, and that block of data is used to specify the S-boxes for the whole cipher. As discussed before, the key-dependent S-boxes are derived by alternating fixed S-box lookups with XORs of key material.

7.11.1 Equivalence of Subkeys

In this section, we discuss whether different sequences of subkeys can give equivalent encryption functions. Recall that F' is the F function without the addition of the round subkeys. For this analysis we keep the function F' constant. That is, we only vary the subkeys K_i and not S. The properties of a Feistel cipher ensure that no pair of 2-round subkey sequences can be equivalent for all inputs. It is natural to ask next whether any pairs of three sequential rounds' subkeys can exist that cause exactly the same encryption.

For a pair of subkey sequences, (k_0, k_1, k_2) and (k_0^*, k_1^*, k_2^*), to be equivalent in their effects, every input block (L_0, R_0) must encrypt to the same output block (L_1, R_2) for both sequences of subkeys. Note that $k_0 \neq k_0^*$ and $k_2 \neq k_2^*$; as we would otherwise have two sequences of 2-round keys that would define the same 2-round encryption function. We have the following equalities

$$\begin{aligned} R_1 &= R_0 \oplus (k_0 + F'(L_0)) \\ R_1^* &= R_0 \oplus (k_0^* + F'(L_0)) \\ L_1 &= L_0 \oplus (k_1 + F'(R_1)) \\ L_1 &= L_0 \oplus (k_1^* + F'(R_1^*)) \\ R_2 &= R_1 \oplus (k_2 + F'(L_1)) \\ R_2 &= R_1^* \oplus (k_2^* + F'(L_1)) \end{aligned}$$

where $\oplus$ represents bitwise XORing, and $+$ represents 32-bit word-wise addition. Using the two equations for L_1 we get

$$\begin{aligned} k_1 + F'(R_1) &= k_1^* + F'(R_1^*) \\ \delta_1 &= F'(R_1) - F'(R_1^*) \end{aligned}$$

where $\delta_1 = k_1^* - k_1$ is fixed. Let $T = F'(L_0) + k_0$ and observe that when L_0 goes over all possible values, so does T. We get

$$\delta_1 = F'(R_0 \oplus T) - F'(R_0 \oplus (T + \delta_0)) \qquad (1)$$

where $\delta_0 = k_0^* - k_0$. Note that δ_1 and δ_0 depend only on the round keys, and that the equation must hold for all values of R_0 and T. Set $T = 0$ and look at the cases $R_0 = 0$ and $R_0 = \delta_0$. We get

$$F'(0) - F'(\delta_0) = \delta_1 = F'(\delta_0) - F'(0) = -\delta_1$$

The substraction here is modulo 2^{32} for each of the two 32-bit words. That leaves us with the following possible values for δ_1:

$$\delta_1 \in \{(0,0), (0,2^{31}), (2^{31},0), (2^{31},2^{31})\}$$

These are the possible difference values at the output of F' in equation 1. We can easily convert them to difference values at the input of the PHT of F'. Each of the possible values for δ_1 corresponds to exactly one possible value for $(\delta_{T_0}, \delta_{T_1})$:

$$(\delta_{T_0}, \delta_{T_1}) \in \{(0,0), (0,2^{31}), (2^{31},0), (2^{31},2^{31})\}$$

We can write down the analogue to equation 1 for g:

$$\delta_{T_0} = g(R' \oplus T') - g(R' \oplus (T' + \delta_0'))$$

for all R' and T' and where δ_0' is the appropriate half of δ_0. Observe that for the specific values of δ_{T_0} that are possible, subtraction and XOR are the same. For $T' = 0$ this translates in a simple differential equation

$$\delta_{T_0} = g(R') \oplus g(R' \oplus \delta_0')$$

for all R'. We know that g has only one perfect differential: $0 \mapsto 0$, so we conclude that $\delta_0' = 0$. Similarly, we can conclude that the other half of δ_0 must also be zero, and thus $\delta_0 = 0$. This is a contradiction, as $k_0 \neq k_0^*$.

We conclude that there are no two sets of 3-round subkey sequences that result in the same encryption function.

A Conjecture About Equivalent Subkeys in Twofish We believe that, for up to 16 rounds of Twofish, there are no pairs of equivalent round keys, where equivalent round keys lead to the same encryption/decryption function for all possible inputs. There simply do not appear to be enough degrees of freedom in choosing the different subkeys to make pairs of equivalent subkey sequences in as few as 16 rounds. However, we have been unable to prove this.

We also conjecture that there are no pairs of Twofish keys that lead to identical encryptions for all inputs. (Recall that the Twofish keys are used to derive both subkey sequences and also S-boxes.) The function g depends only on half the key entropy, but within that restriction we have verified that no two keys lead to the same function g. This property does not guarantee that the round function F has no key-equivalences (since other key material is added into the outputs of the PHT), but it provides partial heuristic evidence for that contention.

7.11.2 Byte Sequences

The subkeys in Twofish are generated by using the h function, which can be seen as four key-dependent S-boxes followed by an MDS matrix. The input to the S-boxes is basically a counter. In this section we analyze the sequences of outputs that this construction can generate.

All key material is used to define key-dependent S-boxes in h, which are then used to derive subkeys. Each S-box gets a sequence of inputs, $(0, 2, 4, \ldots, 38)$ or $(1, 3, 5, \ldots, 39)$. The S-box generates a corresponding sequence of outputs. The corresponding outputs from the four S-boxes are combined using the MDS matrix multiply to produce the sequence of A_i and B_i words, and those words are processed with the PHT (with a couple of rotations thrown in) to produce a pair of subkey words. Analyzing these byte sequences thus gives us important insights about the whole key schedule.

We can model each byte sequence generated by a key-dependent S-box as a randomly selected non-repeating byte sequence of length 20. This allows us to make many useful predictions about the likelihood of finding keys or pairs of keys with various interesting properties. Because we will be analyzing the key schedule using this assumption in the remainder of this section, we should discuss how reasonable it is to treat this byte sequence as randomly generated. As discussed in Section 7.2.3 we have not found any statistical deviations between our key-dependent S-boxes and the random model in any of our extensive statistical tests.

We are looking at 20-byte-long sequences of distinct bytes. There are 256!/236! of those sequences, which is close to 2^{159}.

7.11.3 Equivalent S-box Keys

We have verified that there are no equivalent S-box keys that generate the same sequence of 20 bytes. In the random model, the chance of this happening for the $N = 256$ case is about $2^{63} \cdot 2^{-159} = 2^{-96}$. This is the chance of such equivalent S-boxes existing at all.

In fact, if we peel off one layer of our q construction and assume the rest of the construction is random,

we can improve that bound. Without loss of generality we look at[17]:

$$s_0(x) = q_0[q_0[q_1[q_1[q_0[x] \oplus k_0] \oplus k_1] \oplus k_2] \oplus k_3]$$

Identical sequences are possible only if the inputs to that last q_0 fixed permutation are identical for both sequences. That means that the task of finding a pair of identical sequences comes down to a simple task: finding a pair of (k_0, k_1, k_2) byte values that leads to a pair of sequences before the XOR with k_3 that have a fixed XOR difference. Then, k_3 can be changed to include that XOR difference, and identical sequences of inputs will go into that last q_0 S-box.

Let

$$t[i] := q_0[q_1[q_1[q_0[i] \oplus k_0] \oplus k_1] \oplus k_2]$$

The goal is to find a pair of $t[i]$ sequences such that

$$t[i] \oplus t^*[i] = \text{constant}$$

Let us assume that t generates a random sequence. The chances of any pair of t, t^* generating such a constant difference is about 2^{-151}. This brings the chance of finding a pair with such a constant difference down to $2^{47} \cdot 2^{-151} = 2^{-104}$.

7.11.4 Byte Difference Sequences

Let us consider the more general problem of how to get a given 20-byte difference sequence between a pair of S-boxes. Suppose we have two S-boxes, each defined using 32 bits of key material, which are not equal, but which must be chosen to give us a given difference sequence in the XOR of their byte sequences. We can estimate the probability of a pair of 4-byte inputs existing with the desired XOR difference sequence as $2^{63} \cdot 2^{-159} = 2^{-96}$. Note that this is the probability that such a pair of inputs exists, not the probability that a random pair of keys will have this property.

7.11.5 The A and B Sequences

From the properties of the byte sequences, we can discuss the properties of the A and B sequences generated by each key M.

$$A_i = \text{MDS}(s_0(i, M), s_1(i, M), s_2(i, M), s_3(i, M))$$

Since the MDS matrix multiply is invertible, and since i is different for each round's subkey words generated, we can see that no A or B value can repeat itself.

Similarly, we can see from the construction of h that each key byte affects exactly one S-box used to generate A or B. Changing a single key byte always alters every one of the 20 bytes of output from that S-box, and so always alters every word in the 20-word A or B sequence to which it contributes.

Consider a single byte of output from one of the S-boxes. If we cycle any one of the key bytes that contributes to that S-box through all 256 possible values, the output of the S-box will also cycle through all 256 possible values. If we take four key bytes that contribute to four different S-boxes, and we cycle those four bytes through all possible values, then the result of h will also cycle through all possible values. This proves that A and B are uniformly distributed for all key lengths, assuming the key M is uniformly distributed.

7.11.6 Difference Sequences in A and B

Let us also consider difference sequences. If we have a specific difference sequence we want to see in A, we are faced with an interesting problem: since the MDS matrix multiply is XOR-linear, each desired output XOR from the matrix multiply allows only one possible input XOR. This means that:

1. A zero output XOR difference in A can occur *only* with a zero output XOR difference in all four of the byte sequences used to build A.

2. Only 1020 possible output differences (out of the 2^{32}) in A_i can occur with a single "active" (altered) S-box. Most differences require all four S-boxes used to form A_i to be active.

3. Each desired output XOR in A requires a specific output XOR in each of the four byte sequences used to form A. This means that getting any desired difference sequence into all 20 A_i values requires getting a desired XOR sequence into all four 20-byte sequences. (Note that if the desired output XOR in A_i is an appropriate value, up to three of the four byte sequences can be identical without much trouble, simply by leaving their key material unchanged.) As mentioned above, this is very unlikely to be possible for a randomly chosen difference pattern in the A sequence. (There are of course difference sequences of A_i's that can occur.)

[17] For ease of discussion, we number the key bytes as $k_0..k_3$ going into one S-box. The actual byte ordering for s_0 and a 256-bit key's subkey-generating S-box is k_0, k_8, k_{16}, k_{24}. The numbering of the key bytes has no effect on the security arguments in this section.

The above analysis is of course also valid for the B sequence.

7.11.7 The Sequence (K_{2i}, K_{2i+1})

As A_i and B_i are uniformly distributed (over all keys), so are all the K_i. As all pairs (A_i, B_i) are distinct, all the pairs (K_{2i}, K_{2i+1}) are distinct, although it might happen that $K_i = K_j$ for any pair of i and j.

7.11.8 Difference Sequences in the Subkeys

Difference sequences in A and B translate into difference sequences in (K_{2i}, K_{2i+1}). However, while it is natural to consider A and B difference sequences in terms of XOR differences, subkeys can reasonably be considered either as XOR differences or as differences modulo 2^{32}. Thus, we may discuss difference sequences:

$$\begin{aligned} D[i, M, M^*] &= K_{i,M} - K_{i,M^*} \\ X[i, M, M^*] &= K_{i,M} \oplus K_{i,M^*} \end{aligned}$$

where the difference is computed between the key value M and M^*.

7.11.9 XOR Differences in the Subkeys

Each round, the subkeys are added to the results of the PHT of two g functions, and the results of those additions are XORed into half of the cipher block. An XOR difference in the subkeys has a fairly high probability of passing through the addition operation and ending up in the cipher block. (The probability of this is determined by the Hamming weight of the XOR difference, not counting the highest-order bit.) However, to get into the subkeys, a XOR difference must first pass through the first addition.

Consider

$$\begin{aligned} x + y &= z \\ (x \oplus \delta_0) + y &= z \oplus \delta_1 \end{aligned}$$

Let k be the number of bits set in δ_0, not counting the highest-order bit. Then, the highest probability value for δ_1 is δ_0, and the probability that this will hold is 2^{-k}. This is true because addition and XOR are very closely related operations. The only difference between the two is the carry between bit positions. If flipping a given bit changes the carry into the next bit position, this alters the output XOR difference. This happens with probability 1/2 per bit. The situation is more complex for multiple adjacent bits, but the general rule still holds: for every bit in the XOR difference not in the high-order bit position, the probability that the difference will pass through correctly is cut in half.

For the subkey generation, consider an XOR difference, δ_0, in A. This affects two subkey words:

$$\begin{aligned} K_{2i} &= A_i + B_i \\ K_{2i+1} &= \mathrm{ROL}(A_i + 2B_i, 9) \end{aligned}$$

where the additions are modulo 2^{32}. If we assume these XOR differences propagate independently in the two subkeys (which appears to be the case), we see that this leads to an XOR difference of δ_0 in the even subkey word with probability 2^{-k}, and the XOR difference $ROL(\delta_0, 9)$ in the odd subkey with the same probability. The most probable XOR difference in the round's subkey block thus occurs with probabiity 2^{-2k}. A desired XOR difference sequence for all 20 pairs of subkey words is thus quite difficult to get to work when $k \geq 3$, assuming the desired XOR difference sequence can be created in the A sequence at all.

When the XOR difference is in B, the result is slightly more complicated; the most probable XOR difference in a round's pair of subkey words may be either $2^{-(2k-1)}$ or 2^{-2k}, depending on whether or not the XOR difference in B covers the next-to-highest-order bit.

7.11.10 Differences in the Subkeys

An XOR difference in A or B is easy to analyze in terms of additive differences modulo 2^{32}: an XOR difference with k active bits has 2^k equally likely additive differences. Note that if we have a additive difference in A, we get it in both subkey words, just rotated left nine bits in the odd subkey word. Thus, k-bit XOR differences lead to a given additive difference in a pair of subkey words with probability 2^{-k}. (The rotation does not really complicate things much for the attacker, who knows where the changed bits are.)

Note that when additive subkey differences modulo 2^{32} are used in an attack, they survive badly through the XOR with the plaintext block. We estimate that XOR differences are much more likely to be directly useful in mounting an attack.

7.11.11 Properties of the Key Schedule and Cipher

One NIST requirement is that the AES candidates have no weak keys. Here we argue that Twofish has

none.

Equivalent Keys A pair of equivalent keys, M, M^*, is a pair of keys that encrypt all plaintexts into the same ciphertexts. We are almost certain that there are no equivalent keys in Twofish. There is no pair of keys, M, M^*, that gives the same subkey sequence. This is easy to see; to get identical subkeys, we have to get an identical A and B sequence at the same time, which requires getting all eight of the key scheduling S-boxes to give the same output for the same input for two different keys. We have verified that this cannot be done.

It is conceivable that different sequences of subkeys and different S-boxes in g could end up producing the same encryption function, thus giving equivalent keys. This seems extremely unlikely to us, but we cannot prove that such equivalent keys do not exist.

Self-Inverse Keys Self-inverse keys are keys for which encrypting a block of data twice with the same key gives back the original data. We do not believe that self-inverse keys exist for Twofish. Keys cannot generate a self-inverse sequence of subkeys, because the same round subkey value can never appear more than once in the cipher. Again, it is conceivable that some keys are self-inverse despite using different subkeys at different points, but it is extremely unlikely.

Pairs of Inverse Keys A pair of inverse keys is a pair of keys M_0, M_1, such that $E_{M_0}(E_{M_1}(X)) = X$, for all X. Pairs of these keys that have identical subkeys, just running backwards, are astronomically unlikely to exist at all. For this to work, each S-box must have a pair of keys that reverses it. This is a kind of collision property; for each S-box, it has probability 2^{-96} of existing at all.

Simple Relations A key complementation property exists when:

$$E_M(P) = C \quad \Rightarrow \quad E_{M'}(P') = C'$$

where P', C', and K' are the bitwise complement of P, C, and K, respectively. No such property has been observed for Twofish.

More generally, a simple relation [Knu94b] is defined as:

$$E_M(P) = C \quad \Rightarrow \quad E_{f(M)}(g(P, M)) = h(C, M)$$

where f, g, and h are simple functions. We have found no simple relations for Twofish, and strongly doubt that they exist.

7.11.12 Key-dependent Characteristics and Weak Keys

The concept of a key-dependent characteristic seems to have been introduced in [BB93] in their cryptanalysis of Lucifer, and also appears in [DGV94a] in an analysis of IDEA.[18] The idea is that certain iterative properties of the block cipher useful to an attacker become more effective against the cipher for a specific subset of keys.

A differential attack on Twofish may consider XOR-based differences, additive differences, or both. If an attacker sends XOR differences through the PHT and subkey addition steps, his differential characteristic probabilities will be dependent on the subkey values involved. In general, low-weight subkeys will give an attacker some advantage, but this advantage is relatively small. (Zero bits in the subkeys improve the probabilities of cleanly getting XOR-based differential characteristics through the subkey addition.) Since there appears to be no special way to choose the key to make the subkey sequence especially low weight, we do not believe this kind of key-dependent differential characteristic will have any relevance in attacking Twofish.

A much more interesting issue in terms of key-dependent characteristics is whether the key-dependent S-boxes are ever generated with especially high probability differential or high bias linear characteristics. The statistical analysis presented earlier shows that the best linear and differential characteristics over all possible keys are still quite unlikely.

Note that the structure of both differential and linear attacks in Twofish is such that such attacks appear to generally require good characteristics through at least three of the four key-dependent S-boxes (if not all four), so a single high-probability differential or linear characteristic for one S-box will not create a weakness in the cipher as a whole.

[18] See [Haw98] for further cryptanalysis of IDEA weak keys.

7.12 Reed-Solomon Code

The RS structure helps defend against many possible related-key attacks by diffusing the key material in a direction "orthogonal" to the flow used in computing the 8-by-8-bit S-boxes of Twofish. For example, a single byte change in the key is guaranteed to affect all four key-dependent S-boxes in g. Since RS codes are MDS [MS77], the minimum number of different bytes between distinct 12-byte vectors generated by the RS code is guaranteed to be at least five. Notice that any attempt in a related-key attack to affect only a single byte in the computation of A or B is guaranteed to affect all four bytes in the computation of T_0 and T_1. The S-box keys are used in reverse order from the associated key bytes so that related-key material is used in a different order in the round function than in the subkey generation.

The reversible RS code used in Twofish was chosen via computer search to minimize implementation cost. The code generator polynomial is

$$x^4 + (\alpha + \frac{1}{\alpha})x^3 + \alpha x^2 + (\alpha + \frac{1}{\alpha})x + 1$$

where α is a root of the primitive polynomial $w(x)$ used to define the field.

Because all of these coefficients are "simple," this RS computation is easily performed with no tables, using only a few shifts and XOR operations; this is particularly attractive for smart cards and hardware implementations. This computation is only performed once per key schedule setup per 64 bits of key, so the concern in choosing an RS code with such simple coefficients was not the performance overhead, but saving ROM space or gates. Precomputing the RS remainders requires only 8 bytes of RAM on a smart card for 128-bit keys.

Note that the RS matrix multiply can be implemented as a simple loop using the generator polynomial without storing the coefficients of the RS matrix.

8 Cryptanalysis of Twofish

We have spent over one thousand man-hours attempting to cryptanalyze Twofish. A summary of our successful attacks is as follows:

- 5-round Twofish (without the post-whitening) with $2^{22.5}$ chosen plaintext pairs and 2^{51} computations of the function g.
- 10-round Twofish (without the pre- and post-whitening) with a chosen-key attack, requiring 2^{32} chosen plaintexts and about 2^{11} adaptive chosen plaintexts, and about 2^{32} work.

The fact that Twofish seems to resist related-key attacks well is arguably the most interesting result, because related-key attacks give the attacker the most control over the cipher's inputs. Conventional cryptanalysis allows an attacker to control both the plaintext and ciphertext inputs into the cipher. Related-key cryptanalysis gives the attacker an additional way into a cipher: the key schedule. A cipher that is resistant to attacks with related keys is necessarily resistant to simpler techniques that only involve the plaintext and ciphertext.[19]

Based on our analysis, we conjecture that there exists no more efficient attack on Twofish than brute force. That is, we conjecture that the most efficient attack against Twofish with a 128-bit key has a complexity of 2^{128}, the most efficient attack against Twofish with a 192-bit key has a complexity of 2^{192}, and the most efficient attack against Twofish with a 256-bit key has a complexity of 2^{256}.

