Performance Comparison of the AES Submissions

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

Version 2.0
February 1, 1999

1 Introduction

The principal goal guiding the design of any encryption algorithm must be security. In the real world, however, performance and implementation cost are always of concern. Making the assumption that the major AES candidates are secure (a big assumption, to be sure, but one that is best dealt with in another paper), the most important properties the algorithms will be judged on will be the performance and cost of implementation.

In this paper, we will completely ignore security. Instead, we will compare the performance of the leading AES candidates on a variety of common platforms: 32-bit CPUs, 64-bit CPUs, cheap 8-bit smart-card CPUs, and dedicated hardware. For each platform, we first make some general observations on the performance issues for each of the platforms, then compare the various AES candidates, and finally look at the specific issues for each of the candidates.

2 Performance as a Function of Key Length

The speed (both encryption and key setup) of most AES candidates is independent of key length. That is, the time required to set up a key and encrypt a block of text is the same, regardless of whether the key is 128, 192, or 256 bits long. A minority of the algorithms have different performance characteristics for different key lengths. The results are summarized in Table 1.

Nine algorithms—CAST-256, Crypton, DFC, E2, Frog, HPC, Mars, RC6, and Serpent—are completely independent of key length. Two algorithms—Loki97 and Twofish—encrypt and decrypt independent of key length, but take different times to set up different-size keys[1]. Twofish takes longer to set up a longer key, while Loki97 actually takes less time to set up a longer key than it does to set up a shorter key. Four algorithms—DEAL, Magenta, Rijndael, and SAFER+—encrypt and decrypt at different speeds, depending on key length.

In this paper, we will concentrate on key setup and encryption for 128-bit keys. Obviously, encryption speed is more important than key setup speed. The reader should keep in mind that DEAL and Magenta encrypt 33% slower using 256-bit keys. Rijndael encrypts 20% slower for 192-bit keys and 40% slower for 256-bit keys. SAFER+ has the largest performance degradation for large keys; it is 50% slower for 192-bit keys and 100% slower for 256-bit keys.

3 Performance on 32-bit CPUs

Efficiency on 32-bit CPUs is one of NIST's stated performance criteria. Actually, there are two different 32-bit efficiencies that need to be considered. The first is what NIST delineated: performance on

*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.

[1] Twofish is actually much more complicated. There are ways to implement Twofish where the encryption speed is different for different key lengths, but they are only suitable for encrypting small text blocks. See [SKW+98a, WS98, SKW+99] for more details.

Algorithm Name	Key Setup	Encryption
CAST-256 [Ada98]	constant	constant
Crypton [Lim98]	constant	constant
DEAL [Knu98]	increasing	128,192: 6 rounds 256: 8 rounds
DFC [GGH+98]	constant	constant
E2 [NTT98]	constant	constant
Frog [GLC98]	constant	constant
HPC [Sch98]	constant	constant
Loki97 [BP98]	decreasing	constant
Magenta [JH98]	increasing	128,192: 6 rounds 256: 8 rounds
Mars [BCD+98]	constant	constant
RC6 [RRS+98]	constant	constant
Rijndael [DR98a]	increasing	128: 10 rounds 192: 12 rounds 256: 14 rounds
SAFER+ [CMK+98]	increasing	128: 8 rounds 192: 12 rounds 256: 16 rounds
Serpent [ABK98a]	constant	constant
Twofish [SKW+98a]	increasing	constant

Table 1: Speed of AES Candidates for Different Key Lengths.

the high-end Pentium Pro (and similar) CPUs that are likely to dominate desktop computing for the next few years. To be sure, this is an arbitrary measure—in ten years, using Pentium Pro performance as a yardstick will seem as quaint as comparing algorithm performance on 80286 chips does today—but it is a good measure of relative high-end performance. Even when 64-bit CPUs become commonplace, algorithms that are the fastest on the Pentium Pro are likely to remain the fastest on these new microprocessors.

