TWOFISH ENCRYPTION ALGORITHM

CS–627: Cryptology
Fall 2004

Horatiu Paul Stancu

Table of contents

Introduction:

Twofish is a block cipher by Counterpane Labs. It was one of the five Advanced Encryption Standard (AES) finalists. Twofish is unpatented, and the source code is uncopyrighted and license-free; it is free for all uses.

1. General Description:

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.

- 128-bit block
- 128-, 192-, or 256-bit key
- 16 rounds
- Works in all standard modes
- Encrypts data in:
    - 18 clocks/byte on a Pentium
    - 16.1 clocks/byte on a Pentium Pro

Twofish - Performance vs. Other Block Ciphers (on a Pentium)

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

Twofish - Performance on Smart Cards

Results for Twofish on a 6805 CPU, with several different space-time tradeoff options:

RAM, ROM, or EEPROM for Key	Working RAM	Code and Table Size	Clocks per Block	Time per Block @ 4MHz
24	36	2200	26500	6.6 msec
24	36	2150	32900	8.2 msec
24	36	2000	35000	8.7 msec
24	36	1750	37100	9.3 msec
184	36	1900	15300	3.8 msec
184	36	1700	18100	4.5 msec
184	36	1450	19200	4.8 msec
1208	36	1300	12700	3.2 msec
1208	36	1100	15500	3.9 msec
1208	36	850	16600	4.2 msec
3256	36	1000	11900	3.0 msec

Twofish - Hardware Tradeoffs

Gate count	*h* blocks	Clocks per Block	Interleave Levels	Clock Speed	Throughput (Mbits/sec)	Startup clocks	Notes
8000	0.25	324	1	80 MHz	32	20	(1)
14000	1	72	1	40 MHz	71	4	(2)
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	(3)

- Extensively cryptanalyzed
- Unpatented
- Uncopyrighted
- Free

(Bruce Schneier, http://www.schneier.com/twofish-brief.html)

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, they imposed the following performance criteria on their 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 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.

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:

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

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. 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.

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.

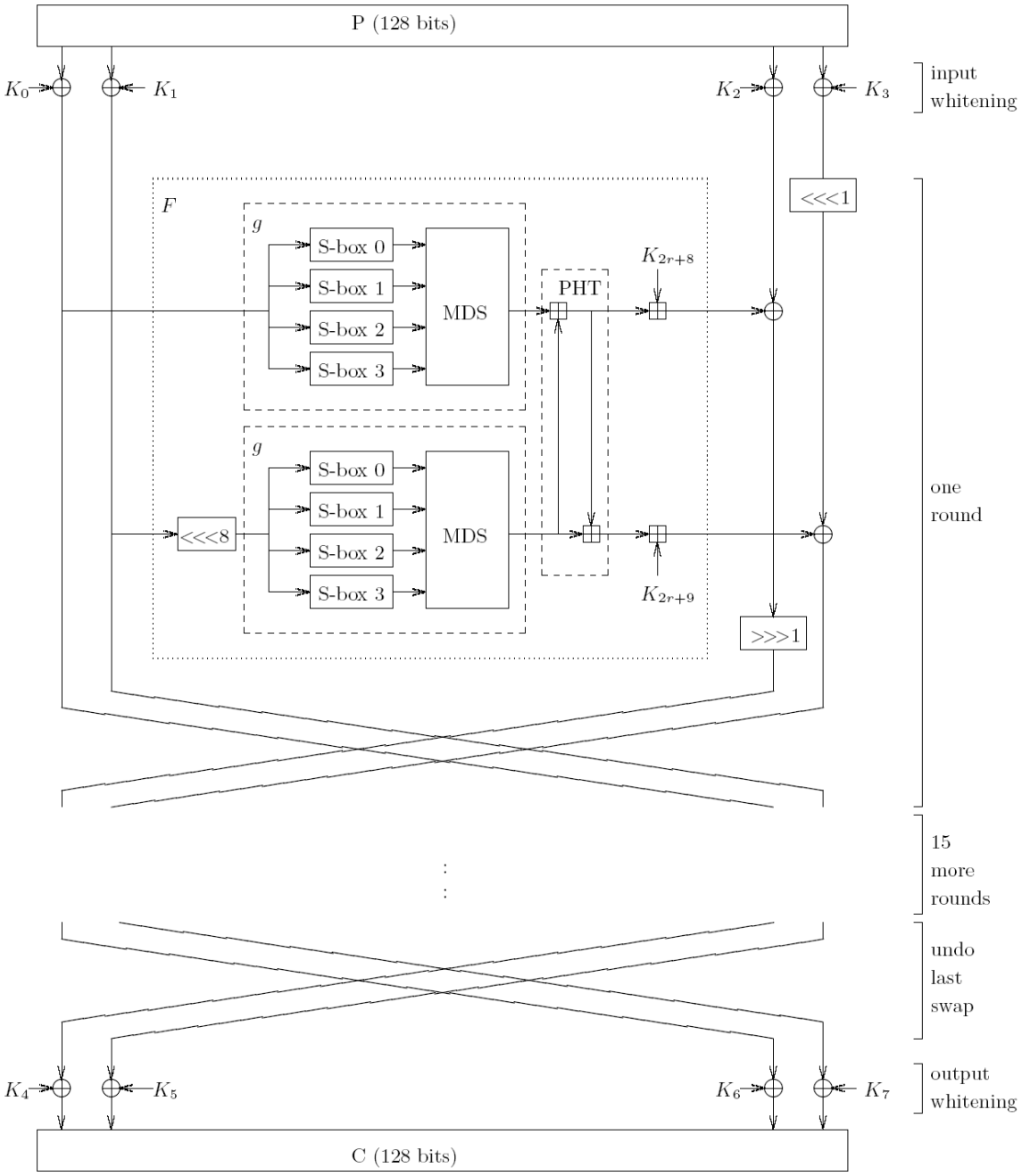

Figure 1: Twofish

In each 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…… p_{15} are first split into 4 words P_0,..., 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…… 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.

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

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 encryption 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.

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

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 *Ki* for $i = 0 \ldots\ldots\ldots 39$ 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 decryption 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

5.2. 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 a 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.

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

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

(Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson,1998)

Bibliography

1. **(Bruce Schneier, http://www.schneier.com/twofish-brief.html)**
2. **(Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson,1998, http://www.schneier.com/paper-twofish-paper.pdf)**