8.1 Differential Cryptanalysis

This attack breaks a 5-round Twofish variant with a 128-bit key, without the post-whitening, with $2^{22.5}$ chosen plaintexts and 2^{51} work. Due to the high computational requirements, we have not implemented this attack, though we have validated some of its elements. We also discuss an attack on four rounds of Twofish with no post-XOR requiring 2^{32} work, and attacks on a Twofish variant with fixed S-boxes breaking six rounds with 2^{67} work, and breaking seven rounds with 2^{131} work. None of the attacks appears to be extensible to enough rounds to pose any substantial threat to the full 16 round Twofish.

8.1.1 Overview of the Attack

The general idea of the attack is to force a characteristic for the F function of the form $(a, b) \rightarrow (X, 0)$ to occur in the second round. We realize this characteristic internally by causing the same low Hamming-weight XOR difference to occur in the outputs of both g computations in the second round of the cipher. When this happens, and there are k bits in that XOR difference, there is a 2^{-k} probability that the

[19] We have discussed the relevance of related-key attacks to practical implementations of a block cipher in [KSW96, KSW97]. Most importantly, related-key attacks affect a cipher's ability to be used as a one-way hash function.

output of the PHT will be unchanged in one of its two words. That, in turn, leads to a detectable pattern in the output difference from the third round, and thus to a detectable pattern (with a great deal of work) in the output difference in the output from the cipher's fifth round.

8.1.2 The differential

The attack requires that we get a specific, predictable difference sequence. We choose:

r	$\Delta_{R_{r,0}}$	$\Delta_{R_{r,1}}$	$\Delta_{R_{r,2}}$	$\Delta_{R_{r,3}}$
0	0	0	a'	b'
1	a	b	0	0
2	0	X	a	b
3	Y	Z	0	X
4	Q	R	Y	Z
5	S	T	Q	R

where the table gives the difference patterns for the round values for each of the 5 rounds, and where $a' = ROL(a,1)$, $b' = ROR(b,1)$. We have $Z = ROL(a,1) \oplus F_{2,1}$, and the low-order bit of $F_{2,1}$ is guaranteed to be zero; also, we choose a so that the low-order bit of $ROL(a,1)$ is also zero. Then the difference Z is detectable, because the low-order bit of Z is always zero. To recognize right pairs, we shall (roughly speaking) guess the last-round subkey and decrypt up one round, checking whether the low-order bit of the difference is as expected. There are also several other ways to distinguish the Y, Z difference, but they do not change the difficulty of the attack. Note that X, Y, Z, S, T, Q, and R are all differences whose values we do not care about, so long as the low-order bit of Z is always zero.

The only critical event for this attack occurs at $r = 2$, where we need the characteristic $(a, b) \rightarrow (0, X)$ to happen with high probability. Based on the properties of the MDS matrix, we know that there are three single-byte output XORs for s_0 that lead to an output XOR of g with Hamming weight of only 8. If the same one of those differences goes into the MDS matrix in both g computations in the second round, then, with probability 2^{-8}, we get an offsetting pair of values in the two g outputs; one g output has some value added to it modulo 2^{32}, and the other output has the same value subtracted from it. When the outputs go through the PHT, they lead to only one output word from the whole F being changed, which is what makes this attack possible. Therefore, we choose $a = (\alpha, 0, 0, 0)$ to be a single-byte difference in s_0, and $b = ROR(a, 8)$ so that the single-byte difference into the second g computation in round 2 lines up with the single-byte difference in a.

The thing that makes this attack hard is getting that event to occur several times in succession. We choose input structures to guarantee that, after $2^{22.5}$ chosen plaintexts, we are very likely to get at least one batch of 2048 "right pairs," which have the 5-round differential characteristic shown above. With 2048 successive events of this kind, we can mount an attack in which we attempt to recover the S-box entries. When our guess is successful, these will be correct, and we will be able to recover most of the key.

8.1.3 The Input Structure

The input structure works as follows: The input pair is

$$(A, B, C, D), (A^*, B^*, C^*, D^*)$$

The input structures we use must thus accomplish two things:

1. Select 2^n plaintext pairs in such a way that there is a high probability of at least one pair being a right pair. Recall that a right pair is a pair that follows the whole differential characteristic described above.

2. For each plaintext pair generated in the previous step, generate about 2048 related plaintext pairs, such that if the first plaintext pair is a right pair, so are all 2048 related plaintext pairs.

The first step allows us to get the characteristic through; the second step allows us to mount the whole 5-round attack, by giving us enough data to recover the low bits of the S-box entries.

Getting One Potential Right Pair Let us consider pairs of inputs to the F function in the second round. We must first get a desired XOR difference to occur in the *output* of the active S-box of the attack, inside the g function computation. This XOR difference must occur in the output of the active S-box s_0 in both parallel g functions in the second round's F function. We will call a pair of texts for which this happens a *potential right pair*. We expect there to be between three and four output XOR differences from s_0 that will lead to some 8-bit output XOR difference from the whole g function, and we do not care which of these differences we hit. However, we do not know s_0, and so we cannot expect to choose the best bytewise input difference to s_0 to get one of our desired XOR differences.

Simulations show that, with a randomly selected s_0, a randomly selected (non-zero) input XOR difference has about a 1/2 probability of causing any desired output XOR difference, when it is tried on all 256 possible input values. (That is, $s_0[i] \oplus s_0[i \oplus \delta]$ will hit any desired output XOR with probability about 1/2, for any nonzero δ, when i is stepped through all possible byte values.) Our structure attempts to get this sequence of inputs into the same byte position in both parallel g functions in the second round's F function at the same time. This requires 256 different input pairs, there is a probability of 1/2 of one of them being a potential right pair for each of the three or four useful s_0 output XORs. This will generate a potential right pair with probability 7/8 if there are three useful s_0 output XORs.

There is a complication to all this. The second round's F function input is the right half of the plaintext, XORed with the first round's F function output. Since we do not know the first round's F function output, we cannot directly control the inputs to the second round. Instead, we must guess the XOR of the high-order byte of each word of the F function output. (This ignores the input whitening, but that merely offsets our guess and has no effect.) This means that, to have at least a 7/8 probability of getting one potential right pair, we must try 2^{16} different input pairs.

Getting Right Pairs from Potential Right Pairs A potential right pair has the property that it is getting the same XOR difference in the output of both parallel g functions in the second round. Because this XOR difference has a Hamming weight of only 8, the probability is at least 2^{-8} that the two g functions will have a very useful relationship modulo 2^{32}: one g function will have some value added to it modulo 2^{32}; the other g function in the same plaintext will have that value subtracted from it.

Consider two random values, U and V. If we XOR some nonzero value with Hamming weight 8, T, into both, it is possible that each time a bit in U is changed from a zero to a one, the corresponding bit in V is changed from a one to a zero. When this happens:

$$U + V = (U \oplus T) + (V \oplus T) \tag{2}$$

This is what happens in a right pair.

For any U and T, the values of V for which this holds are characterized by $(U \oplus T \oplus V) \wedge T = 0$ where the $\wedge$ is the bitwise-and operation. That is, in all bit positions where T has a 1-bit, the bit values of U and V must be opposite. Looking only at one bit position, XORing T into both U and V changes one of the bits to be added from zero to one, and the other bit to be added from one to zero. Thus, the result remains the same. When T has a Hamming weight of 8, then 2^{-8} of all V values will satisfy equation 2 for given U and T.

From a single potential right pair, we must now try many different values for the input to the second g function in the second round, without altering the high-order byte values. When we try about 256 different values of this kind with a potential right pair, we expect to get a real right pair. This can be done by changing any of the bytes that go to the second round's second g function, without changing any other bytes. This means that, from potential right pair $(A, B, C, D), (A^*, B^*, C^*, D^*)$, we derive a new potential right pair by leaving the relationship between the pair the same, but altering D in its low-order two bytes.

Because we do not know which of the plaintext pairs described above has the potential right pair, we must apply this process to all of them. Thus, we generate 256 different plaintext pairs for each of the 2^{16} plaintext pairs we already have, giving 2^{24} plaintext pairs. Out of all this, we expect to get at least one right pair.

In Section 8.3.1, we found empirically that a characteristic of the form $(a, b) \rightarrow (0, X)$ for the round function has a probability of about 2^{-20}. This allows us to improve the probability of the characteristic for the F function, and reduce the number of plaintext pairs needed to about 2^{20}.

Turning One Right Pair into 2048 Right Pairs We now have 2^{20} plaintext pairs, of which at least one is, with fairly high probability, a right pair. Unfortunately, to mount the 5-round attack, we need a batch of 2048 such pairs. The reason we need 2048 pairs is that we will recover two bits of information about each entry in each S-box; this makes for $2 \cdot 4 \cdot 256 = 2048$ unknown bits, and each of the 2048 pairs give us one bit of information on those unknowns. Once we have recovered information on all (or most) S-box entries, we can then readily recover the keying material entering each S-box by brute force.

Consider, again, equation 2. We already saw that this equation is equivalent to $(U \oplus T \oplus V) \wedge T = 0$. This implies $((U \oplus W) \oplus T \oplus (V \oplus W)) \wedge T = 0$, for any value of W. So if we have a (U, V, T) that satisfies equation 2, then $(U \oplus W, V \oplus W, T)$ also satisfies that equation for any value of W.

This gives us a way to derive many plaintext pairs from one right pair. If we have a right pair $(A, B, C, D), (A^*, B^*, C^*, D^*)$, then we can generate a new right pair, $(A_i, B_i, C_i, D_i), (A_i^*, B_i^*, C_i^*, D_i)$. To do this, we must manipulate the C_i, D_i values in such a way that the same XOR difference occurs in the output of both g functions in the second round. In doing this, we face the same constraints as in getting a desired output XOR difference from s_0, and so we use the same solution: send in the same byte inputs to s_1 in C and D. We then alter the byte inputs to s_1 in the altered C_i and D_i in the same way, so that the same actual pair of inputs occurs in s_1 in both g functions. This is guaranteed to give the same output XOR in both g functions. The C_i^*, D_i^* values have the same relationship to the C_i, D_i values as do the C^*, D^* values to the C, D values.

This introduces still one more complication to our input structure. We again do not know the output from the first round's F function, and so do not know how to get identical bytes into s_1 in the second round. We must try 256 different possible values in our inputs, to deal with all possible byte XORs between the s_1 input in C and in D in the second round. This increases the total number of batches of plaintext pairs requested by a factor of 256.

Summary of the Input Structure To get a five round attack to work, the input structure is highly complex. We thus summarize how it is built:

1. We fix A and B throughout the attack.
2. Let the bytes of C, D be denoted by (c_0, c_1, c_2, c_3),(d_0, d_1, d_2, d_3). We fix c_3, d_2, d_3.
3. We let c_1 range over all possible values, and force $d_1 = c_1$.
4. We let c_2 range over 16 possible values.
5. We let the pair (c_0, d_0) range over any 1450 possible values; those values can be arbitrary, so long as they are distinct.
6. Finally, we form all $2^8 \cdot 16 \cdot 1450 = 2^{22.5}$ possible combinations of $c_1, d_1, c_2, (c_0, d_0)$, and get the encryption of the $2^{22.5}$ resulting plaintexts. This gives us 2^{44} possible plaintext pairs which can be chosen from the set of available texts; we shall use them as described below. The intuition is that $2^{44} > 2^{18} \cdot 256 \cdot 2048$, so it is plausible that there could be enough plaintext pairs, if the input structure is well-chosen.

Using the Structure We shall describe how we use the available plaintexts to construct the necessary plaintext pairs. First, we guess the value v of the byte XOR mentioned above (used to ensure that the s_1 inputs are the same in round 2). From now on, we shall think of v as fixed. We can restrict our attention to those plaintext pairs $(C, D), (C^*, D^*)$ with $c_1 \oplus c_1^* = v$; for those pairs, we'll also have $d_1 \oplus d_1^* = v$ for free; and those two relations ensure that the byte inputs into s_1 match up in the second round as desired if we have guessed v correctly. This gives us about $2^{44}/2^8 = 2^{36}$ plaintext pairs with appropriate values of $(c_1, d_1), (c_1^*, d_1^*)$. We shall also insist that $c_2 = c_2^*$; this reduces the number of usable plaintext pairs to $2^{36}/16 = 2^{32}$.

Next let's look for a right pair. Suppose that some value of $(C, D), (C^*, D^*)$ leads to a right pair. Then we note that what makes this a right pair depends only on $(c_0, d_0), (c_0^*, d_0^*)$. Therefore, given any one right pair, you can easily get a batch of $2^{32}/1450^2 = 2048$ right pairs by fixing $(c_0, d_0), (c_0^*, d_0^*)$ and letting c_1, c_1^*, c_2, c_2^* range over all possible values such that $c_1 \oplus c_1^* = v$ and $c_2 = c_2^*$.

Based on these observations, we try each of the 1450^2 possibilities for $(c_0, d_0), (c_0^*, d_0^*)$ in turn, testing each of them for a right pair. When we find one right pair, the structure ensures we'll find 2048 simultaneous right pairs.

8.1.4 Recovering Key Material

We describe next how to use a batch of 2048 simultaneous right pairs to recover all of the key material entering the S-boxes. We focus on one value of $(c_0, d_0), (c_0^*, d_0^*)$, considering it fixed; the issue is how to identify whether it contains 2048 right pairs, and if so, how to recover key material.

Suppose we knew we had a batch of right pairs. Then, the difference T is visible from a ciphertext pair, and $T = ROL(Z, 1) \oplus \Delta F_{4,1}$. Of course, we know that the low-order bit of Z is zero; therefore, we need only look at T and $F_{4,1}$ mod 4. We let t_0, t_1 be the outputs of the two g functions in round four, so that $F_{4,1} = t_0 + 2t_1 + K_{17} \bmod 2^{32}$. We guess the low-order bit of K_{17}, so that the low-order bit of $F_{4,1}$ depends only on the low-order bit of t_0, and the next-lowest bit of $F_{4,1}$ depends only on an XOR of the low-order bit from t_1, the next-lowest bit from t_0 (and possibly the low-order bit of t_0, depending on K_{17} mod 2). This, in turn, means that the next-lowest bit of the observable difference T depends only on the xor of low-order bits from the g function outputs.

At this stage, we could guess the key material entering the S-boxes, compute the low-order bits of t_0 and t_1 for each ciphertext pair, and check our guess against the observed values of T. However, this would already require 2^{64} work for a 128-bit key, and 2^{128} work for a 256-bit key, which is too high for comfort. Therefore, we propose an alternate analysis method with much lower complexity.

Our key recovery technique is based on establishing and solving linear equations. We construct $2 \cdot 4 \cdot 256 = 2048$ formal unknowns $x_{i,j,k} \in \mathrm{GF}(2)$ as two linear combinations on each of the 256 entries in each of the four S-boxes. For example, $x_{0,3,17}$ would represent the low-order bit of the output of the MDS matrix when applied to $s_3(17)$; similarly for the other x's. The main observation is that the observable difference T gives us one equation on those unknowns, for each ciphertext pair which arises from a right pair. This is true because the low-order bit of t_0 is obtained as a GF(2)-linear combination of x's, namely $x_{0,0,p} \oplus x_{0,1,q} \oplus x_{0,2,r} \oplus x_{0,3,s}$ if the input to g is (p, q, r, s). With this idea in mind, we use our batch of 2048 pairs to generate 2048 linear equations on the 2048 unknowns. This system can be solved by standard linear algebra techniques (such as Gaussian elimination, although a more sophisticated algorithm is more appropriate since we will have a very sparse matrix).

Therefore, our algorithm for identifying right pairs and recovering key material is as follows. Look at the 2048 candidate pairs generated by one value of $(c_0, d_0), (c_0^*, d_0^*)$, using them to generate a system of 2048 linear equations on 2048 unknowns. If this system of linear equations has no solution, then we know that those candidate pairs do not represent right pairs, and we may move on to another value of $(c_0, d_0), (c_0^*, d_0^*)$. (When we are looking at the wrong value of $(c_0, d_0), (c_0^*, d_0^*)$, the system of linear equations will be inconsistent with good probability, and thus the cost of filtering should not be too high.) Otherwise, we solve the system of linear equations, thus obtaining two bits of information on each S-box entry (or on most of them). This enables us to isolate the key material entering each S-box, and solve for each S-box in turn. The key material entering, say, s_0 can be recovered by a brute force search using the known values of $x_{\cdot,0,\cdot}$; this between 2^{16} work (for a 128-bit key) and 2^{32} work (for a 256-bit key). (Alternatively, the key material entering s_0 could be recovered by a table lookup in a 2^{16}–2^{32}-entry table.) If some S-box does not admit any choice of keys, then we can also reject the batch of 2048 candidate pairs, and move on to another value of $(c_0, d_0), (c_0^*, d_0^*)$. However, when we finally encounter a right pair, this procedure will recover all of the key material entering the S-boxes. Finally, the rest of the key can be recovered by other techniques; for example, see below for a more effective differential attack which works when the S-boxes are known.

In the end, we reject each wrong batch of pairs with about 2048^2 work, on average, and recover the key material from the right pairs with at most 2^{32} work. This leads to a total work factor of about $1450^2 \cdot 2048^2 \cdot 256 + 2^{32} = 2^{51}$; in addition, the attack needs about $2^{22.5}$ chosen plaintexts.

8.1.5 Refinements and Extensions

Attacking Four Rounds The above structure can be used in a much easier 4-round attack. The attacker can immediately rule out all but the right batch of ciphertext pairs, since he can see the Z difference word directly. He can then mount a 2^{32} effort search for S; the right key will reveal an unchanged low-order bit in A after g is computed. The total work expected is 2^{32} trial g computations.

Known g Keys We can also consider a variant of Twofish with known S-boxes in g. In this case, we can use the same basic attack, but push it through more rounds.

Consider a 6-round Twofish variant with this input structure. To expose the low-order bit of the Z difference, we must now guess the last round's subkey. A correct guess will allow us to compute the fifth round's g functions, which will allow us to recognize the sequence of right pairs by the low-order bit of the Z difference, just as in the above attack. There is an improvement, however: we only really need to know the first g computation result in the fifth round, since the PHT has the effect of making the low-order bit of the second word of F function output dependent only on that value and one bit of the fifth round subkey. This means we need to know only the first input to the g function in the fifth round, which means we need only guess 32 bits of the sixth round subkey. The resulting attack breaks six rounds with about 2^{67} work.

Consider a 7-round Twofish variant with this input structure. We now need to know the whole input to the sixth round F function. If we guess the whole seventh round subkey, and 32 bits of the sixth round subkey, we can mount the same attack. We thus get an attack with 2^{131} work, which is of some theoretical interest against 192-bit and 256-bit keys.

It may also be possible to improve the number of plaintexts required by a factor of several thousand, because we now know the difference tables for our S-boxes.

8.2 Extensions to Differential Cryptanalysis

8.2.1 Higher-Order Differential Cryptanalysis

Higher-order differential cryptanalysis looks at higher order relations (e.g., quadratic) between pairs of plaintext and ciphertexts [Lai94, Knu95b]. These attacks seem to work best against algorithms with simple algebraic functions, only a few rounds, and poor short-term diffusion. In particular, we are not aware of any higher-order differential attack reported in the open literature that is successful against more than 6 rounds of the target cipher. We cannot find any higher-order differentials that can be exploited in the cryptanalysis of Twofish.

8.2.2 Truncated Differentials

Attacks using truncated differentials apply a differential attack to only a partial block [Knu95b]. We have not found any truncated attacks against Twofish. The almost complete diffusion within a round function makes it very difficult to isolate a portion of the block and ignore the rest of the block. Truncated differential attacks are most often successful when most of the cipher's internal components operate on narrow bit paths; Twofish's 64-bit-wide F function seems to make truncated differential characteristics hard to find. Additionally, truncated differential attacks are often difficult to extend to more than a few rounds in a cipher. We believe that Twofish is secure against truncated differential attacks.

8.3 Search for the Best Differential Characteristic

When assessing the security of ciphers built out of S-boxes and simple diffusion boxes, it is often easy to obtain a crude lower bound on the complexity of a differential attack by simply bounding the number of active S-boxes that must be involved in any successful differential characteristic. (An inactive S-box is one that is fed with the trivial characteristic $0 \rightarrow 0$; all others are considered active.) If any differential characteristic of sufficient length must include at least n active S-boxes, and there is no characteristic for any of the S-boxes with probability greater than $\mathrm{DP}_{\max}$, then one can conclude that the probability of any differential characteristic can be at most $(\mathrm{DP}_{\max})^n$. Of course, this approach often results in extremely crude and conservative bounds, in the sense that the true complexity of mounting a practical differential attack is often much higher than the bound indicates. Nonetheless, when $(\mathrm{DP}_{\max})^n$ is sufficiently small, the results can provide powerful evidence of the cipher's security against differential cryptanalysis.