The second is performance on low-end 32-bit CPUs—such as the 80386 and 68000 variants, as well as various simple RISC chips—that will increasingly replace 8-bit CPUs on high-end smart cards and computing-intensive embedded applications.

Performance on these commodity 32-bit CPUs is much more important than performance on the Pentium Pro/II. Over the course of AES's lifetime, the architecture of desktop computing will likely change beyond recognition. The Pentium Pro/II architecture has oddities not present in other 32-bit CPUs and also not present in Merced (now called Pentium III) and other 64-bit CPUs. But just as 8-bit CPUs are still being used today (and will still be used tomorrow), commodity 32-bit CPUs will find their particular price/performance "sweet spot" in widely deployed embedded applications. Performance on these CPUs will still be important a decade from now.

3.1 Comparing Performance on 32-bit CPUs

We looked at the performance of the major AES candidates on both the Pentium and the Pentium Pro. We either used the data in the candidate's AES submission documentation or, where such data was unavailable or unreliable, calculated our own or used the data from Brian Gladman's web page [Gla98]. Where we were unsure if a particular optimization technique would work, we gave the algorithm the benefit of the doubt. When there were multiple sources of data, we took the fastest. The results are shown in Table 2. Numbers derived from estimates and not from an actual implementation are marked with an asterisk. Crypton and Rijndael have very different key setup speeds for encryption and decryption. Rijndael's encryption key setup is 300 clocks, and its decryption key setup is 1400 clocks. Crypton's encryption and decryption key setups are 540 and 1370 clocks, respectively. For all tables except

Algorithm Name	Key Setup Pentium Pro C (clocks)	Encrypt Pentium Pro C (clocks)	Encrypt Pentium Pro ASM (clocks)	Encrypt Pentium ASM (clocks)
CAST-256	4300	660	600*	600*
Crypton	955	476	345	390
DEAL	4000*	2600	2200	2200
DFC	7200	1700	750	?
E2	2100	720	410	410*
Frog	1386000	2600	?	?
HPC	120000	1600	?	?
Loki97	7500	2150	?	?
Magenta	50	6600	?	?
Mars	4400	390	320*	550*
RC6	1700	260	250	700*
Rijndael	850	440	291	320
SAFER+	4000	1400	800*	1100*
Serpent	2500	1030	900*	1100*
Twofish	8600	400	258	290

Table 2: AES Candidates' Performance with 128-bit Keys on Pentium-Class CPUs.

that for hashing, we used the average.

The C performance numbers are included because they were required by NIST, and were included in each of the algorithms' submission documentation. Since in any application where encryption speed is a potential bottleneck, the algorithm will be hand-coded in optimized assembly language, we primarily focused on assembly language implementations. The algorithm is a discrete and well-defined chunk of code with a single entry point that needs to run fast: a perfect example of something that should be coded in assembly. [2]

This comparison is for 128-bit keys; as noted earlier, some algorithms have a running time that depends on the size of the key. We concentrated on 128-bit keys, as these are the fastest (key setup in Loki97 is the only exception), and will provide adequate security for virtually any application for the next several decades (at least).

The AES submissions vary greatly in their 32-bit CPU performance, from 250 clock cycles per block (RC6) to 6600 (Magenta). Some candidates' performances depends heavily on the particular details of the 32-bit CPU (RC6 and Mars), while others are largely CPU-independent.

For bulk encryption, the fastest algorithms on general 32-bit CPUs are (in order) Twofish, Rijndael, Crypton, and E2, followed by Mars and RC6. Both Mars and RC6 are significantly faster on 32-bit CPUs with fast 32-bit multiplications and variable rotations, like the Pentium Pro and Pentium II, but are significantly slower on CPUs without these features. In fact, both of these algorithms also run more slowly than the four fastest on a DEC Alpha. Restricting the comparison to the Pentium Pro/II (the NIST reference platform), the fastest algorithms are (in order) RC6, Twofish, Mars, Rijndael, Crypton, and E2. All candidates other than these six are significantly slower across all 32-bit CPUs.