We follow this approach to develop additional evidence that Twofish is likely to be secure against differential attacks. Section 7.2.3 already provides convenient values of $\mathrm{DP}_{\max}$ for us; the hard part is to determine a bound on the number n of active S-boxes.

Our approach was based on the observation that the number of active S-boxes can be counted by examining the active bytes positions of the examined differential characteristics. By an *active byte* of a differential characteristic, we mean one that has a non-zero difference at that position of the characteristic. Of course, an S-box in the F function is active exactly when its input byte is active, so characterizing the pattern of active bytes in a given differential characteristic allows us to count the number of S-boxes it includes. Moreover, due to the heavily byte-oriented nature of the F function, we can obtain bounds on the propagation of active bytes through the F function; for example, if there is one active byte at the input to the F function, there must be at least four active bytes at its output. Furthermore, knowing the pattern of active bytes in the difference at the output of the F function lets one derive constraints on the pattern of active bytes at the inputs of F functions at neighboring rounds.

For convenience, we introduce a simple notation to indicate the location of active bytes: a `x` represents an (unspecified) non-zero byte difference (and thus an active byte position), while a `0` represents a zero byte difference (and thus an inactive byte position). An even shorter notation treats the string specification of active bytes as a binary string, and packs it into a hexadecimal string for more convenient display. For example, the characteristic $\texttt{0000000000B4000C}_{16} \rightarrow \texttt{591FE47201234500}_{16}$ for the F function might be represented in our first notation as `00000x0x` $\rightarrow$ `xxxxxxx0`, which indicates that two of the byte positions in the input difference are active and all but one of the bytes in the output difference are active; a shorter form for the

same characteristic is $05_{16} \rightarrow FE_{16}$. Thus, any one such pattern represents a large class of possible differential characteristics.

In this approach, we can count the minimum number of S-boxes over all differential characteristic by fully searching the set of all patterns of active bytes. Our implementation used Matsui's algorithm for pruned search of all differential characteristics [Mat95]. We conservatively assumed that a good analyst might be able to bypass the first round of Twofish and mount a 3-R attack, thus needing to cover only rounds 2–13. Therefore, we concentrated our efforts on 12-round characteristics.

When implementing this approach, it is difficult to model the effects of the one-bit rotations and the PHT precisely, since they introduce non-linear dependence across byte boundaries. To simplify the task, we considered a restricted model that introduces two small inaccuracies:

1. We ignore the one-bit rotations. (In some cases, the one-bit rotations can cause diffusion across byte boundaries in a form that we did not model.)

2. A few troublesome differential characteristics for the PHT are ignored: namely, those that rely on a carry bit propagating through at least 8 bit positions. For example, the characteristic $(01FFFE80_{16}, 01FFFE80_{16}) \rightarrow (00000100_{16}, 00000080_{16})$ for the PHT has positive probability, yet it is not considered in our model. These sorts of characteristics were omitted because they are harder to characterize in full generality, and also because they have very low probability. (Experiments suggest that the previous example has a probability of perhaps 2^{-36} or so.)

Any attack which does not violate this model must obey our bounds below.

In this model, we found that there are no good high-probability differential characteristics covering more than 12 rounds of Twofish. Our results show that, under this model, the best 12-round differential characteristic must involve at least 20 active S-boxes.

The reasoning is as follows. Our implementation of Matsui's algorithm took too long to search the space of all differential characteristics that were 9 rounds or longer, so we were forced to generalize from data on characteristics covering 8 or fewer rounds. The program did show that covering 4 rounds requires at least 6 active S-boxes, covering 6 rounds requires at least 10 active S-boxes, and covering 8 rounds requires at least 14 active S-boxes. Of course, any 12-round characteristic must include a sub-characteristic covering the first 4 rounds and a sub-characteristic covering the last 8 rounds; therefore, any characteristic covering 12 rounds must include at least $6 + 14 = 20$ active S-boxes. (This bound is probably not tight.)

For example, one of the best 8-round characteristics found was

```
1 88->FE  (2 active)
2 88<-00  (0 active)
3 88->00  (2 active)
4 88<-F0  (4 active)
5 00->F0  (0 active)
6 00<-F0  (4 active)
7 88->F0  (2 active)
8 88<-00  (0 active)
9 88  00
```

Here we see that the characteristics for the F function were $88_{16} \rightarrow FE_{16}$, $0 \rightarrow 0$, $88_{16} \rightarrow F0_{16}$, and so on.

We may now add in the results of Section 7.2.3. For all 128-bit keys, we are guaranteed that $\mathrm{DP}_{\max} \leq 18/256$, so any attacker would need at least $(\mathrm{DP}_{\max})^{-20} \approx 2^{76.6}$ chosen plaintexts; for 192-bit keys, we have $\mathrm{DP}_{\max} \leq 24/256$, so the best an attacker could hope for is an attack needing $2^{68.3}$ chosen plaintexts. Of course, even if these attacks were possible, they could only work for a very small class of weak keys (constituting about 2^{-48} of the 128-bit keyspace, or in the second case, about 2^{-64} of the 192-bit keyspace).

It is much more natural to demand that the attack work for a significant percentage of all keys. In this case, $\mathrm{DP}_{\max} = 12/256$ is a more representative value (for all key sizes), and then any differential attack working for a significant fraction of the keyspace would require at least $2^{88.3}$ chosen plaintexts. This already rules out all practical attacks.

These estimates are likely to significantly underestimate the complexity of a differential attack. We list here some of the practical barriers a real attacker would have to surmount:

1. The attacker would have to find specific differences for the active byte positions that fit together and that still make the attack work. This involves picking specific differences, where the average probability of any particular differential of an S-box is of course *much* lower than the maximum probability.

(Doing so is already hard within the model; the one-bit rotations make it even harder, since they eliminate any hope for good iterative characteristics.)

2. Those specific differences would have to propagate through the key-dependent S-boxes, the MDS matrix, the PHT, and the subkey addition with high probability. The PHT and subkey addition are particularly thorny barriers: in our model, the attacker was not charged with the cost of pushing a difference through them, but in real life, the overall probability of the differential characteristic would be significantly reduced by the combined probabilities of passing the non-trivial round characteristics through the PHT and subkey addition. An attacker would have to find a way to minimize those costs. Because the PHT and the subkey addition rely on addition modulo 2^{32}, pushing a difference with significant probability through requires a difference with relatively low Hamming weight. However, this imposes significant constraints on the difference pushed through the MDS matrix; and Section 7 shows that it is very difficult to cause the difference at the output MDS matrix to have low Hamming weight.

3. The attacker would most probably have to find a high-probability characteristic for each of the four S-boxes $s_0, \ldots, s_3$ at the same time. We have seen earlier that attaining $\mathrm{DP}_{\max}$ for even one S-box requires exceptional luck—only about 2^{-12} of all 128-bit keys attain $\mathrm{DP}_{\max} = 18/256$ at any one S-box—and attaining $\mathrm{DP}_{\max}$ at all four S-boxes places even more constraints on the key. Of course, even for the small class of weak keys where this constraint is satisfied, the attacker would have to learn which input and output differences for the S-box attain that maximum, and identify a way to piece them together into full 12-round characteristics. Since the S-boxes are key-dependent, even the former task is non-trivial, and the latter may well be impossible in most cases.

Therefore, we believe that any realistic differential attack is likely to require many more texts, in practice, than our bound might indicate.

A brief note on the implications of these results is in order. Recall that our model is an imperfect idealization of Twofish; in some cases, the abstraction has introduced inaccuracies. The right way to interpret these results, then, is to conclude that any differential attack which respects this model is extremely unlikely to succeed.

This provides compelling evidence that attackers who respect our model will probably get nowhere. But what about the adversary who "breaks the rules"?

We believe that piecing together an attack that avoids the difficulties imposed by our model will not be easy. There are three choices:

1. Try to take advantage of the one-bit rotations. We believe that the one-bit rotations make cryptanalysis harder, if they have any effect at all. By forcing neighboring characteristics for the round function to "line up" with shifted versions of each other, the one-bit rotation makes it very hard to piece together several short characteristics into a useful characteristic covering many rounds. As an important special case, this should make good iterative characteristics very hard to find (if they even exist). As another important special case, if s_0 (say) has only one input difference with probability $\mathrm{DP}_{\max}$, it makes it difficult to use that input difference every time that s_0 is active; this serves to increase the difficulty of finding high-probability characteristics even further.

2. Try to model the PHT more accurately than we have done. This might be an promising avenue for further exploration. However, the characteristics for the PHT which we modeled inaccurately are hard to use in real attacks: they are rare (which means they are hard to piece together with good characteristics for other cipher components), and they typically have a very low probability.

3. Try to do both at the same time. This should be even harder than either of the above.

While our results with this model do not rule out the possibility that such an approach to differential cryptanalysis might be made to work, it seems clear that the analyst will be hard-pressed to achieve much success.

In short, all evidence available to us suggests that differential attacks on Twofish are likely to be well out of reach for the foreseeable future.

8.3.1 Differential characteristics of F

We ran a test for differential characteristics of the F function with one active byte in each 4-byte g input and at most 4 active bytes at the 8-byte F output. We found some interesting differentials. For example:

Prob.	Differential
2^{-20}	x000 0x00 → 0000 xxxx
2^{-26}	x000 0x00 → 0000 0xxx
2^{-22}	x000 0x00 → xxxx 0000
2^{-26}	000x x000 → 0000 xx0x
2^{-21}	000x x000 → 0000 xxxx
2^{-22}	000x x000 → xxxx 0000
2^{-20}	00x0 000x → 0000 xxxx
2^{-22}	00x0 000x → xxxx 0000
2^{-14}	0x00 0000 → 0x0x 0x0x

These probabilities are very approximate as they derive from numerical experiments with not too many samples. Note that in all but the last of these, the two active input bytes are going through the same key-dependent S-box due to the 8-bit rotate at the input of the second g function. The leading cause of all of these differentials seems to be that these two bytes generate the same output differential from the S-box, and thus the same differential at the output of the MDS matrix. With some positive probability, the PHT maps this into an output differential where one half is zero. This probability depends on the bit pattern of the differential just before the PHT.

The last characteristic has a different explanation. The input differential results in a differential at the output of the MDS matrix that has a relatively high chance of being converted into a 0x0x pattern after a constant is added to it. For example, the differential $\texttt{03f2037a}_{16}$ has a fair chance of being converted to a `0x0x` pattern after a 32-bit addition.

All of these characteristics have a relatively low probability. We have not found any attack that directly relates to this property, but it provides a useful starting point for further research. In particular, upper bounds on the differential characteristics of F of this type could be used to improve our bound on the best differential characteristic for the full cipher.

8.4 Linear Cryptanalysis

We perform a similar analysis to that of Section 8.3 in the context of linear cryptanalysis. The approach is much the same, as are the results.

Our model is much as before: it is centered around the pattern of active bytes, and it involves a few inaccuracies in special cases. (Here a byte position is considered *active* if any of its bits are active in the linear approximation for that round; i.e., if the mask Γ selects at least one bit from that byte position.) Our characterization of linear approximations for the PHT was a bit weaker than the corresponding analysis for differential attacks; as a result, our model is less accurate than before. In particular:

1. As before, the one-bit rotations are not always treated properly.
2. More importantly, our model of the PHT fails to accurately treat some low-probability approximations for the PHT where there are fewer active bytes at the output than at the input.

Because of the second limitation, our model should be considered only a heuristic; it may fail to capture some important features of the round function.

A search for the best linear characteristics within this model found that every 12-round characteristic has at least 36 active S-boxes. Here is an example of one characteristic we found:

```
 1 FE->FF  (1 active)
 2 FF<-FF  (2 active)
 3 FF->DD  (4 active)
 4 CC<-DD  (6 active)
 5 CC->00  (0 active)
 6 CC<-00  (6 active)
 7 CC->77  (4 active)
 8 00<-77  (0 active)
 9 00->77  (4 active)
10 66<-77  (6 active)
11 66->00  (0 active)
12 66<-00  (3 active)
13 66  07
```

The first few approximations for the F function are $\texttt{01}_{16} \to \texttt{FF}_{16}$, $\texttt{22}_{16} \to \texttt{FF}_{16}$, $\texttt{33}_{16} \to \texttt{DD}_{16}$, and so on.

To translate these results into an estimated attack complexity, we turn to the results of Section 7.2.3. We have $\mathrm{LP}_{\max} \le (108/256)^2$, so this suggests an attacker would need at least $(\mathrm{LP}_{\max})^{-36} \approx 2^{89.6}$ known plaintexts; and such an analysis would only work for a very small class of weak keys (representing about $2^{-49.6}$ of the keyspace for 128-bit keys, or about 2^{-68} of the 192-bit keyspace).

It is much more natural to demand that the attack work for a significant percentage of all keys. In this

case, $\mathrm{LP}_{\max} = (80/256)^2$ is a much more representative value, and thus in this model any linear attack working for a significant fraction of the keyspace would require at least $2^{120.8}$ chosen plaintexts.

These results are only heuristic estimates, and probably overestimate the probability of the best linear characteristic. Nonetheless, they present some useful evidence for the security of Twofish against linear cryptanalysis.

8.4.1 Multiple Linear Approximations

Multiple linear approximations [KR94, KR95] allow one to combine the bias of several high-probability linear approximations. However, it only provides a significant advantage over traditional linear cryptanalysis when there are a number of linear approximations whose bias is close to that of the best linear approximation. In practice, this seems to improve linear attacks by a small constant factor. Hence, we do not feel that Twofish is vulnerable to this kind of cryptanalysis.

8.4.2 Non-linear Cryptanalysis

Another generalization of linear cryptanalysis looks at non-linear relations [KR96a]: e.g., quadratic relations. While this attack, combined with the technique of multiple approximations [KR94], managed to improve the best linear attack against DES a minute amount [SK98], we do not believe it can be brought to bear against Twofish for the same reasons that it is immune to linear cryptanalysis.

8.4.3 Generalized Linear Cryptanalysis

This generalization of linear cryptanalysis uses the notion of binary I/O sums [HKM95, Har96, JH96]. An attacker attempts to find a statistical imbalance that can be described as the result of some group operation on some function of the plaintext and some function of the ciphertext. We have not found any such statistical imbalances, and believe Twofish to be immune to this kind of analysis.

8.4.4 Partitioning Cryptanalysis

Partitioning cryptanalysis is another generalization of linear cryptanalysis [Har96, JH96, HM97].[20] An attacker trying to carry out a partitioning attack is generally trying to find some way of partitioning the input and output spaces of the round function so that knowledge of which partition the input to a round is in gives some information about which partition the output from a round is in. This can be seen as a general form of a failure of the block cipher to get good confusion; an attacker after N rounds can distinguish the output from the Nth round from a random block of bits, because the output is somewhat more likely to be in one specific partition than in any of the others. This can be used in a straightforward way to attack the last round of the cipher.

We have been unable to find any useful way to partition the input and output spaces of the Twofish F function or a Twofish round that works consistently across many keys, because of the key-dependent S-boxes. For the 128-bit key case, there are 2^{64} different F functions, presumably each with its own most useful partitioning. We are not aware of any general way to partition F's inputs and outputs to facilitate such attacks.

8.4.5 Differential-linear Cryptanalysis

Differential-linear cryptanalysis uses a combination of techniques from both differential and linear cryptanalysis [LH94]. Due to the need to cover the last part of the cipher with two copies of a linear characteristic, the bias of the linear characteristic is likely to be extremely small unless the linear portion of the attack is confined to just three or four rounds. (The available linear characteristics for Twofish's round function have a relatively low probability, and are very hard to combine due to the MDS and PHT mappings.) This means that the cryptanalyst would need to cover almost all of the rounds with the differential characteristic, making a differential-linear analysis not much more powerful than a purely differential analysis. Therefore, we feel that differential-linear cryptanalysis is unlikely to be successful against the Twofish structure. In our analysis, we have found no differential-linear attacks that work against Twofish.

8.5 Interpolation Attack

The interpolation attack [JK97, MSK98b] is effective against ciphers that use simple alegbraic functions. The principle of the attack is simple: if the ciphertext can be represented as a polynomial or rational expresson (with N coefficients) of the plaintext, then the polynomial or rational expression can be reconstructed using N plaintext/ciphertext pairs.

[20] Similar ideas can be found in [Vau96b].

However, interpolation attacks are often only workable against ciphers with a very small number of rounds, or against ciphers whose rounds functions have very low algebraic degree. Twofish's S-boxes already have relatively large algebraic degree, and the combination of operations from different algebraic groups (including both addition mod 2^{32} and XOR) should help increase the degree even further. Therefore, we believe that Twofish is secure against interpolation attacks after even only a small number of rounds.

8.6 Partial Key Guessing Attacks

A good key schedule should have the property that, when an attacker guesses some subset of the key bits, he does not learn very much about the subkey sequence or other internal operations in the cipher. The Twofish key schedule has this property.

Consider an attacker who guesses the even words of the key M_e. He learns nothing of the key S to g. For each round subkey block, he now knows A_i. If he guesses K_0, he can compute the corresponding K_1. He can carry this attack out against as many round subkeys as he likes, but each guess takes 32 bits. We can see no way for the attacker to actually test the 96-bit guess that it would take to attack even one round's subkey in this way on the full Twofish.

An alternative is to guess the key input S to g. This is only half the length of the full key M, but provides no information about the round keys K_i. The differential attack described in Section 8.1 is the best way we were able to find to test such a partial key guess. We can see no way to test a guess of S on the full sixteen round Twofish.

8.7 Related-key Cryptanalysis

Related-key cryptanalysis [Bih94, KSW96, KSW97] uses a cipher's key schedule to break plaintexts encrypted with related keys. In its most advanced form, differential related-key cryptanalysis, both plaintexts and keys with chosen differentials are used to recover the keys. This type of analysis has had considerable success against ciphers with simplistic key schedules—e.g., GOST [GOST89] and 3-Way [DGV94b]—and is a realistic attack in some circumstances. A conventional attack is usually judged in terms of the number of plaintexts or ciphertexts needed for the attack, and the level of access to the cipher needed to get those texts (e.g., known plaintext, chosen plaintext, adaptive chosen plaintext).

8.7.1 Resistance to Related-key Slide Attacks

A "slide" attack occurs in an iterated cipher when the encryption of one block for rounds 1 through n is the same as the encryption of another block for rounds $s+1$ to $s+n$. An attacker can look at two encryptions, and can slide the rounds forward in one of them relative to another. S-1 [Anon95] can be broken with a slide attack [Wag95a]. Travois [Yuv97] has identical round functions, and can also be broken with a slide attack. Conventional slide attacks allow one to break the cipher with only known- or chosen-plaintext queries; however, as we shall see next, there is a generalization to related-key attacks as well.

Related-key slide attacks were first discovered by Biham in his attack on a DES variant [Bih94]. To mount a related-key slide attack on Twofish, an attacker must find a pair of keys M, M^* such that the key-dependent S-boxes in g are unchanged, but the subkey sequences slide down one round. This amounts to finding, for each of the eight byte-permutations used for subkey generation, a change in the keys such that:

$$s_i(j, M) = s_i(j + 2s, M^*)$$

for n values of j. In total, this requires $8n$ of these relations to hold.

Let us look in more detail for a fixed key M. Let $m \in \{5, \ldots, 8\}$ be the number of S-boxes used to compute the round keys that are affected by the difference between M and M^*. Observe that $m \geq 5$ due to the restriction that S cannot change and the properties of the RS matrix that at least 5 inputs must change to keep the output constant. There are at most $\binom{8}{m} 2^{32m-128}$ possible choices of M^*. We have a total of nm 8-bit relations that need to be satisfied. The expected number of M^* that satisfy these relations is thus $\binom{8}{m} \cdot 2^{-8nm+32m-128}$. For $n \geq 4$ this is dominated by the case $m = 5$; we will ignore the other cases for now. So for each M we can expect about 2^{38-40n} keys M^* that support a slide attack for $n \geq 4$. This means that any specific key is unlikely to support a slide attack with $n \geq 4$. Over all possible key pairs, we expect $2^{293-40n}$ pairs M, M^* for which a slide of $n \geq 4$ occurs. Thus, it is unlikely that a pair exists at all with $n \geq 8$.

Swapping Key Halves It is worth considering what happens when we swap key halves. That is, we swap the key bytes so that the values of M_e and M_o are exchanged. In that case, the sequence of A_i

and B_i values totally changes, because of the different index values used. We can see no useful attack that could come from this.