For comparison, on a Pentium, DES encrypts at 43 clock cycles per byte, IDEA at 74 clock cycles per byte, 16-round RC5 at 25 clock cycles per byte, and Blowfish at 20 clock cycles per byte [PRB98].

[2] For fastest assembly-language performance on current Intel CPUs, Twofish's implemementation compiles key-dependent data directly in the encryption and decryption code. In this implementation, a separate copy of the executable code is stored for each key along with the key-dependent S-box tables. This copy of the code is modified by patching immediate values in the code with actual subkey values, removing the need to load the subkey from data cache during execution and freeing up registers. Although this key-specific code could be technically considered to be "self-modifying," the code is totally reentrant, and the process is much more akin to compilation.

This compilation trick seems to be helpful only in register-poor CPUs like the Pentium or Pentium Pro, where the availability of an extra register can make a significant difference in performance. Most other AES candidates could also benefit somewhat from using a similar compiled mode, but the resulting gains are believed to be fairly small (considerably less than 10%). Twofish has been specially designed to take advantage of this compiled option, achieving about 20% faster execution than a naive assembly language implementation on a Pentium Pro.

For encryption of short blocks of plaintext, it makes sense to look at both encryption speed and key setup speed. Again, RC6's and Mars's reliance on CPU-specific operations means that we need to consider general 32-bit CPUs and the Pentium Pro/II separately. Table 3 compares the speed to key and encrypt of six of the AES candidates for different size texts on the Pentium (and by extension, most general-purpose 32-bit CPUs.

To calculate key setup speeds in assembly, we estimated assembly-language key setup time by taking the percentage speedup between C and assembly encryption on the Pentium Pro/II, and applying that to the assembly-language key setup.

Rijndael is the fastest algorithm for small blocks of text, followed closely by Crypton. What is surprising, though, is how quickly all the algorithms converge to their raw encryption speed. For 1024-byte texts—the size of a very small e-mail message—all the algorithms are within 15%. For 4096-byte texts—the size of a more reasonable e-mail message—the speed ordering of the algorithms is the same as it would be if the key schedule were not taken into account. Note that the AES candidates not listed here are slower.

Table 4 compares the speed to key and encrypt of seven of the AES candidates for different size texts on the Pentium Pro/II. The primary difference of note is how much faster RC6 and Mars are on this one platform.

The speed of the key schedule becomes even more important when we look at the algorithm as a hash function. There are many hash function constructions that rely on block ciphers as primitives, and all involve changing the encryption key with every block. Table 5 summarizes this data.

These values are the sum of the number of clock cycles required to set up a key plus the number of clock cycles required to encrypt one block. Starting with the data in Table 2, we calculated the Pentium Pro/II assembly key setup time by multiplying the C key setup time by the percentage speed improvement between the encryption speed in C and assembly; this is a reasonable first estimate of the speed improvement of taking the key setup from C to assembly. Then we added in the encryption speeds in both Pentium Pro/II and Pentium. While the real performance numbers are likely to be somewhat different, this is a reasonable first estimate.

The fastest hash function is Rijndael, followed closely by Crypton.[3] These two algorithms are particularly suited as hash functions because key setup in the encryption direction is much faster than key setup in the decryption direction. E2, RC6, and Twofish are slower, but also provide acceptable hash function performance. SAFER+ and Serpent are marginal.

All of these speeds are significantly slower than dedicated hash functions. SHA-1, for example, hashes at a rate of 13 clock cycles per byte on a Pentium, and RIPEMD-160 hashes at 16 clock cycles per byte on the same machine [PRB98]. Thus, it is likely that AES will be used as a hash function only when space (either code space in software or gates in hardware) is at a premium.

3.2 Algorithm-Specific Comments on 32-bit Performance

3.2.1 CAST-256

CAST-256 is an incomplete Feistel network. Each round is fast—17 clock cycles—but 48 rounds gives a total speed of 815 clock cycles per block. Hence, CAST-256 is about three times slower than the fastest AES candidates, even though it uses mostly 32-bit operations. CAST's use of only simple RISC operations implies that the relative performance of CAST-256 will be fairly uniform across processor types.