Permuting the Subkeys Although there is no known attack stemming from it, it is interesting to ask whether there exist pairs of keys that give permutations of one another's subkeys. There are 20! ways that the rounds' subkey blocks could be permuted. This is almost as large as 2^{64}, and so there may very well be pairs of keys that give permutations of one another's round subkey blocks.

8.7.2 Resistance to Related-key Differential Attacks

A related-key differential attack seeks to mount a differential attack on a block cipher through the key, as well as or instead of through the plaintext/ciphertext port. Against Twofish, such an attack must control the subkey difference sequence for at least the rounds in the middle. For the sake of simplifying discussions of the attack, let us consider an attacker who wants to put a chosen subkey difference into the middle twelve rounds' subkeys. That is, he wants to change M to M^*, and control $D[i, M, M*]$ for $i = 12..35$. At the same time, he needs to keep the g function, and thus the key S, from changing. All else being equal, the longer the key, the more freedom an attacker has to mount a related-key differential attack. We thus will assume the use of 256-bit keys for the remainder of this section. Note that a successful related-key attack on 128- or 192-bit keys that gets only zero subkey differences in the rounds whose subkey differences it must control translates directly to an equivalent related-key attack on 256-bit keys.

Consider the position of the attacker if he attempts a related-key differential attack with different S keys. This must result in different g outputs for all inputs, since we know that there are no pairs of S values that lead to identical S-boxes. Assuming the pair of S values does not lead to linearly related S-boxes, it will not be possible to compensate for this change in S with changes in the subkeys in single rounds. The added difficulty is approximately that of adding 24 active S-boxes to the existing related-key attack. For this reason, we believe that any useful related-key attack will require a pair of keys that keeps S unchanged.

8.7.3 The Zero Difference Case

The simplest related-key attack to analyze is the one that keeps both S and also the middle twelve rounds' subkeys unchanged. It thus seeks to generate identical A and B sequences for twelve rounds, and thus to keep the individual byte sequences used to derive A and B identical.

The RS code used to derive S from M strictly limits the ways an attacker can change M without altering S. The attacker must try to keep the number of active subkey generating S-boxes as low as possible, since each active S-box is another constraint on his attack. The attacker can keep the number of active S-boxes down to five without altering S, and so this is what he should do. With only the key bytes affecting these five subkey generation S-boxes active, he can alter between one and four bytes in all five S-boxes; the nature of the RS matrix is that if he needs to alter four bytes in any one of these S-boxes, he must alter bytes in all five. In practice, in order to maximize his control over the byte sequences generated by these S-boxes, he must alter four bytes in all five active S-boxes.

To get zero subkey differences, the attacker must get zero differences in the byte sequences generated by all five active S-boxes. Consider a single such byte sequence: the attacker tries to find a pair of four-byte key inputs such that they lead to identical byte sequences in the middle twelve rounds, which means the middle twelve bytes. There are 2^{63} pairs of key inputs from which to choose, and about 2^{95} possible byte sequences available. If the byte sequences behave more-or-less like random functions of the key inputs, this implies that it is extremely unlikely that an attacker can find a pair of key inputs that will get identical byte sequences in these middle twelve rounds. We discuss this kind of analysis of byte sequences in Section 7.11.2. From this analysis, we would not expect to see a pair of keys for even one S-box with more than eight successive bytes unchanged, and we would expect even eight successive bytes of unchanged byte sequence to require control of all four key bytes into the S-box. We would expect a specific pair of key bytes to be required to generate these similar byte sequences.

To extend this to five active S-boxes, we expect there to be, at best, a single pair of values for the twenty active key bytes that leave the middle eight subkeys unchanged.

8.7.4 Other Difference Sequences

An attacker who has control of the XOR difference sequences in A_i, B_i does not necessarily have great control over the XOR or modulo 2^{32} difference sequence that appears in the subkeys.

First, we must consider the context of a related-key differential attack. The attacker does not generally know all of the key bytes generating either A_i or B_i. Instead, he knows the XOR difference sequence in A_i and B_i.

Consider an A_i value with an XOR difference of δ. If the Hamming weight of δ is k, not including the high-order bit, then the best estimate for the XOR difference that ends up in the two subkey words for a given round generally has probability about 2^{-2k}. (Control of the A_i, B_i XOR difference sequence does not make controlling the subkey XOR differences substantially easier.)

Consider an A_i value with an XOR difference of δ. If the Hamming weight of δ is k, then the best estimate for the modulo 2^{32} difference of the two subkey words for a given round has probability about 2^{-k}.

This points out one of the difficulties in mounting any kind of successful related-key attack with nonzero A_i, B_i difference sequences. If an attacker can find a difference sequence for A_i, B_i that keeps $k = 3$, and needs to control the subkey differences for twelve rounds, he has a probability of about 2^{-72} of getting the most likely XOR subkey difference sequence, and about 2^{-36} of getting the most likely modulo 2^{32} difference sequence.

8.7.5 Probability of a Successful Attack With One Related-Key Query

We consider the use of the RS matrix in deriving S from M to be a powerful defense against related-key differential attacks, because it forces an attacker to keep at least five key generation S-boxes active. Our analysis suggests that any useful control of the subkey difference sequence requires that each active S-box in the attack have all four key bytes changed.

Further, our analysis suggests that, for nearly any useful difference sequence, each active S-box in the attack has a specific pair of defining key bytes it needs to work. An attacker specifying his key relation in terms of bytewise XOR has five pairs of sequences of four key bytes each, which he wants to get. This leaves him with a probability of a pair of keys with his desired relation actually leading to the desired attack of about 2^{-115}, which moves the attack totally outside the realm of practical attacks.

So long as an attacker is unable to improve this, either by finding a way to get useful difference sequences into the subkeys without having so many active key bytes, or by finding a way to mount related-key attacks with different S values for the different keys, we do not believe that any kind of related-key differential attack is feasible.

Note the implication of this: clever ways to control a couple extra rounds' subkey differences are not going to make the attacks feasible, unless they also allow the attacker to use far fewer active key bytes. For reference, note that with one altered key byte per active subkey generation S-box, the attacker ends up with a 2^{-39} probability that a pair of related keys will yield an attack; with two key bytes per active S-box, this increases to 2^{-78}; with three key bytes per active S-box, it increases to 2^{-117}. In practice, this means that any key relation requiring more than one byte of key changed per active S-box appears to be impractical.

8.7.6 Conclusions

Our analysis suggests that related-key differential attacks against the full Twofish are not workable. Note, however, that we have spent less time working on resistance to chosen key attacks, such as will be available to an attacker if Twofish is used in the straightforward way to define a hash function. For this reason, we recommend that more analysis be done before Twofish is used in the straightforward way as a hash function, and we note that it appears to be much more secure to use Twofish in this way with 128-bit keys than with 256-bit keys, despite the fact that this also slows the speed of a hash function down by a factor of two.

8.8 A Related-Key Attack on a Twofish Variant

8.8.1 Overview of the Attack

Results We define a partial chosen key attack on a 10-round Twofish variant without the pre- or post XORs. The attack requires us to control twenty selected bytes of the key, and to set them differently for a pair of related keys K, K^*. The remaining twelve bytes of the two keys, which are the same for both keys, are unknown to us; recovering them is the objective of the attack. This attack can be converted into a related-key differential attack, but it requires 2^{155} randomly selected related key pairs be tried before one will prove vulnerable to the attack.

The attack requires two keys chosen to have the desired relation. Under K, it requires about 1024 chosen plaintexts. Under K^*, it requires about 2^{32} chosen plaintexts, and about 2^{11} adaptive chosen plaintexts. The resulting attack recovers the twelve unknown key bytes with about 2^{32} effort.

How The Attack Works We start by requesting one plaintext encrypted under key K, and 2^{64} related plaintexts encrypted under K^*. We expect one of these to be a right pair, which means it will give the same result for its left ciphertext half under K^* as the original plaintext block did under key K. We request another plaintext from K and K^* to verify that this was really a right pair, rather than just an accidental occurrence. We now vary the plaintexts requested under K^*, to isolate and learn the differential properties of each S-box in g. This allows us to learn S. With knowledge of the active twenty key bytes of the attack, and knowledge of S, we are able to learn the remaining twelve bytes of key, and thus to break the cipher.

8.8.2 Finding a Key Pair

The first step in the attack is finding a pair of keys, K, K^*, with some fixed XOR relationship, such that we get a useful subkey difference sequence, while leaving the S key unchanged. We described above how such keys can exist. Here, we assume the existence of some pair of keys with a subkey XOR difference sequence zero in rounds 1..8 but with nonzero XOR subkey differences in rounds 0 and 9.

Structure of the Key Pair Based on the discussion above, we assume that the keys K, K^* differ in twenty bytes, with the changed bytes selected in such a way to leave S unchanged between the two keys, and to change all four key bytes used to define five of the S-boxes used for subkey generation. We note that a random pair of keys chosen to have this property has only a 2^{-155} probability of being the pair of keys that will generate our desired subkey difference sequence. This makes this attack impractical as a differential related-key attack, though an attacker who is able to control the active twenty bytes of key involved in a pair of keys could mount the attack for a chosen pair of keys. (That is, the attacker could choose the values of the active twenty bytes of both K and K^*, while remaining ignorant of the other twelve bytes of K and K^*, which must be identical for both keys. The attacker would then attempt to learn those remaining twelve bytes of key.)

Subkey Differences We actually care only about the subkey difference in the round subkey words. Recall that the subkey words are generated from a pair of 32-bit words, which are generated from the subkey generation S-boxes. Let's call these values U, V and U^*, V^*. We don't know any of U, V, U^*, V^*, but we do know $U' = U \oplus U^*$ and $V' = V \oplus V^*$.

8.8.3 Choosing the Plaintexts to Request

From U', V', it is possible to generate a set of about 2^{32} XOR difference values that we can expect to see result from the changed subkeys when they're used in the F function in the first round. We thus do the following:

1. Request the encryption of some fixed 128-bit block A, B, C, D under key K.

2. For each of the 2^{32} different 64-bit XOR difference values, X_i, Y_i, we might expect from U', V', request the encryption of 128-bit block $A, B, C \oplus X_i, D \oplus Y_i$).

3. Consider only those ciphertexts from the second step that lead to the same right half of ciphertext as the encryption under K in the first step. We may need to request one more encryption under both K and K^*, if we have more than one such ciphertext. The goal is to determine the output XOR from the first round's F function when the input is A, B, under the two keys. (The only difference in the output is caused by the difference in round subkeys.)

8.8.4 Extracting the Key Material

At this point, we know the single XOR difference from the F function. We can now learn the S-boxes in the g function. To do this, we replace the high-order byte of A in the right pair with 256 different values. Thus, we request the encryption of A_j, B, C, D under K^*. This causes one of 256 possible XOR differences in the g function output. We expect to have to try about 512 requests with different XOR difference values X_i, Y_i, where these values are derived from the X_i, Y_i values that generated the original right pair based on the possible XOR differences in g. We thus request 512 plaintexts of the form $A_j, B, C \oplus X_i, D \oplus Y_i$) to be encrypted under K^*. One of these, we expect to be a right pair, which we can recognize, again, by the fact that the right half of

the ciphertext is the same for A_j, B, C, D encrypted under K, and for $A_j, B, C \oplus X_i, D \oplus Y_i$ encrypted under K^*.

After carrying this out for each of the A_j, we have good information about the XOR differences being generated from g by the changes between A and A_j as input. In particular, we will note the fourteen cases where we probably got 8-bit Hamming weights in the output XOR difference of g. This information should be enough to brute-force this S-box, $s3$. That is, we try all 2^{32} possible S-boxes for $s3$, and expect one to be the best fit for our data.

We repeat the attack for each of the other bytes in A, so that we recover all four S-boxes in g. When we know these S-boxes, this means we know S. Note that at the beginning of the attack, we already know twenty of the thirty-two key bytes used. More importantly, due to the structure of the RS matrix used to derive S from the raw key bytes, we know all but three bytes used to derive each four byte "row" of values in S. This allows us to guess and check the remaining key bytes three bytes at a time, thus recovering the entire key.

8.9 Side-Channel Cryptanalysis and Fault Analysis

Resistance to these attacks was not part of the AES criteria, and hence not a major concern in this design. However, we do have these comments to make on the design.

Side-channel cryptanalysis [KSWH98b] uses information about the cipher in addition to the plaintext or ciphertext. Examples include timing [Koc96], power consumption (including differential power analysis [Koc98]), NMR scanning, and electronic emanations.[21] With many algorithms it is possible to reconstruct the key from these side channels. While total resistance to side-channel cryptanalysis is probably impossible, we note that Twofish executes in constant time on most processors.

Fault analysis [BDL97, BS97] can be used to successfully cryptanalyze this cipher. Again, we believe that total resistance to fault analysis is an impossible design constraint for a cipher. The resistance to fault analysis of any block cipher can be improved using classical fault tolerance techniques.

[21] The NSA refers to this particular side channel as "TEMPEST."

8.10 Attacking Simplified Twofish

8.10.1 Twofish with Known S-boxes

As discussed above, our differential attack on Twofish with fixed S-boxes works for 6-round Twofish, and requires only 2^{67} effort. A 7-round differential attack requires 2^{131} effort.

Additionally, much of Twofish's related-key attack resistance comes from the derivation of the S-boxes, so a related-key attack is much easier against a known S-box variant.

8.10.2 Twofish Without Round Subkeys

We attack a Twofish variant without any subkeys, thus whose whole key material is in the key-dependent S-boxes. The attack requires about 2^{33} chosen plaintexts; it breaks the Twofish variant with about 2^{36} effort, even when the cipher has a 128-bit key. (Note that this is the amount of key material used to define the S-boxes in normal Twofish with a 256-bit key.)

It is obvious that a Twofish variant with fixed S-boxes and no subkeys would be insecure—there would be no key material injected. We develop a slide attack on Twofish with any number of rounds, and a 128-bit key. (The Twofish key would be 256 bits, but since we never use more than 128 bits of key in the key-dependent S-boxes, the effective key size is 128 bits.)

Note that this attack would not work if we used round subkeys that were simply counter values, as would happen if we used the identity permutation for all the key-scheduling S-boxes. We are not currently aware of any attack on such a Twofish variant.

Overview of the Attack In a slide attack, we attempt to find a pair of plaintexts, (P, P^*), such that P^* has the same value as the intermediate value after one or more rounds of encrypting P. If we can find and expose this value, we gain direct access to the results of a single round of the cipher; we then attack that single round to recover the key.

Our attack has two parts:

1. We must first find a pair of plaintexts, (P, P^*), such that P^* is the result of encrypting P with one round.
2. We use this pair to derive four relations on g and thus determine the specific S-boxes used in g. This yields the effective key of the cipher.

Finding Our Related Pair of Texts Twofish is a Feistel cipher, operating on two 64-bit halves in each block. We use the representation where the left half of the block is the input to the Feistel function, and the halves are swapped after each round. Let (L_0, R_0) represent a given pair of 64-bit halves input into the cipher, and let (L_N, R_N) represent the resulting ciphertext. To mount this attack, we need to find some (L_0, R_0, L_1) such that L_1 is the value of the left half of the block after the first round. To test whether we have such a pair, we can try encrypting (L_0, R_0) and (L_1, L_0). When we have a triple with the desired properties, we get (L_N, R_N) from the first encryption, and (L_{N+1}, L_N) from the second encryption.

We use a trick by Biham [Bih94] to get such a pair of plaintexts with only 2^{33} chosen plaintexts: we choose a fixed L_0, and encrypt it with 2^{32} randomly selected R_0 values as (L_0, R_0). We then encrypt the same L_0 with 2^{32} random R_1 values, as (R_1, L_0). By the birthday paradox, we expect to see one pair of values for which $R_1 = R_0 \oplus F(L_0)$. To find this pair, we sort the ciphertexts from the first batch of encryptions on their left halves, and the ciphertexts from the second batch of encryptions on their right halves. When we find a match, the probability is good that we have a pair with the desired property.

Extracting g Values Once we have this triple (L_0, R_0, L_1), we also have two relations on F: $F(L_0) = R_0 \oplus L_1$ and $F(L_N) = R_N \oplus L_{N+1}$. Note, however, that we do not yet have direct g values. Instead, we have two instances of the results of combining two g outputs with a PHT. Since we have the actual PHT output values, we can simply undo the PHT, yielding relations on g. Our pair has given us two relations on F, and thus four relations on g.

Extracting the S-boxes The g outputs are the result of applying four key-dependent S-boxes to the input bytes, and then combining those bytes with an MDS matrix multiply. Since the MDS multiply is invertible, we invert it to get back the actual S-box outputs for all four different values. If those values are all different, then we have four bytes of output from each S-box. We can try all 2^{32} possible key input values for each S-box, and see which ones are consistent with the results; for most sets byte values, only one or two S-boxes will match them. We thus learn each key-dependent S-box in g, perhaps with a couple of alternative values. We try all possible alternatives against any plaintext/ciphertext values we have available, and very quickly recover the correct S-boxes. Since this is the only key material in this Twofish variant, the attack is done.

8.10.3 Twofish with Non-bijective S-boxes

We decided early in the design process to use purely bijective S-boxes. One rationale was to ensure that the 64-bit round function be bijective. That way, iterative 2-round differential characteristics cannot exist; when they do exist, they often result in the highest-probability multi-round characteristic, so avoiding them should help to reduce the risk of a successful differential attack. Also, attacks based on non-surjective round functions [BB95, RP95b, RPD97, CWSK98] are sure to fail when the 64-bit Feistel round function is bijective.[22]

We argue here that this was a good design decision, by showing that a Twofish variant which uses non-bijective S-boxes is likely to be much easier to break.

Observe that when q_0 and q_1 are non-bijective, their 3-way composition into s_i is likely to be even more non-surjective than either of q_0 or q_1 on its own. It is easily seen that the expected size of the range of a random function on 8 bits (such as q_0 or q_1) is $r_1 = 1 - (1 - 1/256)^{256} \approx 1 - e^{-1} \approx 0.632$. When we compose such a function twice, its expected range becomes $r_2 = 1-(1-1/256)^{256 r_1} \approx 1-e^{-r_1} \approx 0.469$; and the expected size of the range of a 3-way composition will be $r_3 \approx 1 - e^{-r_2} \approx 0.374$. In other words, we expect that only about $96 = 0.374 \times 256$ of all possible 8-bit values can appear as the output of S. Therefore, the output of the h function can attain only about $2^{26.3} \approx 96^4$ possible values, and the 64-bit Feistel function can attain only about $2^{52.6}$ of all 2^{64} possible outputs. This is certainly a rather large certificational weakness. (This gets worse when the key size grows, since the number of compositions of the q functions gets larger.)

We point out a serious differential attack when using non-bijective S-boxes. Consider the probability $p_{\Delta x}$ that a given input difference Δx yields a collision in the S-box output; i.e., $\Delta x \mapsto 0$. Let $p = \sum_{\Delta x \neq 0} p_{\Delta x}/255$ be the average probability over all non-zero input differences, and let $m = \max_{\Delta x \neq 0} p_{\Delta x}$ be the maximum probability over all non-zero input differences. We have $\mathbf{E}\, p = 3/256$; also, $\Pr(p \geq 2/256) \approx 0.78$, $\Pr(p \geq 10/256) \approx 0.02$, and $\Pr(p \geq 16/256) \approx 0.0002$. As for the distribution of m, empirically $\mathbf{E}\, m \approx 11.7/256$; experiments suggest $\Pr(m \geq 10/256) \approx 0.975$, $\Pr(m \geq 16/256) \approx .04$, and $\Pr(m \geq 22/256) \approx 0.0002$.

[22] Vaudenay's attack on Blowfish took advantage of non-bijectivity in the Blowfish round function [Vau96a].

Consider a Twofish variant with non-bijective S-boxes and no rotations. We obtain a 2-round iterative differential characteristic with probability m, and thus a 13-round differential characteristic with probability m^6. We find that, for 97.5% of the keys, one can break the variant with about 2^{28} chosen plaintexts; for 4% of the keys, it can be broken with 2^{24} chosen plaintexts; and for a class of weak keys consisting of 0.02% of the keys, the variant cipher can be broken with roughly 2^{21} chosen plaintexts. (Of course, one can trade off the number of texts needed against the size of the weak key class; the figures given are just examples of points on a smooth curve.)