3.2.2 Crypton

Crypton is a very clever enhancement to the Square algorithm [DKR97]. It is structured so that encryption and decryption are identical operations (with reversed subkey schedules), and there are two separate 8-by-8-bit S-boxes used in the transformation, which are easily built from 4-bit permutations. Crypton replaces Square's MDS matrix with a simpler (and admittedly less powerful) diffusion operation that can be implemented very cheaply in hardware and on smart cards. Perhaps because the S-boxes and diffusion are weaker than in Square, Crypton consists of twelve rounds instead of eight for Square (and ten, twelve, or fourteen for Rijndael).

An optimized version on a 32-bit CPU looks identical to Square except the fixed lookup tables are different, so performance scales based on the number of rounds. Encryption speed is 390 clocks per block in Pentium/Pentium Pro assembler, and is uniform across different 32-bit CPUs.

[3] Previous versions of this report mentioned possible weaknesses in Crypton's key schedule, making it unsuitable as a hash function. In the interest of fairness, we withhold judgment on this issue pending better documentation of the weaknesses.

Text Size (bytes)	Crypton	E2	Mars	RC6	Rijndael	Serpent	Twofish
16	73	100	260	146	59	205	175
32	49	63	147	95	39	137	119
64	37	44	91	69	30	103	91
128	30	35	63	57	25	86	70
256	27	30	48	50	22	77	48
512	26	38	41	47	21	73	38
2^{10}	25	27	38	45	21	71	31
2^{11}	25	26	36	45	20	70	25
2^{12}	25	26	35	44	20	69	22
2^{13}	24	26	35	44	20	69	21
2^{14}	24	26	35	44	20	69	20
2^{15}+	24	26	34	44	20	69	19

Table 3: Clock Cycles, per Byte, to Key and Encrypt Different Text Sizes on a Pentium.

Text Size (bytes)	Crypton	E2	Mars	RC6	Rijndael	Serpent	Twofish
16	70	100	246	118	53	193	132
32	46	63	133	67	36	125	93
64	34	44	76	41	27	90	73
128	28	35	48	28	23	73	64
256	25	30	34	22	20	65	48
512	23	28	27	19	19	61	33
2^{10}	22	27	24	17	19	58	25
2^{11}	22	26	22	16	18	57	20
2^{12}	22	26	21	16	18	57	18
2^{13}	22	26	20	16	18	57	17
2^{14}	22	26	20	16	18	56	17
2^{15}+	22	26	20	16	18	56	16

Table 4: Clock Cycles, per Byte, to Key and Encrypt Different Text Sizes on a Pentium Pro/II.

3.2.3 DEAL

DEAL has basically the same performance characteristics as triple-DES. It is a six-round Feistel network with DES itself used as the round function. The Pentium and Pentium Pro performance is thus about 2000 clocks per block. In a sense, DEAL is a straw-man algorithm that provides a worst-case performance benchmark for AES.

3.2.4 Decorrelated Fast Cipher

DFC is an eight-round Feistel cipher, in which the round function includes a 64-bit modular addition and multiplication over $2^{64} + 13$, as well as a single small (6-by-32-bit) S-box lookup. The modular multiply does a nice job of diffusing bits, but it also hurts performance significantly. Despite the low number of rounds, DFC takes 750 clocks per block in assembly language on a Pentium Pro. This is about the speed of DES and nearly three times slower than the fastest candidates. Due to the heavy reliance on multiply operations, we expect the performance on general 32-bit CPUs to be significantly worse. With only eight rounds, the speed vs. security tradeoff for DFC feels mismatched compared to most of the other AES candidates.