Next we consider a Twofish variant with non-bijective S-boxes, but with all rotations left intact. If we look at any 2-round differential characteristic whose first round is the trivial characteristic and whose second round involves just one active S-box, we expect its probability to be about p. One difficulty is that the rotations prevent us from finding an iterative 2-round characteristic. However, we can certainly piece together 6.5 different 2-round differential characteristics (each of the right form so it will have probability about p) to find a 13-round characteristic with expected probability p^6. (The latter probability can probably be improved to about $3p^6$, due to the extra degrees of freedom, but as we are only doing a back-of-the-envelope estimate anyway, we will omit these considerations.) Thus, we can find a differential attack that succeeds with about 2^{39} chosen plaintexts for the majority (about 78%) of the keys; also, for 2% of the keys, this variant can be broken with 2^{28} chosen plaintexts; and for a class of weak keys consisting of 0.02% of the keys, the variant cipher can be broken with roughly 2^{24} chosen plaintexts.

This analysis clearly shows the value of bijective S-boxes in stopping differential cryptanalysis. Also, it helps motivate the benefit of the rotations: they make short iterative characteristic much harder to find, thereby conferring (we hope) some additional resistance against differential attacks.

9 Trap Doors in Twofish

We assert that Twofish has no trap doors. As designers, we have made every effort to make Twofish secure against all known (and unknown) cryptanalyses, and we have made no effort to build in a secret way of breaking Twofish. However, there is no way to prove this, and not much reason for anyone to believe us. We can offer some assurances.

In this paper, we have outlined all of the design elements of Twofish and our justifications for including them. We have explained, in great detail, how we chose Twofish's "magic constants": the RS code, q_0, q_1, and the MDS matrix. There are no mysterious design elements; everything has an explicit purpose. Moreover, we feel that the use of key-dependent S-boxes makes it harder to install a trap door into the cipher. As difficult as it is to create a trap door for a particular set of carefully constructed S-boxes, it is much harder to create one that works with all possible S-boxes or even a reasonably useful subset of them (of relative size 2^{-20} or so).

Additionally, any trap door would have to survive 16 rounds of Twofish. It would have to work even though there is almost perfect diffusion in each round. It would have to survive the pre- and post-whitening. These design elements have long been known to make any patterns difficult to detect; trap doors would be no different.

None of this constitutes a proof. Any reasonable proof of general security for a block cipher would also prove $\mathbf{P} \neq \mathbf{NP}$. Rather than outlining the proof here, we would likely skip the AES competition and go collect our Fields Medal.

However, we have made headway towards a philosophical proof. Assume for a moment that, despite the difficulties listed above, we did manage to put a trap door into Twofish. This would imply one of two things:

One, that we have invented a powerful new cryptanalytic attack and have carefully crafted Twofish to be resistant to all known attacks but vulnerable to this new one. We cannot prove that this is not true. However, we can point out that as cryptographers we would achieve much more fame and glory by publishing our powerful new cryptanalytic attack. In fact, we would probably publish it along with this paper, making sure Twofish is immune so that we can profitably attack the other AES submissions.

The other possibility is that we have embedded a trap door into the Twofish magic constants and then transformed them by some means so that finding them would be a statistical impossibility (see [Har96] for some discussion of this possibility). The resulting construction would seem immune to current cryptanalytic techniques, but we as designers would know a secret transformation rule that we could apply to facilitate cryptanalysis. Again, we cannot prove that this is not true. However, it has been shown that this type of cipher, called a "master-key cryptosystem," is equivalent to a public-key cryptosystem [BFL96]. Again, as cryptographers we

would achieve far greater recognition by publishing a public-key cryptosystem that is not dependent on factoring [RSA78] or the discrete logarithm problem [DH76, ElG85, NIST94]. And the resulting algorithm's dual capabilities as both a symmetric and public-key algorithm would make it far more flexible than the AES competition.

There is a large gap between a weakness that is exploitable in theory and one that is exploitable in practice. Even the best attack against DES (a linear-cryptanalysis-like attack combining quadratic approximations and a multiple-approximation method) requires just under 2^{43} plaintext/ciphertext blocks [SK98], which is equivalent to about 64 terabytes of plaintext/ciphertext encrypted under a single key. A useful trap door would need to work with much less plaintext—a few thousand blocks—or it would have to reduce the effective keyspace to something on the order of 2^{72}. We believe that, given the quality of the public cryptanalytic research community, it would be impossible to put a weakness of this magnitude into a block cipher and have it remain undetected through the AES process. And we would be foolish to even try.

10 When is a Cipher Insecure?

More and more recent ciphers are being defined with a variable number of rounds: e.g., SAFER-K64 [Mas94], RC5, and Speed [Zhe97]. This means that it is impossible to categorically state that a given cipher construction is insecure: there might always be a number of rounds n for which the cipher is still secure. However, while this might theoretically be true, this is not a useful engineering definition of "secure." After all, the user of the cipher actually has to choose how many rounds to use. In a performance-driven model, it is useful to compare ciphers of equal speed, and compare their security, or compare ciphers of equal security and compare their speeds. For example, FEAL-32 is secure against both differential and linear attacks. Its speed is 65 clock cycles per byte of encryption, which makes it less desirable than faster, also secure, alternatives.

With that in mind, Table 9 gives performance metrics for block and stream ciphers on the Pentium processor.[23]

11 Using Twofish

11.1 Chaining Modes

All standard block-cipher chaining modes work with Twofish: CBC, CFB, OFB, counter [NBS80]. We are aware of no problems with using Twofish with any commonly used chaining mode. (See [Sch96] for a detailed comparison of the various modes of operation.) A cryptanalyst considering OFB-, CFB-, or CBC-mode encryption with Twofish may collapse the pre- and post-XOR of key material into a single XOR, but does not appear to benefit much from this.

11.2 One-Way Hash Functions

The most common way of using a block cipher as a hash function is a Davies-Meyer construction [Win84]:

$$H_i = H_{i-1} \oplus E_{M_i}(H_{i-1})$$

There are fifteen other variants [Pre93]. We believe that Twofish can be used securely in any of these formats; note, however, that the key schedule has been analyzed mainly for related-key attacks, not for the class of chosen-key attack that hash functions must resist. Additionally, the 128-bit block size makes straightforward use of the Davies-Meyer construction useful only when collision-finding attacks can be expected to be unable to try 2^{64} trial hashes to find a collision.

As keys that are non-standard sizes have equivalent keys that are longer, any use of Twofish in a Davies-Meyer construction must ensure that only a single key length is used.

11.3 Message Authentication Codes

Any one-way hash function can be used to build a message authentication code using existing techniques [BCK96]. Again, we believe Twofish's strong key schedule makes it very suitable for these constructions.

11.4 Pseudo-Random Number Generators

Twofish can also be used as a primitive in a pseudo-random number generator suitable for generating session keys, public-key parameters, protocol nonces, and so on [Plu94, KSWH98a, Gut98, KSWH98c].

[23] These metrics are based on theoretical analyses of the algorithms and actual hand-tooled assembly-language implementations [SW97, PRB98].

Algorithm	Key Length	Width (bits)	Rounds	Cycles	Clocks/Byte
Twofish	variable	128	16	8	18.1
Blowfish	variable	64	16	8	19.8
Square	128	128	8	8	20.3
RC5-32/16	variable	64	32	16	24.8
CAST-128	128	64	16	8	29.5
DES	56	64	16	8	43
Serpent	128, 192, 256	128	32	32	45
SAFER (S)K-128	128	64	8	8	52
FEAL-32	64, 128	64	32	16	65
IDEA	128	64	8	8	74
Triple-DES	112	64	48	24	116

Table 9: Performance of different block ciphers (on a Pentium)

11.5 Larger Keys

Even though it would be straightforward to extend the Twofish key schedule scheme to larger key sizes, there is currently no definition of Twofish for key lengths greater than 256 bits. We urge caution in trying to extend the key length; our experience with Twofish has taught us that extending the key length can have important security implications.

11.6 Additional Block Sizes

There is no definition of Twofish for block lengths other than 128 bits. While it may be theoretically possible to extend the construction to larger block sizes, we have not evaluated these constructions at all. We urge caution in trying to extend the block size; many of the constructions we use may not scale well to 256 bits, 512 bits, or larger blocks.

11.7 More or Fewer Rounds

Twofish is defined to have 16 rounds. We designed the key schedule to allow natural extensions to more or fewer rounds if and when required. We strongly advise against reducing the number of rounds. We believe it is safe to increase the number of rounds, although we see no need to do so.

11.8 Family Key Variant: Twofish-FK

We often see systems that use a proprietary variant of an existing cipher, altered in some hopefully security-neutral way to prevent that system from interoperating with standard implementations of the cipher. A family key is a way of designing this into the algorithm: each different family key is used to define a different variant of the cipher. In some sense, the family key is like an additional key to the cipher, but in general, it is acceptable for the family key to be very computationally expensive to change. We would expect nearly all Twofish implementations that used any family key to use only one family key.

Our goals for the family key algorithm are as follows:

- No family key variant should be substantially weaker than the original cipher.
- Related-key attacks between different unknown but related family keys, or between a known family key and the original cipher, should be hard to mount.
- The family key should not merely reorder the set of 128-by-128-bit permutations provided by the cipher; it should change that set.

A Twofish family key is simply a 256-bit random Twofish key. This key, FK, is used to derive several blocks of bits, as follows:

1. Preprocessing Step:

   This step is done once per family key.

   (a) $T_0 = FK$.
   (b) $T_1 = (E_{FK}(0), E_{FK}(1)) \oplus FK$.
   (c) $T_2 = E_{FK}(2)$.
   (d) $T_3 = E_{FK}(3)$.
   (e) $T_4 =$ First 8 bytes of $E_{FK}(4)$.

   Note that, using our little-endian convention, the small integers used as plaintext inputs should occur in the first plaintext byte.

2. Key Scheduling Step: This step is done once per key used under this family key.

   (a) Before subkey generation, T_0 is XORed into the key, using as many of the leading bytes of T_0 as necessary.

   (b) After subkey generation, T_2 is XORed into the pre-XOR subkeys, T_3 is XORed into the post-XOR subkeys, and T_4 is XORed into each round's 64-bit subkey block.

   (c) Before the cipher's S-boxes are derived, T_1 is XORed into the key. Once again, we use as many of the leading bytes of T_1 as we need. Each byte of the key is then passed through the byte permutation q_0, and the result is passed through the RS matrix to get S. Note that the definition of T_1 means that the effect of the first XOR of FK into the key is undone.

Note the properties of the alterations made by any family key:

- The keyspace is simply permuted for the initial subkey generation.
- The subkeys are altered in a simple way. However, there is strong evidence that this alteration cannot occur by changing keys, based on the same difference sequence analysis used in discussing related-key attacks on Twofish.
- The S-boxes used in the cipher are altered in a powerful way, but one which does not alter the basic required properties of the key schedule. Getting no changes in the S-boxes used in the cipher still requires changing at least five bytes of key, and those five bytes of key must change five of the S-boxes used for subkey generation.

11.8.1 Analysis

Effects of Family Keys on Cryptanalysis We are not aware of any substantial difference in the difficulty of cryptanalyzing the family key version of the cipher rather than the regular version. The cipher's operations are unchanged by the family key; only subkey and S-box generation are changed. However, they are changed in simple ways; the S-boxes are generated in exactly the same way as before, but the key material provided to them is processed in a simple way first; the round subkeys are generated in the same way as before (again, with the key material processed in a simple way first), and then have a constant 64-bit value XORed in. Related-key attacks of the kind we have been able to consider are made slightly harder, rather than easier, by this 64-bit value. The new constants XORed into the pre- and post-XOR subkeys simply permute the input and output space of the cipher in a very simple way.

Related Keys Across Family Keys Related-key attacks under the same family key appear, as we said above, to be at least as hard as related-key attacks in normal Twofish. There is still the question, however, of whether there are interesting related keys across different family keys. It is hard to see how such related keys would be used, but the analysis may be worth pursuing anyway.

An interesting question, from the perspective of an attacker, is whether there are pairs of keys that give identical subkeys except for the constant values XORed into them, and that also give identical S-boxes. By allowing an attacker to put a constant XOR into all round subkeys, such pairs of keys would provide a useful avenue of attack.

This can be done by finding a pair of related family keys, FK, FK^*, which are identical in their first 128 bits, and a pair of 128-bit cipher keys, M, M^*, such that M generates the same set of S-boxes with FK that M^* does with FK^*. For a random pair of M, M^* values, this has probability 2^{-64} of happening. Thus, an attacker given complete control over M, M^* and knowledge of FK, FK^* can find such a pair of keys and family keys. However, this does not seem to translate into any kind of clean related-key attack; the attacker must actually choose the specific keys used.

We do not consider this to be a valid attack against the system. In general, related-key attacks between family keys seem unrealistic, but one which also requires the attacker to be able to choose specific key values he is trying to recover is also pointless.

A Valid Related-key Attack An attacker can also try to find quadruples FK, FK^*, M, M^* such that the subkey generation and the S-box generation both get identical values. This requires that

$$\begin{aligned} T_0 \oplus M &= T_0^* \oplus M^* \\ T_1 \oplus M &= T_1^* \oplus M^* \end{aligned}$$

If FK, FK^* have the property that

$$T_0 \oplus T_0^* = T_1 \oplus T_1^* = \delta$$

then related-key pairs $M, M \oplus \delta$ will put a fixed XOR difference into every round subkey, and may allow

some kind of related-key attack. Again, we do not think this is an interesting attack; the attacker must force the family keys to be chosen this way, since the probability that any given pair of family keys will work this way is (for 128-bit cipher keys) 2^{-128}. We do not expect such relationships to occur by chance until about 2^{64} family keys are in use.

12 Historical Remarks

Twofish originated from an attempt to take the original Blowfish design and modify it for a 128-bit block. We wanted to leverage the speed and good diffusion of Blowfish, while also improving it where we could. We wanted the new cipher to have a bijective F function, a much more efficient key schedule, and to be implementable in custom hardware and smart cards in addition to 32-bit processors (i.e., have smaller tables). And we wanted it to be even faster than Blowfish (per byte encrypted), if possible.

Initial thoughts were to have the Blowfish round structure operate on the four 32-bit subblocks in a circular structure, but there were problems getting the diffusion to work in both the encryption and decryption directions. Having two parallel Blowfish round functions and letting them interact via a two-dimensional Feistel structure ran into the same problems. Our solution was to have a single Feistel structure with two Blowfish-like 32-bit round functions and to combine them using a PHT (an idea stolen from SAFER). This idea also provided nearly complete avalanche during the round.

Round subkeys are required to avoid slide attacks against identical round functions. We used addition instead of XOR to take advantage of the Pentium LEA opcode and implement them in effectively zero time.

We used 8-by-8-bit S-boxes and an MDS matrix (an idea stolen from Square, although Square uses a single fixed S-box) instead of random 8-by-32-bit S-boxes, both to simplify the key schedule and ensure that the g function is bijective. This construction ensured that Twofish would be efficient on 32-bit processors (by precomputing the S-boxes and MDS matrix into four 8-by-32-bit S-boxes) while still allowing it to be computed on the fly in smart cards. And since our MDS matrix is only ever computed in one direction, we did not have to worry about the matrix's efficiency in the reverse direction (which Square had to consider).

The construction also gave us considerable performance flexibility. We worked hard to keep this flexibility, so implementers would have a choice of how much key pre-processsing to do depending on the amount of plaintext to be encrypted. And we tried to maintain these tradeoffs for 32-bit microprocessors, 8-bit microprocessors, and custom hardware.

Since one goal was to be able to keep the complete design in our heads, any complication that did not have a clear purpose was deleted. Additional complications we chose not to introduce were key-dependent MDS matrices or round-dependent variations on the byte ordering and the PHT. We also toyed with the idea of swapping 32-bit words within the Feistel halves (something we called the "twist"), but abandoned it because we saw no need for the additional complexity.

We did keep in the one-bit rotations of the target block, primarily to reduce the vulnerability to any attack based on byte boundaries. The particular manifestation of this one-bit rotation was due to a combination of performance and cryptanalytic concerns.

We considered using all the key bytes, rather than just half, to define the key-dependent S-boxes. Unfortunately, this made the key setup time for high-end machines unreasonably large, and also made encryption too slow on low-end machines. By dropping this to half the key bits, these performance figures were improved substantially. By carefully selecting how the key bytes are folded down to half their size before being used to generate the cipher's S-boxes, we were able to ensure that pairs of keys with the same S-boxes would have very different subkey sequences.

The key schedule gave us the most trouble. We had to resist the temptation to build a cryptographic key schedule like the ones used by Blowfish, Khufu, and SEAL, because low-end implementations needed to be able to function with minimal additional memory and, if necessary, to compute the subkeys as needed by the cipher. However, a simple key schedule can be an important weak point in a cipher design, leaving the whole cipher vulnerable to partial-key guessing attacks, related-key attacks, or to attacks based on waiting for especially weak keys to be selected by a user. Though our final key schedule is rather complex, it is conceptually much simpler than many of our intermediate designs. Reusing many of the primitives of the cipher (each round's subkeys are generated by a computation nearly identical to the one that goes on inside the round function, except for the specific inputs involved) made it possible to investigate properties of both the key schedule and of the cipher at the same time.

We spent considerable time choosing q_0 and q_1. Since these provide the primary non-linearity in the cipher, they had to be strong. We wanted to be able to construct them algebraically, for applications where storing 512 bytes of fixed tables was not possible. In the end, we built permutations from random parameters, and tested the permutations against our required criteria.

And finally, we cryptanalyzed Twofish. We cryptanalyzed and cryptanalyzed and cryptanalyzed, right up to the morning of the submission deadline. We're still cryptanalyzing; there's no stopping.

13 Conclusions and Further Work

We have presented Twofish, the rationale behind its design, and the results of our initial cryptanalysis. Design and cryptanalysis go hand in hand—it is impossible to do one without the other—and it is only in the analysis that the strength of an algorithm can be demonstrated.

During the design process, we learned several lessons about cipher design:

- The encryption algorithm and key schedule must be designed in tandem; subtle changes in one affect the other. It is not enough to design a strong round function and then to graft a strong key schedule onto it (unless you are satisfied with an inefficient and inelegant construction, like Blowfish has); both must work together.
- There is no such thing as a key-dependent S-box, only a complicated multi-stage nonlinear function that is implemented as a key-dependent S-box for efficiency.
- Keys should be as short as possible. It is much harder to design an algorithm with a long key than an algorithm with a short key. Throughout our design process, we found it easier to design and analyze Twofish with a 128-bit key than Twofish with a 192- or 256-bit key.
- Build a cipher with strong local encryption and let the round function handle the global diffusion. Designing Twofish in this manner made it very hard to mount any statistical cryptanalytical attacks.
- Consider performance at every stage of the design. Having a code optimization guru on our team from the beginning drastically changed the way we looked at design tradeoffs, to the ultimate benefit of Twofish.
- Analysis can go on forever. If the submission deadline were not on 15 June 1998, we would still be cryptanalyzing and tweaking Twofish.

We believe Twofish to be an ideal algorithm choice for AES. It is efficient on large microprocessors, smart cards, and dedicated hardware. The multiple layers of performance tradeoffs in the key schedule make it suitable for a variety of implementations. And the attention to cryptographic detail in the design—both the encryption function and the key schedule—make it suitable as a codebook, output-feedback and cipher-feedback stream cipher, one-way hash function, and pseudo-random number generator.

We welcome any new cryptanalysis from the cryptographic community. We plan on continuing to evaluate Twofish all through the AES selection process. Specifically:

- Whether the number of rounds can safely be reduced. At this point our best non-related-key attack—a differential attack—can only break five rounds. If no better attacks are found after a few years, it may be safe to reduce the number of rounds to 14 or even 12. Twelve-round Twofish can encrypt and decrypt data at about 250 clock cycles per block on a Pentium, Pentium Pro, and Pentium II.
- Whether there are alternative fixed tables that increase security. We have chosen both the MDS matrix and the fixed permutations, q_0 and q_1, to meet our mathematical requirements. In the event we find better constants that make Twofish even harder to cryptanalyze, we may want to revise the algorithm.
- Whether we can define a Twofish variant with fixed S-boxes. This variant would have a faster key-setup time than the algorithm presented here—about 1200 clock cycles instead of 7800 clock cycles on a Pentium Pro—and the same encryption and decryption speeds. Further research is required on what the fixed S-boxes would look like, and how much data could be safely encrypted with this variant.
- Whether we can improve our lower bound on the complexity of a differential attack.

Developing Twofish was a richly rewarding experience, and one of our most satisfying cryptographic projects to date. We look forward to the next phase of the AES selection process.

14 Acknowledgments

The authors would like to thank Carl Ellison, Paul Kocher, and Randy Milbert, who read and commented on drafts of the paper, and Beth Friedman, who copyedited the (almost) final version of the paper. Additionally, the authors would like to thank NIST for initiating the AES process, and Miles Smid, Jim Foti, and Ed Roback for putting up with a never-ending steam of questions and complaints about its details. This work has been funded by Counterpane Systems and Hi/fn Inc.

References

[AB96a] R. Anderson and E. Biham, "Two Practical and Provably Secure Block Ciphers: BEAR and LION," *Fast Software Encryption, Third International Workshop Proceedings*, Springer-Verlag, 1996, pp. 113–120.

[AB96b] R. Anderson and E. Biham, "Tiger: A Fast New Hash Function," *Fast Software Encryption, Third International Workshop Proceedings*, Springer-Verlag, 1996, pp. 89–97.

[Ada97a] C. Adams, "Constructing Symmetric Ciphers Using the CAST Design Procedure," *Designs, Codes and Cryptography*, v.12, n.3, Nov 1997, pp. 71–104.

[Ada97b] C. Adams, "DES-80," *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, School of Computer Science, Carleton University, 1997, pp. 160–171.

[AGMP96] G. Álvarez, D. De la Guia, F. Montoya, and A. Peinado, "Akelarre: A New Block Cipher Algorithm," *Workshop on Selected Areas in Cryptography (SAC '96) Workshop Record*, Queens University, 1996, pp. 1–14.

[Anon95] Anonymous, "this looked like it might be interesting," sci.crypt Usenet posting, 9 Aug 1995.

[AT93] C.M. Adams and S.E. Tavares, "Designing S-boxes for Ciphers Resistant to Differential Cryptanalysis," *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, Rome, Italy, 15–16 Feb 1993, pp. 181–190.

[BAK98] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A New Block Cipher Proposal," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 222–238.

[BB93] I. Ben-Aroya and E. Biham, "Differential Cryptanalysis of Lucifer," *Advances in Cryptology — CRYPTO '93 Proceedings*, Springer-Verlag, 1994, pp. 187–199.

[BB94] E. Biham and A. Biryukov, "How to Strengthen DES Using Existing Hardware," *Advances in Cryptology — ASIACRYPT '94 Proceedings*, Springer-Verlag, 1994, pp. 398–412.

[BB95] E. Biham and A. Biryukov, "An Improvement of Davies' Attack on DES," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 461–467.

[BB96] U. Blumenthal and S. Bellovin, "A Better Key Schedule for DES-Like Ciphers," *Pragocrypt '96 Proceedings*, 1996, pp. 42–54.

[BCK96] M. Bellare, R. Canetti, and H. Karwczyk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 1–15.

[BDL97] D. Boneh, R.A. DeMillo, and R.J. Lipton "On the Importance of Checking Cryptographic Protocols for Faults," *Advances in Cryptology — EUROCRYPT '97 Proceedings*, Springer-Verlag, 1997, pp. 37–51.

[BDR+96] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Weiner, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security," Jan 1996.

[BFL96] M. Blaze, J. Feigenbaum, and F. T. Leighton, *Master-Key Cryptosystems*, DIMACS Technical Report 96-02, Rutgers University, Piscataway, 1996.

[Bih94] E. Biham, "New Types of Cryptanalytic Attacks Using Related Keys," *Journal of Cryptology*, v. 7, n. 4, 1994, pp. 229–246.

[Bih95] E. Biham, "On Matsui's Linear Cryptanalysis," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 398–412.

[Bih97] E. Biham, "A Fast New DES Implementation in Software," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 260–271.

[BK98] A. Biryukov and E. Kushilevitz, "Improved Cryptanalysis of RC5," *Advances in Cryptology — EUROCRYPT '98 Proceedings*, Springer-Verlag, 1998, pp. 85–99.

[BKPS93] L. Brown, M. Kwan, J. Pieprzyk, and J. Seberry, "Improving Resistance to Differential Cryptanalysis and the Redesign of LOKI," *Advances in Cryptology — ASIACRYPT '91 Proceedings*, Springer-Verlag, 1993, pp. 36–50.

[BPS90] L. Brown, J. Pieprzyk, and J. Seberry, "LOKI: A Cryptographic Primitive for Authentication and Secrecy Applications," *Advances in Cryptology — AUSCRYPT '90 Proceedings*, Springer-Verlag, 1990, pp. 229–236.

[Bro98] L. Brown, "Design of LOK97," draft AES submission, 1998.

[BS92] E. Biham and A. Shamir, "Differential Cryptanalysis of Snefru, Khafre, REDOC II, LOKI, and Lucifer," *Advances in Cryptology — CRYPTO '91 Proceedings*, Springer-Verlag, 1992, pp. 156–171.

[BS93] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag, 1993.

[BS95] M. Blaze and B. Schneier, "The MacGuffin Block Cipher Algorithm," *Fast Software Encryption, Second International Workshop Proceedings*, Springer-Verlag, 1995, pp. 97–110.

[BS97] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," *Advances in Cryptology — CRYPTO '97 Proceedings*, Springer-Verlag, 1997, pp. 513–525.

[CDN95] G. Carter, E. Dawson, and L. Nielsen, "DESV: A Latin Square Variation of DES," *Proceedings of the Workshop on Selected Areas in Cryptography (SAC '95)*, Ottawa, Canada, 1995, pp. 158–172.

[CDN98] G. Carter, E. Dawson, and L. Nielsen,, "Key Schedules of Iterative Block Ciphers," *Third Australian Conference, ACISP '98*, Springer-Verlag, to appear.

[Cla97] C.S.K. Clapp, "Optimizing a Fast Stream Cipher for VLIW, SIMD, and Superscalar Processors," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 273–287.

[Cla98] C.S.K. Clapp, "Joint Hardware/Software Design of a Fast Stream Cipher," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 75–92.

[CM98] H. Chabanne and E. Michon, "JEROBOAM" *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 49–59.

[Cop94] D. Coppersmith, "The Data Encryption Standard (DES) and its Strength Against Attacks," *IBM Journal of Research and Development*, v. 38, n. 3, May 1994, pp. 243–250.

[Cop98] D. Coppersmith, personal communication, 1998.

[CW91] T. Cusick and M.C. Wood, "The REDOC-II Cryptosystem," *Advances in Cryptology — CRYPTO '90 Proceedings*, Springer-Verlag, 1991, pp. 545–563.

[CWSK98] D. Coppersmith, D. Wagner, B. Schneier, and J. Kelsey, "Cryptanalysis of TWOPRIME," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 32–48.

[Dae95] J. Daemen, "Cipher and Hash Function Design," Ph.D. thesis, Katholieke Universiteit Leuven, Mar 95.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD," *Fast Software Encryption, Third International Workshop Proceedings*, Springer-Verlag, 1996, pp. 71–82.

[DC98] J. Daemen and C. Clapp, "Fast Hashing and Stream Encryption with PANAMA," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 60–74.

[DGV93] J. Daemen, R. Govaerts, and J. Vandewalle, "Block Ciphers Based on Modular Arithmetic," *Proceedings of the 3rd Symposium on: State and Progress of Research in Cryptography*, Fondazione Ugo Bordoni, 1993, pp. 80–89.

[DGV94a] J. Daemen, R. Govaerts, and J. Vandewalle, "Weak Keys for IDEA," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 159–167.

[DGV94b] J. Daemen, R. Govaerts, and J. Vandewalle, "A New Approach to Block Cipher Design," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 18–32.

[DH76] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, v. IT-22, n. 6, Nov 1976, pp. 644–654.

[DH79] W. Diffie and M. Hellman, "Exhaustive Cryptanalysis of the NBS Data Encryption Standard," *Computer*, v. 10, n. 3, Mar 1979, pp. 74–84.

[DK85] C. Deavours and L.A. Kruh, *Machine Cryptography and Modern Cryptanalysis*, Artech House, Dedham MA, 1985.

[DKR97] J. Daemen, L. Knudsen, and V. Rijmen, "The Block Cipher Square," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 149–165.

[ElG85] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory,* v. IT-31, n. 4, 1985, pp. 469–472.

[Fei73] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, v. 228, n. 5, May 1973, pp. 15–23.

[Fer96] N. Ferguson, personal communication, 1996.

[FNS75] H. Feistel, W.A. Notz, and J.L. Smith, "Some Cryptographic Techniques for Machine-to-Machine Data Communications, *Proceedings on the IEEE*, v. 63, n. 11, 1975, pp. 1545–1554.

[FS97] N. Ferguson and B. Schneier, "Cryptanalysis of Akelarre," *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, School of Computer Science, Carleton University, 1997, pp. 201–212.

[GC94] H. Gilbert and P. Chauvaud, "A Chosen-Plaintext Attack on the 16-Round Khufu Cryptosystem," *Advances in Cryptology — CRYPTO '94 Proceedings*, Springer-Verlag, 1994, pp. 359–368.

[GOST89] GOST, Gosudarstvennyi Standard 28147-89, "Cryptographic Protection for Data Processing Systems," Government Committee of the USSR for Standards, 1989.

[Gut98] P. Gutmann, "Software Generation of Random Numbers for Cryptographic Purposes," *Proceedings of the 1998 Usenix Security Symposium*, 1998, pp. 243–257.

[Har96] C. Harpes, *Cryptanalysis of Iterated Block Ciphers*, ETH Series on Information Processing, v. 7, Hartung-Gorre Verlang Konstanz, 1996.

[Haw98] P. Hawkes, "Differential-Linear Weak Key Classes of IDEA," *Advances in Cryptology — EUROCRYPT '98 Proceedings*, Springer-Verlag, 1998, pp. 112–126.

[HKM95] C. Harpes, G. Kramer, and J. Massey, "A Generalization of Linear Cryptanalysis and the Applicability of Matsui's Piling-up Lemma," *Advances in Cryptology — EUROCRYPT '95 Proceedings*, Springer-Verlag, 1995, pp. 24–38.

[HKR+98] C. Hall, J. Kelsey, V. Rijmen, B. Schneier, and D. Wagner, "Cryptanalysis of SPEED," unpublished manuscript, 1998.

[HKSW98] C. Hall, J. Kelsey, B. Schneier, and D. Wagner, "Cryptanalysis of SPEED," *Financial Cryptography '98 Proceedings*, Springer-Verlag, 1998, to appear.

[HM97] C. Harpes and J. Massey, "Partitioning Cryptanalysis," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 13–27.

[HT94] H.M. Heys and S.E. Tavares, "On the Security of the CAST Encryption Algorithm," Canadian Conference on Electrical and Computer Engineering, 1994, pp. 332–335.

[Jeff+76] T. Jefferson et al., "Declaration of Independence," Philadelphia PA, 4 Jul 1776.

[JH96] T. Jakobsen and C. Harpes, "Bounds on Non-Uniformity Measures for Generalized Linear Cryptanalysis and Partitioning Cryptanalysis," *Pragocrypt '96 Proceedings*, 1996, pp. 467–479.

[JK97] T. Jakobsen and L. Knudsen, "The Interpolation Attack on Block Ciphers," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 28–40.

[Kie96] K. Kiefer, "A New Design Concept for Building Secure Block Ciphers," *Proceedings of the 1st International Conference on the Theory and Applications of Cryptography, Pragocrypt '96*, CTU Publishing House, 1996, pp. 30–41.

[KKT94] T. Kaneko, K. Koyama, and R. Terada, "Dynamic Swapping Schemes and Differential Cryptanalysis," *IEICE Tranactions*, v. E77-A, 1994, pp. 1328–1336.

[KLPL95] K. Kim, S. Lee, S. Park, and D. Lee, "Securing DES S-boxes Against Three Robust Cryptanalysis," *Proceedings of the Workshop on Selected Areas in Cryptography (SAC '95)*, Ottawa, Canada, 1995, pp. 145–157.

[KM97] L.R. Knudsen and W. Meier, "Differential Cryptanalysis of RC5," *European Transactions on Communication*, v. 8, n. 5, 1997, pp. 445–454.

[Knu93a] L.R. Knudsen, "Cryptanalysis of LOKI," *Advances in Cryptology — ASIACRYPT '91*, Springer-Verlag, 1993, pp. 22–35.

[Knu93b] L.R. Knudsen, "Cryptanalysis of LOKI91," *Advances in Cryptology — AUSCRYPT '92*, Springer-Verlag, 1993, pp. 196–208.

[Knu93c] L.R. Knudsen, "Iterative Characteristics of DES and s^2DES," *Advances in Cryptology — CRYPTO '92*, Springer-Verlag, 1993, pp. 497–511.

[Knu94a] L.R. Knudsen, "Block Ciphers — Analysis, Design, Applications," Ph.D. dissertation, Aarhus University, Nov 1994.

[Knu94b] L.R. Knudsen, "Practically Secure Feistel Ciphers," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 211–221.

[Knu95a] L.R. Knudsen, "New Potentially 'Weak' Keys for DES and LOKI," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 419–424.

[Knu95b] L.R. Knudsen, "Truncated and Higher Order Differentials," *Fast Software Encryption, 2nd International Workshop Proceedings*, Springer-Verlag, 1995, pp. 196–211.

[Knu95c] L.R. Knudsen, "A Key-Schedule Weakness in SAFER K-64," *Advances in Cryptology— CRYPTO '95 Proceedings*, Springer-Verlag, 1995, pp. 274–286.

[Koc96] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 104–113.

[Koc98] P. Kocher, personal communication, 1998.

[KPL93] K. Kim, S. Park, and S. Lee, "Reconstruction of s^2DES S-Boxes and their Immunity to Differential Cryptanalysis," *Proceedings of the 1993 Japan-Korea Workshop on Information Security and Cryptography*, Seoul, Korea, 24–26 October 1993, pp. 282–291.

[KR94] B. Kaliski Jr., and M. Robshaw, "Linear Cryptanalysis Using Multiple Approximations," *Advances in Cryptology — CRYPTO '94 Proceedings*, Springer-Verlag, 1994, pp. 26–39.

[KR95] B. Kaliski Jr., and M. Robshaw, "Linear Cryptanalysis Using Multiple Approximations and FEAL," *Fast Software Encryption, Second International Workshop Proceedings*, Springer-Verlag, 1995, pp. 249–264.

[KR96a] L. Knudsen and M. Robshaw, "Non-Linear Approximations in Linear Cryptanalysis," *Advances in Cryptology — EUCROCRYPT '96*, Springer-Verlag, 1996, pp. 224–236.

[KR96] J. Kilian and P. Rogaway, "How to Protect DES Against Exhaustive Key Search," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 252–267.

[KR97] L.R. Knudsen and V. Rijmen, "Two Rights Sometimes Make a Wrong," *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, School of Computer Science, Carleton University, 1997, pp. 213–223.

[KRRR98] L.R. Knudsen, V. Rijmen, R. Rivest, and M. Robshaw, "On the Design and Security of RC2," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 206–221.

[KSHW98] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure Applications of Low-Entropy Keys," *Information Security. First International Workshop ISW '97 Proceedings*, Springer-Verlag, 1998, 121–134.

[KSW96] J. Kelsey, B. Schneier, and D. Wagner, "Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES," *Advances in Cryptology — CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 237–251.

[KSW97] J. Kelsey, B. Schneier, and D. Wagner, "Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA," *Information and Communications Security, First International Conference Proceedings*, Springer-Verlag, 1997, pp. 203–207.

[KSWH98a] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 168–188.

[KSWH98b] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers," *ESORICS '98 Proceedings*, Springer-Verlag, 1998, to appear.

[KSWH98c] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Yarrow: A Pseudorandom Number Generator," in preparation.

[Kwa97] M. Kwan, "The Design of ICE Encryption Algorithm," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 69–82.

[KY95] B.S. Kaliski and Y.L. Yin, "On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm," *Advances in Cryptology—CRYPTO '95 Proceedings*, Springer-Verlag, 1995, pp. 445–454.

[Lai94] X. Lai, "Higher Order Derivations and Differential Cryptanalysis," *Communications and Cryptography: Two Sides of One Tapestry*, Kluwer Academic Publishers, 1994, pp. 227–233.

[LC97] C.-H. Lee and Y.-T. Cha, "The Block Cipher: SNAKE with Provable Resistance Against DC and LC Attacks," *Proceedings of JW-ISC '97*, KIISC and ISEC Group of IEICE, 1997, pp. 3–17.

[Lee96] M. Leech, "CRISP: A Feistel Network with Hardened Key Scheduling," *Workshop on Selected Areas in Cryptography (SAC '96) Workshop Record*, Queens University, 1996, pp. 15–29.

[LH94] S. Langford and M. Hellman, "Differential-Linear Cryptanalysis," *Advances in Cryptology — CRYPTO '94 Proceedings*, Springer-Verlag, 1994, pp. 17–26.

[LM91] X. Lai and J. Massey, "A Proposal for a New Block Encryption Standard," *Advances in Cryptology — EUROCRYPT '90 Proceedings*, Springer-Verlag, 1991, pp. 389–404.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91 Proceedings*, Springer-Verlag, 1991, pp. 17–38.

[MA96] S. Mister and C. Adams, "Practical S-Box Design," *Workshop on Selected Areas in Cryptography (SAC '96) Workshop Record*, Queens University, 1996, pp. 61–76.

[Mad84] W.E. Madryga, "A High Performance Encryption Algorithm," *Computer Security: A Global Challenge*, Elsevier Science Publishers, 1984, pp. 557–570.

[Mas94] J.L. Massey, "SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 1–17.

[Mat94] M. Matsui, "Linear Cryptanalysis Method for DES Cipher," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 386–397.

[Mat95] M. Matsui, "On Correlation Between the Order of S-Boxes and the Strength of DES," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 366–375.

[Mat96] M. Matsui, "New Structure of Block Ciphers with Provable Security Against Differential and Linear Cryptanalysis," *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 205–218.

[Mat97] M. Matsui, "New Block Encryption Algorithm MISTY," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 54–68.

[McD97] T.J. McDermott, "NSA comments on criteria for AES," letter to NIST, National Security Agency, 2 Apr 97.

[Mer91] R.C. Merkle, "Fast Software Encryption Functions," *Advances in Cryptology — CRYPTO '90 Proceedings*, Springer-Verlag, 1991, pp. 476–501.

[MS77] F.J. MacWilliams and N.J.A. Sloane, "The Theory of Error-Correcting Codes," North-Holland, Amsterdam, 1977.

[MSK98a] S. Moriai, T. Shimoyama, and T. Kaneko, "Higher Order Differential Attack of a CAST Cipher," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 17–31.

[MSK98b] S. Moriai, T. Shimoyama, and T. Kaneko, "Interpolation Attacks of the Block Cipher: SNAKE," unpublished manuscript, 1998.

[Mur90] S. Murphy, "The Cryptanalysis of FEAL-4 with 20 Chosen Plaintexts," *Journal of Cryptology*, v. 2, n. 3, 1990, pp. 145–154.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[NBS80] National Bureau of Standards, NBS FIPS PUB 46, "DES Modes of Operation," National Bureau of Standards, U.S. Department of Commerce, Dec 1980.

[NIST93] National Institute of Standards and Technology, "Secure Hash Standard," U.S. Department of Commerce, May 1993.

[NIST94] National Institute of Standards and Technologies, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[NIST97a] National Institute of Standards and Technology, "Announcing Development of a Federal Information Standard for Advanced Encryption Standard," *Federal Register*, v. 62, n. 1, 2 Jan 1997, pp. 93–94.

[NIST97b] National Institute of Standards and Technology, "Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)," *Federal Register*, v. 62, n. 117, 12 Sep 1997, pp. 48051–48058.

[NK95] K. Nyberg and L.R. Knudsen, "Provable Security Against Differential Cryptanalysis," *Journal of Cryptology*, v. 8, n. 1, 1995, pp. 27–37.

[NM97] J. Nakajima and M. Matsui, "Fast Software Implementation of MISTY on Alpha Processors," *Proceedings of JW-ISC '97*, KIISC and ISEC Group of IEICE, 1997, pp. 55–63.

[Nyb91] K. Nyberg, "Perfect Nonlinear S-boxes," *Advances in Cryptology — EUROCRYPT '91 Proceedings*, Springer-Verlag, 1991, pp. 378–386.

[Nyb93] K. Nyberg, "On the Construction of Highly Nonlinear Permutations," *Advances in Cryptology — EUROCRYPT '92 Proceedings*, Springer-Verlag, 1993, pp. 92–98.

[Nyb94] K. Nyberg, "Differentially Uniform Mappings for Cryptography," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 55–64.

[Nyb95] K. Nyberg, "Linear Approximation of Block Ciphers," *Advances in Cryptology — EUROCRYPT '94 Proceedings*, Springer-Verlag, 1995, pp. 439–444.

[Nyb96] K. Nyberg, "Generalized Feistel Networks," *Advances in Cryptology — ASIACRYPT '96 Proceedings*, Springer-Verlag, 1996, pp. 91–104.