3.2.5 E2

E2 is a twelve-round Feistel cipher, with each round function including two levels of subkey XOR and 8-by-8-bit S-box lookup, with linear mixing in between. Only simple RISC operations are used, so the performance should be relatively uniform across

Algorithm Name	Hash Speed Pentium Pro ASM (clocks)	Hash Speed Pentium ASM (clocks)
CAST-256	282*	282*
Crypton	46*	49*
DEAL	349*	349*
DFC	245*	?
E2	100*	100*
Frog	?	?
HPC	?	?
Loki97	?	?
Magenta	?	?
Mars	246*	260*
RC6	118*	146*
Rijndael	32*	34*
SAFER+	193*	212*
Serpent	193*	205*
Twofish	132	175

Table 5: Hash-Function Performance, per Byte, of AES Candidates (128-bit key) on Pentium and Pentium Pro/II.

processor types. The speed of E2 is reasonable at 410 clocks per block in assembly language (making it the fourth fastest on the Pentium and the sixth fastest on the Pentium Pro/II), and is uniform across different 32-bit CPUs.

3.2.6 Frog

Frog has a byte-oriented structure. It seems that any implementation has to operate on individual bytes, which means that Frog cannot take advantage of 32-bit operations. This hurts the performance on larger CPUs. Frog's key schedule is by far the slowest of any AES candidate.

3.2.7 Hasty Pudding Cipher

HPC was optimized for 64-bit CPUs. The heavy use of rotations of 64-bit words are difficult to implement (and therefore slow) on most 32-bit CPUs.

3.2.8 Loki97

Loki97 is one of the few algorithms that uses a bit-level permutation. Although these are very fast in hardware, their cost in software is similar to that of a full S-box layer. Performance on 32-bit CPUs is poor compared to most other candidates, and may be slower than triple-DES.

3.2.9 Magenta

Magenta is, by far, the slowest of any of the AES candidates. The core of the algorithm is formed by a byte-level FFT-like structure. This seems to force any 32-bit implementation to use 8-bit manipulations, which is generally inefficient. Magenta also has a very large number of operations per round. Each Π function contains 16 8-bit table lookups. There are four Π functions in each T, and three T functions in each round for a total of 192 S-box lookups per round, and 1152 per encryption with a 128-bit key. (As a comparison, Frog uses a total of 128 S-box lookups per encryption.) There seems to be no way of implementing Magenta in software at a speed comparable to the other candidates.

3.2.10 Mars

Mars is a fast cipher that relies on a veritable "kitchen sink" of primitive operator elements, including rotation, multiplication, and 8-by-32-bit S-box lookups. Mars has the fastest claimed speed for a C version (390 clocks per block) on the Pentium Pro of any of the AES candidates. However, this speed advantage is largely due to the use of a different optimizing compiler (DJGPP), which would probably benefit other candidates dramatically as well; using the standard AES compiler (Borland), Mars is fairly slow at 900 clocks/block.

Mars encryption has three phases: an initial forward mixing, the cryptographic "core," and a final mixing. The mixing phases are unkeyed (except for input/output whitening) and constitute roughly two cryptographic cycles each. The cryptographic core consists of sixteen rounds of an unbalanced Feistel cipher (totaling roughly four cycles), with each round using a 32-bit to 96-bit keyed expansion function E. Both fixed and data-dependent rotations are used extensively throughout the cipher. A careful study of the encryption round reveals a fairly long critical path through the E function that limits the throughput. On a Pentium Pro, this critical path appears to be at least 12 clock cycles, which leads to a core processing time of at least 192 clocks per block. With the forward and reverse mixing requiring at least 100 clocks, the overall speed of this algorithm in assembler on a Pentium Pro appears to be no faster than 310-320 clocks. Because of its use of data-dependent rotation and multiplication, the performance of Mars on a Pentium or Pentium MMX chip is considerably slower: about 550 clocks/block.

As with RC6, the relative performance of Mars across processor types will vary considerably, and timing attacks are also of concern in implementations where multiplies and data-dependent rotations do not take constant time. Compared to RC6, Mars is somewhat slower and far less elegant, yet it retains most of the performance concerns discussed below for RC6.