[OCo94a] L. O'Connor, "Enumerating Nondegenerate Permutations," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 368–377.

[OCo94b] L. O'Connor, "On the Distribution of Characteristics in Bijective Mappings," *Advances in Cryptology — EUROCRYPT '93 Proceedings*, Springer-Verlag, 1994, pp. 360–370.

[OCo94c] L. O'Connor, "On the Distribution of Characteristics in Composite Permutations," *Advances in Cryptology — CRYPTO '93 Proceedings*, Springer-Verlag, 1994, pp. 403–412.

[Plu94] C. Plumb, "Truly Random Numbers," *Dr. Dobbs Journal*, v. 19, n. 13, Nov 1994, pp. 113-115.

[Pre93] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*, Ph.D. dissertation, Katholieke Universiteit Leuven, Jan 1993.

[PRB98] B. Preneel, V. Rijmen, A. Bosselaers, "Recent Developments in the Design of Conventional Cryptographic Algorithms," *State of the Art and Evolution of Computer Security and Industrial Cryptography, Lecture Notes in Computer Science*, B. Preneel, R. Govaerts, J. Vandewalle, Eds., Springer-Verlag, 1998, to appear.

[QDD86] J.-J. Quisquater, Y. Desmedt, and M. Davio, "The Importance of 'Good' Key Scheduling Schemes," *Advances in Cryptology — CRYPTO '85 Proceedings*, Springer-Verlag, 1986, pp. 537–542.

[RAND55] RAND Corporation, *A Million Random Digits with 100,000 Normal Deviates*, Glencoe, IL, Free Press Publishers, 1955.

[RC94] P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 56–63.

[RC97] P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm," full version of [RC94], available at `http://www.cs.ucdavis.edu/~rogaway/papers/seal.ps`, 1997.

[RDP+96] V. Rijmen, B. Preneel, A. Bosselaers, and E. DeWin, "The Cipher SHARK," *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 99–111.

[Rij97] V. Rijmen, *Cryptanalysis and Design of Iterated Block Ciphers,* Ph.D. thesis, Katholieke Universiteit Leuven, Oct 1997.

[RIPE92] Research and Development in Advanced Communication Technologies in Europe, *RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation (R1040)*, RACE, June 1992.

[Riv91] R.L. Rivest, "The MD4 Message Digest Algorithm," *Advances in Cryptology — CRYPTO '90 Proceedings*, Springer-Verlag, 1991, pp. 303–311.

[Riv92] R.L. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, Apr 1992.

[Riv95] R.L. Rivest, "The RC5 Encryption Algorithm," *Fast Software Encryption, 2nd International Workshop Proceedings*, Springer-Verlag, 1995, pp. 86–96.

[Riv97] R. Rivest, "A Description of the RC2(r) Encryption Algorithm," Internet-Draft, work in progress, June 1997.

[Ros98] G. Rose, "A Stream Cipher Based on Linear Feedback over GF(2^8)," *Third Australian Conference, ACISP '98*, Springer-Verlag, to appear.

[RP95a] V. Rijmen and B. Preneel, "Cryptanalysis of MacGuffin," *Fast Software Encryption, Second International Workshop Proceedings*, Springer-Verlag, 1995, pp. 353–358.

[RP95b] V. Rijmen and B. Preneel, "On Weaknesses of Non-surjective Round Functions," *Proceedings of the Workshop on Selected Areas in Cryptography (SAC '95)*, Ottawa, Canada, 1995, pp. 100–106.

[RPD97] V. Rijman, B. Preneel and E. DeWin, "On Weaknesses of Non-surjective Round Functions," *Designs, Codes, and Cryptography*, v. 12, n. 3, 1997, pp. 253–266.

[RSA78] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, Feb 1978, pp. 120–126.

[SAM97] T. Shimoyama, S. Amada, and S. Moriai, "Improved Fast Software Implementation of Block Ciphers," *Information and Communications Security, First International Conference, ICICS '97 Proceedings*, Springer-Verlag, 1997, pp. 203–207.

[Sch94] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 191–204.

[Sch96] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

[Sco85] R. Scott, "Wide Open Encryption Design Offers Flexible Implementation," *Cryptologia*, v. 9, n. 1, Jan 1985, pp. 75–90.

[Sel98] A.A. Selçuk, "New Results in Linear Cryptanalysis of RC5," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 1–16.

[SK96] B. Schneier and J. Kelsey, "Unbalanced Feistel Networks and Block Cipher Design," *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 121–144.

[SK98] T. Shimoyama and T. Kaneko, "Quadratic Relation of S-box and Its Application to the Linear Attack of Full Round DES," *Advances in Cryptology — CRYPTO '98 Proceedings*, Springer-Verlag, 1998, in preparation.

[SM88] A. Shimizu and S. Miyaguchi, "Fast Data Encipherment Algorithm FEAL," *Advances in Cryptology — EUROCRYPT '87 Proceedings*, Springer-Verlag, 1988, pp. 267–278.

[SMK98] T. Shimoyama, S. Moriai, and T. Kaneko, "Improving the Higher Order Differential Attack and Cryptanalysis of the *KN* Cipher," *Information Security. First International Workshop ISW '97 Proceedings*, Springer-Verlag, 1998, pp. 32–42.

[SV98] J. Stern and S. Vaudenay, "CS-Cipher," *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp. 189–205.

[SW97] B. Schneier and D. Whiting, "Fast Software Encryption: Designing Encryption Algorithms for Optimal Speed on the Intel Pentium Processor," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 242–259.

[Vau95] S. Vaudenay, "On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER," *Fast Software Encryption, Second International Workshop Proceedings*, Springer-Verlag, 1995, pp. 286–297.

[Vau96a] S. Vaudenay, "On the Weak Keys in Blowfish," *Fast Software Encryption, 3rd International Workshop Proceedings*, Springer-Verlag, 1996, pp. 27–32.

[Vau96b] S. Vaudenay, "An Experiment on DES Statistical Cryptanalysis," *3rd ACM Conference on Computer and Communications Security*, ACM Press, 1996, pp. 139–147.

[Wag95a] D. Wagner, "Cryptanalysis of S-1," sci.crypt Usenet posting, 27 Aug 1995.

[Wag95b] D. Wagner, personal communication, 1995.

[WH87] R. Winternitz and M. Hellman, "Chosen-key Attacks on a Block Cipher," *Cryptologia*, v. 11, n. 1, Jan 1987, pp. 16–20.

[Whe94] D. Wheeler, "A Bulk Data Encryption Algorithm," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 127–134.

[Wie94] M.J. Wiener, "Efficient DES Key Search," TR-244, School of Computer Science, Carleton University, May 1994.

[Win84] R.S. Winternitz, "Producing One-Way Hash Functions from DES," *Advances in Cryptology: Proceedings of Crypto 83*, Plenum Press, 1984, pp. 203–207.

[WN95] D. Wheeler and R. Needham, "TEA, a Tiny Encryption Algorithm," *Fast Software Encryption, 2nd International Workshop Proceedings*, Springer-Verlag, 1995, pp. 97–110.

[WSK97] D. Wagner, B. Schneier, and J. Kelsey, "Cryptanalysis of the Cellular Message Encryption Algorithm," *Advances in Cryptology — CRYPTO '97 Proceedings*, Springer-Verlag, 1997, pp. 526–537.

[YLCY98] X. Yi, K.Y. Lam, S.X. Cheng, and X.H. You, "A New Byte-Oriented Block Cipher," *Information Security. First International Workshop ISW '97 Proceedings*, Springer-Verlag, 1998, 209–220.

[YMT97] A.M. Youssef, S. Mister, and S.E. Tavares, "On the Design of Linear Transformations for Substitution Permutation Encryption Networks," *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, School of Computer Science, Carleton University, 1997, pp. 40–48.

[YTH96] A.M. Youssef, S.E. Tavares, and H.M. Heys, "A New Class of Substitution-Permutation Networks," *Workshop on Selected Areas in Cryptography (SAC '96) Workshop Record*, Queens University, 1996, pp. 132–147.

[Yuv97] G. Yuval, "Reinventing the Travois: Encrytion/MAC in 30 ROM Bytes," *Fast Software Encryption, 4th International Workshop Proceedings*, Springer-Verlag, 1997, pp. 205–209.

[ZG97] F. Zhu and B.-A. Guo, "A Block-Ciphering Algorithm Based on Addition-Multiplication Structure in GF(2^n)," *Workshop on Selected Areas in Cryptography (SAC '97) Workshop Record*, School of Computer Science, Carleton University, 1997, pp. 145–159.

[Zhe97] Y. Zheng, "The SPEED Cipher," *Financial Cryptography '97 Proceedings*, Springer-Verlag, 1997, pp. 71–89.

[ZMI90] Y. Zheng, T. Matsumoto, and H. Imai, "On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses," *Advances in Cryptology — CRYPTO '89 Proceedings*, Springer-Verlag, 1990, pp. 461–480.

[ZPS93] Y. Zheng, J. Pieprzyk, and J. Seberry, "HAVAL — A One-Way Hashing Algorithm with Variable Length of Output," *Advances in Cryptology — AUSCRYPT '92 Proceedings*, Springer-Verlag, 1993, pp. 83–104.

A Twofish Test Vectors

A.1 Intermediate Values

The following file shows the intermediate values of three Twofish computations. This particular implementation does not swap the halves but instead applies the F function alternately between the two halves.

```
FILENAME:  "ecb_ival.txt"

Electronic Codebook (ECB) Mode
Intermediate Value Tests

Algorithm Name:       TWOFISH
Principal Submitter:  Bruce Schneier, Counterpane Systems

==========

KEYSIZE=128

KEY=00000000000000000000000000000000

;
;makeKey:   Input key             -->  S-box key      [Encrypt]
;             00000000 00000000  -->  00000000
;             00000000 00000000  -->  00000000
;           Subkeys
;             52C54DDE 11F0626D   Input whiten
;             7CAC9D4A 4D1B4AAA
;             B7B83A10 1E7D0BEB   Output whiten
;             EE9C341F CFE14BE4
;             F98FFEF9 9C5B3C17   Round subkeys
;             15A48310 342A4D81
;             424D89FE C14724A7
;             311B834C FDE87320
;             3302778F 26CD67B4
;             7A6C6362 C2BAF60E
;             3411B994 D972C87F
;             84ADB1EA A7DEE434
;             54D2960F A2F7CAA8
;             A6B8FF8C 8014C425
;             6A748D1C EDBAF720
;             928EF78C 0338EE13
;             9949D6BE C8314176
;             07C07D68 ECAE7EA7
;             1FE71844 85C05C89
;             F298311E 696EA672
;
```

```
PT=00000000000000000000000000000000

Encrypt()

R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 52C54DDE  11F0626D  7CAC9D4A  4D1B4AAA.
R[ 1]: x= 52C54DDE  11F0626D  C38DCAA4  7A0A91B6.    t0=C06D4949. t1=41B9BFC1.
R[ 2]: x= 55A538DE  5C5A4DB6  C38DCAA4  7A0A91B6.    t0=7C4536B9. t1=67A58299.
R[ 3]: x= 55A538DE  5C5A4DB6  899063BD  893E49A9.    t0=60DAC1A4. t1=2D84C23D.
R[ 4]: x= 2AE61A96  84BC42D3  899063BD  893E49A9.    t0=607AAEAD. t1=6ED2DBF9.
R[ 5]: x= 2AE61A96  84BC42D3  F14F2618  821B5F36.    t0=067D0B49. t1=318EACB4.
R[ 6]: x= 0FFE0AD1  D6B87B70  F14F2618  821B5F36.    t0=58554EDB. t1=62585CF7.
R[ 7]: x= 0FFE0AD1  D6B87B70  CD0D38A1  C069BD9B.    t0=839B0017. t1=B3A89DB0.
R[ 8]: x= A85CE579  DE2661CE  CD0D38A1  C069BD9B.    t0=E9BC6975. t1=F0DDA4C3.
R[ 9]: x= A85CE579  DE2661CE  7A39754C  973ABD2A.    t0=54687CDF. t1=9044BF4B.
R[10]: x= 013077D7  B3528BA1  7A39754C  973ABD2A.    t0=77FC927F. t1=8B8678CC.
R[11]: x= 013077D7  B3528BA1  D57933FD  F8EA8B1B.    t0=E3C81108. t1=828E7493.
R[12]: x= 64F0EAA1  DA27090C  D57933FD  F8EA8B1B.    t0=B33C25D6. t1=83068533.
R[13]: x= 64F0EAA1  DA27090C  F64F1005  99149A52.    t0=A0AA2F81. t1=FFF30DB7.
R[14]: x= B0681C46  606D0273  F64F1005  99149A52.    t0=114C17C5. t1=EB143CFF.
R[15]: x= B0681C46  606D0273  EB27628F  2C51191D.    t0=677DA87D. t1=989D1459.
R[16]: x= C1708BA9  9522A3CE  EB27628F  2C51191D.    t0=9357B338. t1=AC9926BF.
R[17]: x= 5C9F589F  322C12F6  2FECBFB6  5AC3E82A.

CT=9F589F5CF6122C32B6BFEC2F2AE8C35A

Decrypt()

CT=9F589F5CF6122C32B6BFEC2F2AE8C35A


R[17]: x= 5C9F589F  322C12F6  2FECBFB6  5AC3E82A.
R[16]: x= C1708BA9  9522A3CE  EB27628F  2C51191D.    t0=9357B338. t1=AC9926BF.
R[15]: x= B0681C46  606D0273  EB27628F  2C51191D.    t0=677DA87D. t1=989D1459.
R[14]: x= B0681C46  606D0273  F64F1005  99149A52.    t0=114C17C5. t1=EB143CFF.
R[13]: x= 64F0EAA1  DA27090C  F64F1005  99149A52.    t0=A0AA2F81. t1=FFF30DB7.
R[12]: x= 64F0EAA1  DA27090C  D57933FD  F8EA8B1B.    t0=B33C25D6. t1=83068533.
R[11]: x= 013077D7  B3528BA1  D57933FD  F8EA8B1B.    t0=E3C81108. t1=828E7493.
R[10]: x= 013077D7  B3528BA1  7A39754C  973ABD2A.    t0=77FC927F. t1=8B8678CC.
R[ 9]: x= A85CE579  DE2661CE  7A39754C  973ABD2A.    t0=54687CDF. t1=9044BF4B.
R[ 8]: x= A85CE579  DE2661CE  CD0D38A1  C069BD9B.    t0=E9BC6975. t1=F0DDA4C3.
R[ 7]: x= 0FFE0AD1  D6B87B70  CD0D38A1  C069BD9B.    t0=839B0017. t1=B3A89DB0.
R[ 6]: x= 0FFE0AD1  D6B87B70  F14F2618  821B5F36.    t0=58554EDB. t1=62585CF7.
R[ 5]: x= 2AE61A96  84BC42D3  F14F2618  821B5F36.    t0=067D0B49. t1=318EACB4.
R[ 4]: x= 2AE61A96  84BC42D3  899063BD  893E49A9.    t0=607AAEAD. t1=6ED2DBF9.
R[ 3]: x= 55A538DE  5C5A4DB6  899063BD  893E49A9.    t0=60DAC1A4. t1=2D84C23D.
R[ 2]: x= 55A538DE  5C5A4DB6  C38DCAA4  7A0A91B6.    t0=7C4536B9. t1=67A58299.
R[ 1]: x= 52C54DDE  11F0626D  C38DCAA4  7A0A91B6.    t0=C06D4949. t1=41B9BFC1.
R[ 0]: x= 52C54DDE  11F0626D  7CAC9D4A  4D1B4AAA.
R[-1]: x= 00000000  00000000  00000000  00000000.

PT=00000000000000000000000000000000

==========

KEYSIZE=192

KEY=0123456789ABCDEFFEDCBA9876543210001122334455667 7

;
;makeKey:   Input key            -->  S-box key     [Encrypt]
;             EFCDAB89 67452301  -->  B89FF6F2
;             10325476 98BADCFE  -->  B255BC4B
;             77665544 33221100  -->  45661061
;           Subkeys
;             38394A24 C36D1175   Input whiten
;             E802528F 219BFEB4
;             B9141AB4 BD3E70CD   Output whiten
;             AF609383 FD36908A
;             03EFB931 1D2EE7EC   Round subkeys
;             A7489D55 6E44B6E8
;             714AD667 653AD51F
;             B6315B66 B27C05AF
;             A06C8140 9853D419
;             4016E346 8D1C0DD4
;             F05480BE B6AF816F
;             2D7DC789 45B7BD3A
;             57F8A163 2BEFDA69
;             26AE7271 C2900D79
;             ED323794 3D3FFD80
;             5DE68E49 9C3D2478
;             DF326FE3 5911F70D
;             C229F13B B1364772
;             4235364D 0CEC363A
;             57C8DD1F 6A1AD61E
;
PT=00000000000000000000000000000000

Encrypt()

R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 38394A24  C36D1175  E802528F  219BFEB4.
R[ 1]: x= 38394A24  C36D1175  9C263D67  5E68BE8F.    t0=988C8223. t1=33D1ECEC.
R[ 2]: x= C8F5099F  0C4B8F53  9C263D67  5E68BE8F.    t0=E8C880BC. t1=19C23B0A.
R[ 3]: x= C8F5099F  0C4B8F53  69948F5E  E67C030F.    t0=C615F1F6. t1=17AE5B7E.
R[ 4]: x= 07633866  59421079  69948F5E  E67C030F.    t0=90AB32AA. t1=7F56EB43.
R[ 5]: x= 07633866  59421079  C015BE79  149B9CEC.    t0=52971E00. t1=F6BC546D.
R[ 6]: x= A042B99D  709EF54B  C015BE79  149B9CEC.    t0=DAA00849. t1=2D2F5FCE.
R[ 7]: x= A042B99D  709EF54B  0CD39FA6  B250BEDA.    t0=EE03FB5B. t1=FB5A051C.
R[ 8]: x= F7B097FA  9E5C4FF7  0CD39FA6  B250BEDA.    t0=09A1B597. t1=18041948.
R[ 9]: x= F7B097FA  9E5C4FF7  77FC8B29  CC2B3F88.    t0=99C9694E. t1=F1687F43.
R[10]: x= A279C718  421A8D38  77FC8B29  CC2B3F88.    t0=5D174956. t1=2F7D5E04.
R[11]: x= A279C718  421A8D38  5B1A0904  12FEBF99.    t0=5BC40012. t1=78D2617B.
R[12]: x= E4409C22  702548A2  5B1A0904  12FEBF99.    t0=C251B3CE. t1=4AC0BD46.
R[13]: x= E4409C22  702548A2  5DDAA2A1  EFB2F051.    t0=91BC2070. t1=6FC0BBF3.
R[14]: x= 8561A604  825D2480  5DDAA2A1  EFB2F051.    t0=A7D24F8E. t1=84878F62.
R[15]: x= 8561A604  825D2480  5CC6CB7B  62A2CE64.    t0=93690387. t1=0EB8FA83.
R[16]: x= 17738CD3  B5142D18  5CC6CB7B  62A2CE64.    t0=5FE8370B. t1=F3D5AB78.
R[17]: x= E5D2D1CF  DF9CBEA9  B8131F50  4822BD92.

CT=CFD1D2E5A9BE9CDF501F13B892BD2248

Decrypt()

CT=CFD1D2E5A9BE9CDF501F13B892BD2248


R[17]: x= E5D2D1CF  DF9CBEA9  B8131F50  4822BD92.
R[16]: x= 17738CD3  B5142D18  5CC6CB7B  62A2CE64.    t0=5FE8370B. t1=F3D5AB78.
R[15]: x= 8561A604  825D2480  5CC6CB7B  62A2CE64.    t0=93690387. t1=0EB8FA83.
R[14]: x= 8561A604  825D2480  5DDAA2A1  EFB2F051.    t0=A7D24F8E. t1=84878F62.
R[13]: x= E4409C22  702548A2  5DDAA2A1  EFB2F051.    t0=91BC2070. t1=6FC0BBF3.
R[12]: x= E4409C22  702548A2  5B1A0904  12FEBF99.    t0=C251B3CE. t1=4AC0BD46.
R[11]: x= A279C718  421A8D38  5B1A0904  12FEBF99.    t0=5BC40012. t1=78D2617B.
R[10]: x= A279C718  421A8D38  77FC8B29  CC2B3F88.    t0=5D174956. t1=2F7D5E04.
R[ 9]: x= F7B097FA  9E5C4FF7  77FC8B29  CC2B3F88.    t0=99C9694E. t1=F1687F43.
R[ 8]: x= F7B097FA  9E5C4FF7  0CD39FA6  B250BEDA.    t0=09A1B597. t1=18041948.
R[ 7]: x= A042B99D  709EF54B  0CD39FA6  B250BEDA.    t0=EE03FB5B. t1=FB5A051C.
R[ 6]: x= A042B99D  709EF54B  C015BE79  149B9CEC.    t0=DAA00849. t1=2D2F5FCE.
R[ 5]: x= 07633866  59421079  C015BE79  149B9CEC.    t0=52971E00. t1=F6BC546D.
R[ 4]: x= 07633866  59421079  69948F5E  E67C030F.    t0=90AB32AA. t1=7F56EB43.
R[ 3]: x= C8F5099F  0C4B8F53  69948F5E  E67C030F.    t0=C615F1F6. t1=17AE5B7E.
R[ 2]: x= C8F5099F  0C4B8F53  9C263D67  5E68BE8F.    t0=E8C880BC. t1=19C23B0A.
R[ 1]: x= 38394A24  C36D1175  9C263D67  5E68BE8F.    t0=988C8223. t1=33D1ECEC.
R[ 0]: x= 38394A24  C36D1175  E802528F  219BFEB4.
R[-1]: x= 00000000  00000000  00000000  00000000.

PT=00000000000000000000000000000000

==========

KEYSIZE=256

KEY=0123456789ABCDEFFEDCBA98765432100011223344556678899AABBCCDDEEFF

;
;makeKey:   Input key            -->  S-box key     [Encrypt]
;             EFCDAB89 67452301  -->  B89FF6F2
;             10325476 98BADCFE  -->  B255BC4B
;             77665544 33221100  -->  45661061
;             FFEEDDCC BBAA9988  -->  8E4447F7
;           Subkeys
;             5EC769BF 44D13C60   Input whiten
;             76CD39B1 16750474
;             349C294B EC21F6D6   Output whiten
;             4FBD10B4 578DA0ED
;             C3479695 9B6958FB   Round subkeys
;             6A7FBC4E 0BF1830B
;             61B5E0FB D78D9730
;             7C6CF0C4 2F9109C8
;             E69EA8D1 ED99BDFF
;             35DC0BBD A03E5018
;             FB18EA0B 38BD43D3
;             76191781 37A9A0D3
;             72427BEA 911CC0B8
;             F1689449 71009CA9
;             B6363E89 494D9855
;             590BBC63 F95A28B5
;             FB72B4E1 2A43505C
;             BFD34176 5C133D12
;             3A9247F7 9A3331DD
;             EE7515E6 F0D54DCD
;
PT=00000000000000000000000000000000

Encrypt()

R[-1]: x= 00000000  00000000  00000000  00000000.
R[ 0]: x= 5EC769BF  44D13C60  76CD39B1  16750474.
R[ 1]: x= 5EC769BF  44D13C60  D38B6C9F  A23B7169.    t0=29C0736C. t1=E4D3D68D.
R[ 2]: x= 99424DFF  FBC14BFC  D38B6C9F  A23B7169.    t0=9D16BBB3. t1=64AD7A3F.
R[ 3]: x= 99424DFF  FBC14BFC  698BE047  6A997290.    t0=E66B9D19. t1=B87B2DFD.
R[ 4]: x= 2C125DD7  5A526278  698BE047  6A997290.    t0=0BB41F61. t1=3945E62C.
R[ 5]: x= 2C125DD7  5A526278  E35CD910  7CB57D06.    t0=D5397903. t1=F35A3092.
R[ 6]: x= D5178F25  00D35CC5  E35CD910  7CB57D06.    t0=8C8927A1. t1=C3D8103E.
R[ 7]: x= D5178F25  00D35CC5  D8447F91  65C2BD96.    t0=4D8B7489. t1=0B2FC79F.
R[ 8]: x= FF92E109  DF621C97  D8447F91  65C2BD96.    t0=C1176720. t1=F301CE95.
R[ 9]: x= FF92E109  DF621C97  28BFEFF5  D45666FB.    t0=9F3BEC03. t1=77BD388E.
R[10]: x= BB79AD2E  AA410F41  28BFEFF5  D45666FB.    t0=8C6DB451. t1=0B8B72BA.
R[11]: x= BB79AD2E  AA410F41  6576A3ED  BFF8215E.    t0=8A317EF8. t1=A1EAEAAE.
R[12]: x= 4A6BBAFF  439F4766  6576A3ED  BFF8215E.    t0=8F8307AA. t1=472014C3.
R[13]: x= 4A6BBAFF  439F4766  F7186836  04CA5304.    t0=CEB0BBE1. t1=C12302BE.
R[14]: x= CBD3C29D  BC31FEBE  F7186836  04CA5304.    t0=5CF5C93C. t1=C1033512.
R[15]: x= CBD3C29D  BC31FEBE  D4E77B7C  5415D5D3.    t0=853A6BB2. t1=9F09EB26.
R[16]: x= 85411C2B  7777DC05  D4E77B7C  5415D5D3.    t0=877AF61D. t1=4B61EEC7.
R[17]: x= E07B5237  B8342305  CAFC0C9F  20FA7CE8.

CT=37527BE0052334B89F0CFCCAE87CFA20

Decrypt()

CT=37527BE0052334B89F0CFCCAE87CFA20


R[17]: x= E07B5237  B8342305  CAFC0C9F  20FA7CE8.
R[16]: x= 85411C2B  7777DC05  D4E77B7C  5415D5D3.    t0=877AF61D. t1=4B61EEC7.
R[15]: x= CBD3C29D  BC31FEBE  D4E77B7C  5415D5D3.    t0=853A6BB2. t1=9F09EB26.
R[14]: x= CBD3C29D  BC31FEBE  F7186836  04CA5304.    t0=5CF5C93C. t1=C1033512.
R[13]: x= 4A6BBAFF  439F4766  F7186836  04CA5304.    t0=CEB0BBE1. t1=C12302BE.
R[12]: x= 4A6BBAFF  439F4766  6576A3ED  BFF8215E.    t0=8F8307AA. t1=472014C3.
R[11]: x= BB79AD2E  AA410F41  6576A3ED  BFF8215E.    t0=8A317EF8. t1=A1EAEAAE.
R[10]: x= BB79AD2E  AA410F41  28BFEFF5  D45666FB.    t0=8C6DB451. t1=0B8B72BA.
R[ 9]: x= FF92E109  DF621C97  28BFEFF5  D45666FB.    t0=9F3BEC03. t1=77BD388E.
R[ 8]: x= FF92E109  DF621C97  D8447F91  65C2BD96.    t0=C1176720. t1=F301CE95.
```

```
R[ 7]: x= D5178F25  00D35CC5  D8447F91  65C2BD96.    t0=4D8B7489. t1=0B2FC79F.
R[ 6]: x= D5178F25  00D35CC5  E35CD910  7CB57D06.    t0=8C8927A1. t1=C3D8103E.
R[ 5]: x= 2C125DD7  5A526278  E35CD910  7CB57D06.    t0=D5397903. t1=F35A3092.
R[ 4]: x= 2C125DD7  5A526278  698BE047  6A997290.    t0=0BB41F61. t1=3945E62C.
R[ 3]: x= 99424DFF  FBC14BFC  698BE047  6A997290.    t0=E66B9D19. t1=B87B2DFD.
R[ 2]: x= 99424DFF  FBC14BFC  D38B6C9F  A23B7169.    t0=9D16BBB3. t1=64AD7A3F.
R[ 1]: x= 5EC769BF  44D13C60  D38B6C9F  A23B7169.    t0=29C0736C. t1=E4D3D68D.
R[ 0]: x= 5EC769BF  44D13C60  76CD39B1  16750474.
R[-1]: x= 00000000  00000000  00000000  00000000.

PT=00000000000000000000000000000000
```

A.2 Full Encryptions

The following file shows a number of (plaintext, ciphertext, key) pairs. These pairs are related, and can easily be tested automatically. The plaintext of each entry is the ciphertext of the previous one. The key of each entry is made up of the ciphertext two and three entries back. We believe that these test vectors provide a thorough test of a Twofish implementation.

```
FILENAME:  "ecb_tbl.txt"

Electronic Codebook (ECB) Mode
Tables Known Answer Test
Tests permutation tables and MDS matrix multiply tables.

Algorithm Name:       TWOFISH
Principal Submitter:  Bruce Schneier, Counterpane Systems

==========

KEYSIZE=128

I=1
KEY=00000000000000000000000000000000
PT=00000000000000000000000000000000
CT=9F589F5CF6122C32B6BFEC2F2AE8C35A

I=2
KEY=00000000000000000000000000000000
PT=9F589F5CF6122C32B6BFEC2F2AE8C35A
CT=D491DB16E7B1C39E86CB086B789F5419

I=3
KEY=9F589F5CF6122C32B6BFEC2F2AE8C35A
PT=D491DB16E7B1C39E86CB086B789F5419
CT=019F9809DE1711858FAAC3A3BA20FBC3

I=4
KEY=D491DB16E7B1C39E86CB086B789F5419
PT=019F9809DE1711858FAAC3A3BA20FBC3
CT=6363977DE839486297E661C6C9D668EB

I=5
KEY=019F9809DE1711858FAAC3A3BA20FBC3
PT=6363977DE839486297E661C6C9D668EB
CT=816D5BD0FAE35342BF2A7412C246F752

I=6
KEY=6363977DE839486297E661C6C9D668EB
PT=816D5BD0FAE35342BF2A7412C246F752
CT=5449ECA008FF5921155F598AF4CED4D0

I=7
KEY=816D5BD0FAE35342BF2A7412C246F752
PT=5449ECA008FF5921155F598AF4CED4D0
CT=6600522E97AEB3094ED5F92AFCBCDD10

I=8
KEY=5449ECA008FF5921155F598AF4CED4D0
PT=6600522E97AEB3094ED5F92AFCBCDD10
CT=34C8A5FB2D3D08A170D120AC6D26DBFA

I=9
KEY=6600522E97AEB3094ED5F92AFCBCDD10
PT=34C8A5FB2D3D08A170D120AC6D26DBFA
CT=28530B358C1B42EF277DE6D4407FC591

I=10
KEY=34C8A5FB2D3D08A170D120AC6D26DBFA
PT=28530B358C1B42EF277DE6D4407FC591
CT=8A8AB983310ED78C8C0ECDE030B8DCA4

   :
   :
   :

I=48
KEY=137A24CA47CD12BE818DF4D2F4355960
PT=BCA724A54533C6987E14AA827952F921
CT=6B459286F3FFD28D49F15B1581B08E42

I=49
KEY=BCA724A54533C6987E14AA827952F921
PT=6B459286F3FFD28D49F15B1581B08E42
CT=5D9D4EEFFA9151575524F115815A12E0

==========

KEYSIZE=192

I=1
KEY=000000000000000000000000000000000000000000000000
PT=00000000000000000000000000000000
CT=EFA71F788965BD4453F860178FC19101

I=2
KEY=000000000000000000000000000000000000000000000000
PT=EFA71F788965BD4453F860178FC19101
CT=88B2B2706B105E36B446BB6D731A1E88

I=3
KEY=EFA71F788965BD4453F860178FC191010000000000000000
PT=88B2B2706B105E36B446BB6D731A1E88
CT=39DA69D6BA4997D585B6DC073CA341B2

I=4
KEY=88B2B2706B105E36B446BB6D731A1E88EFA71F788965BD44
PT=39DA69D6BA4997D585B6DC073CA341B2
CT=182B02D81497EA45F9DAACDC29193A65

I=5
KEY=39DA69D6BA4997D585B6DC073CA341B288B2B2706B105E36
PT=182B02D81497EA45F9DAACDC29193A65
CT=7AFF7A70CA2FF28AC31DD8AE5DAAAB63

I=6
KEY=182B02D81497EA45F9DAACDC29193A6539DA69D6BA4997D5
PT=7AFF7A70CA2FF28AC31DD8AE5DAAAB63
CT=D1079B789F666649B6BD7D1629F1F77E

I=7
KEY=7AFF7A70CA2FF28AC31DD8AE5DAAAB63182B02D81497EA45
PT=D1079B789F666649B6BD7D1629F1F77E
CT=3AF6F7CE5BD35EF18BEC6FA787AB506B

I=8
KEY=D1079B789F666649B6BD7D1629F1F77E7AFF7A70CA2FF28A
PT=3AF6F7CE5BD35EF18BEC6FA787AB506B
CT=AE8109BFDA85C1F2C5038B34ED691BFF

I=9
KEY=3AF6F7CE5BD35EF18BEC6FA787AB506BD1079B789F666649
PT=AE8109BFDA85C1F2C5038B34ED691BFF
CT=893FD67B98C550073571BD631263FC78

I=10
KEY=AE8109BFDA85C1F2C5038B34ED691BFF3AF6F7CE5BD35EF1
PT=893FD67B98C550073571BD631263FC78
CT=16434FC9C8841A63D58700B5578E8F67

   :
   :
   :

I=48
KEY=DEA4F3DA75EC7A8EAC3861A9912402CD5DBE44032769DF54
PT=FB66522C332FCC4C042ABE32FA9E902F
CT=F0AB73301125FA21EF70BE5385FB76B6

I=49
KEY=FB66522C332FCC4C042ABE32FA9E902FDEA4F3DA75EC7A8E
PT=F0AB73301125FA21EF70BE5385FB76B6
CT=E75449212BEEF9F4A390BD860A640941

==========

KEYSIZE=256

I=1
KEY=0000000000000000000000000000000000000000000000000000000000000000
PT=00000000000000000000000000000000
CT=57FF739D4DC92C1BD7FC01700CC8216F

I=2
KEY=0000000000000000000000000000000000000000000000000000000000000000
PT=57FF739D4DC92C1BD7FC01700CC8216F
CT=D43BB7556EA32E46F2A282B7D45B4E0D

I=3
KEY=57FF739D4DC92C1BD7FC01700CC8216F0000000000000000000000000000000000
PT=D43BB7556EA32E46F2A282B7D45B4E0D
CT=90AFE91BB288544F2C32DC239B2635E6

I=4
KEY=D43BB7556EA32E46F2A282B7D45B4E0D57FF739D4DC92C1BD7FC01700CC8216F
PT=90AFE91BB288544F2C32DC239B2635E6
CT=6CB4561C40BF0A9705931CB6D408E7FA

I=5
KEY=90AFE91BB288544F2C32DC239B2635E6D43BB7556EA32E46F2A282B7D45B4E0D
PT=6CB4561C40BF0A9705931CB6D408E7FA
CT=3059D6D61753B958D92F4781C8640E58

I=6
KEY=6CB4561C40BF0A9705931CB6D408E7FA90AFE91BB288544F2C32DC239B2635E6
PT=3059D6D61753B958D92F4781C8640E58
CT=6B459286F3FFD28D49F15B1581B08E42

I=49
KEY=BCA724A54533C6987E14AA827952F921
PT=6B459286F3FFD28D49F15B1581B08E42
CT=5D9D4EEFFA9151575524F115815A12E0

==========

KEYSIZE=192

I=1
KEY=000000000000000000000000000000000000000000000000
PT=00000000000000000000000000000000
CT=EFA71F788965BD4453F860178FC19101

I=2
KEY=000000000000000000000000000000000000000000000000
PT=EFA71F788965BD4453F860178FC19101
CT=88B2B2706B105E36B446BB6D731A1E88

I=3
KEY=EFA71F788965BD4453F860178FC191010000000000000000
PT=88B2B2706B105E36B446BB6D731A1E88
CT=39DA69D6BA4997D585B6DC073CA341B2

I=4
KEY=88B2B2706B105E36B446BB6D731A1E88EFA71F788965BD44
PT=39DA69D6BA4997D585B6DC073CA341B2
CT=182B02D81497EA45F9DAACDC29193A65

I=5
KEY=39DA69D6BA4997D585B6DC073CA341B288B2B2706B105E36
PT=182B02D81497EA45F9DAACDC29193A65
CT=7AFF7A70CA2FF28AC31DD8AE5DAAAB63

I=6
KEY=182B02D81497EA45F9DAACDC29193A6539DA69D6BA4997D5
PT=7AFF7A70CA2FF28AC31DD8AE5DAAAB63
CT=D1079B789F666649B6BD7D1629F1F77E

I=7
KEY=7AFF7A70CA2FF28AC31DD8AE5DAAAB63182B02D81497EA45
PT=D1079B789F666649B6BD7D1629F1F77E
CT=3AF6F7CE5BD35EF18BEC6FA787AB506B

I=8
KEY=D1079B789F666649B6BD7D1629F1F77E7AFF7A70CA2FF28A
PT=3AF6F7CE5BD35EF18BEC6FA787AB506B
CT=AE8109BFDA85C1F2C5038B34ED691BFF

I=9
KEY=3AF6F7CE5BD35EF18BEC6FA787AB506BD1079B789F666649
PT=AE8109BFDA85C1F2C5038B34ED691BFF
CT=893FD67B98C550073571BD631263FC78

I=10
KEY=AE8109BFDA85C1F2C5038B34ED691BFF3AF6F7CE5BD35EF1
PT=893FD67B98C550073571BD631263FC78
CT=16434FC9C8841A63D58700B5578E8F67

   :
   :
   :

I=48
KEY=DEA4F3DA75EC7A8EAC3861A9912402CD5DBE44032769DF54
PT=FB66522C332FCC4C042ABE32FA9E902F
CT=F0AB73301125FA21EF70BE5385FB76B6

I=49
KEY=FB66522C332FCC4C042ABE32FA9E902FDEA4F3DA75EC7A8E
PT=F0AB73301125FA21EF70BE5385FB76B6
CT=E75449212BEEF9F4A390BD860A640941

==========

KEYSIZE=256

I=1
KEY=0000000000000000000000000000000000000000000000000000000000000000
PT=00000000000000000000000000000000
CT=57FF739D4DC92C1BD7FC01700CC8216F

I=2
KEY=0000000000000000000000000000000000000000000000000000000000000000
PT=57FF739D4DC92C1BD7FC01700CC8216F
CT=D43BB7556EA32E46F2A282B7D45B4E0D

I=3
KEY=57FF739D4DC92C1BD7FC01700CC8216F0000000000000000000000000000000000
PT=D43BB7556EA32E46F2A282B7D45B4E0D
CT=90AFE91BB288544F2C32DC239B2635E6

I=4
KEY=D43BB7556EA32E46F2A282B7D45B4E0D57FF739D4DC92C1BD7FC01700CC8216F
PT=90AFE91BB288544F2C32DC239B2635E6
CT=6CB4561C40BF0A9705931CB6D408E7FA

I=5
KEY=90AFE91BB288544F2C32DC239B2635E6D43BB7556EA32E46F2A282B7D45B4E0D
PT=6CB4561C40BF0A9705931CB6D408E7FA
CT=3059D6D61753B958D92F4781C8640E58

I=6
KEY=6CB4561C40BF0A9705931CB6D408E7FA90AFE91BB288544F2C32DC239B2635E6
PT=3059D6D61753B958D92F4781C8640E58
```

```
CT=E694657705O5D7F8OEF68CA38AB3A3D6

I=7
KEY=3059D6D61753B958D92F4781C8640E586CB4561C40BF0A9705931CB6D408E7FA
PT=E694657705O5D7F8OEF68CA38AB3A3D6
CT=5AB67A5F8539A4A5FD9FO373BA463466

I=8
KEY=E69465770505D7F8OEF68CA38AB3A3D63059D6D61753B958D92F4781C8640E58
PT=5AB67A5F8539A4A5FD9FO373BA463466
CT=DC096BCD99FC72F79936D4C748E75AF7

I=9
KEY=5AB67A5F8539A4A5FD9FO373BA463466E69465770505D7F8OEF68CA38AB3A3D6
PT=DC096BCD99FC72F79936D4C748E75AF7
CT=C5A3E7CEEOF1B7260528A68FB4EA05F2

I=10
KEY=DC096BCD99FC72F79936D4C748E75AF75AB67A5F8539A4A5FD9FO373BA463466
PT=C5A3E7CEEOF1B7260528A68FB4EA05F2
CT=43D5CEC327B24AB90AD34A79D0469151

    :
    :
    :

I=48
KEY=2E2158BC3E5FC714C1EEECA0EA696D48D2DED73E59319A8138E0331F0EA149EA
PT=248A7F3528B168ACFDD1386E3F51E30C
CT=431058F4DBC7F734DA4F02F04CC4F459

I=49
KEY=248A7F3528B168ACFDD1386E3F51E30C2E2158BC3E5FC714C1EEECA0EA696D48
PT=431058F4DBC7F734DA4F02F04CC4F459
CT=37FE26FF1CF66175F5DDF4C33B97A205
```