3.2.11 RC6

RC6 is by far the most elegant and easily understood AES candidate. It is remarkably simple, as has become almost expected of Rivest's ciphers (e.g., RC4, RC5). On the AES target platform (Pentium Pro/Pentium II), it is also the fastest algorithm, at about 250 clocks per block in assembly language. The unrolled code size of the encryption routine is very modest, and a looping version with very good performance can be implemented with an extremely small code size (under 200 bytes on the Pentium family). The performance in C is very good, even though it is hurt considerably by the lack of intrinsic support for rotations in most C compilers.

However, RC6 does not perform anywhere near as well on many other common platforms. For example, on the Pentium (or Pentium MMX) chip, multiplication and variable rotation opcodes do not pair, and they require 10 and 4 clocks, respectively. Thus, the time required for multiplication and rotation alone is 560 clocks per block on a Pentium, with the other RC6 operations increasing this total to at least 700 clocks for an optimized assembly version. Thus, RC6 is nearly three times slower on a Pentium than on a Pentium Pro. By comparison, most other candidate algorithms have identical or nearly identical performance on the two platforms. On a 386/486 (and many embedded CPUs), the multiplication and rotation speed is also fairly slow. In general, because it does not rely on "basic" RISC instructions as most other AES candidates do, RC6's relative performance will be much less uniform across processor types. This makes it much less attractive for low-end 32-bit CPUs.

The fact that many processors (e.g., 486, 68000, smart cards) have data-dependent execution times for multiplication/rotation is also a concern, since RC6 could potentially leak key information to a timing attack. Constant-time implementations of data-dependent rotations on those platforms is possible, but at a significant speed penalty.

3.2.12 Rijndael

Rijndael is another variant of Square: a fast cipher that works very well across all platforms. It boasts a clean mathematical structure involving only table lookup and XOR operations. Although the encryption and decryption algorithms are not exactly identical, their general structure and performance are virtually indistinguishable.

Rijndael's assembly language on both the Pentium and Pentium Pro processors is about 300 clocks per block. Unlike RC6 and Mars, there are no known CPU platforms (8-bit or 32-bit) on which Rijndael's relative performance would be unduly negatively affected or on which timing attacks would be possible.

3.2.13 SAFER+

SAFER+ is obviously designed for 8-bit CPUs. Every operation in SAFER+ is a simple RISC operation, but there are no 32-bit operations anywhere. This lack of 32-bit operations, particularly in the diffusion phase involving the PHT and the "Armenian shuffle," significantly hurts the relative performance of this algorithm on modern CPUs. A rough estimate is that SAFER+ with a 128-bit key requires 800 clocks per block on a Pentium Pro in assembly, which is slower per-byte than DES. On a Pentium, the speed slows to about 1100 clocks per block. The throughput decreases further for the larger key sizes; in fact, SAFER+ with a 256-bit key is slower than triple-DES. As an SP-network, SAFER+ with a 128-bit key has only one-fourth the number of cycles of Serpent, yet it is approximately the same speed. In general, the speed vs. security tradeoff for SAFER+

feels mismatched compared to most of the other AES candidates.

3.2.14 Serpent

Serpent is an extremely conservative SP-network cipher, carefully designed to allow "bit-slice" implementations on 32-bit CPUs. Its authors claim that Serpent's performance on a Pentium Pro is at least as fast as DES [ABK98a], but this claim involves C implementations, which are always subject to the vagaries of compiler optimization.

In assembly language, Serpent is actually considerably slower than a good DES implementation, which can run at about 45 clocks/byte on a Pentium. A conservative estimate, counting only ALU operations and assuming (very optimistically) that all memory accesses can be accomplished for "free," shows that Serpent requires at least 900 clocks per block on a Pentium Pro (56 clocks/byte); in fact, 960 clocks seems like a more likely practical bound. On a Pentium, this number rises to at least 1100 clocks per block. An optimized C version can achieve about 1020 clocks per block on a Pentium Pro, proving that Serpent compiles very well from C (much better than does DES)—which is an interesting fact but hardly of great importance here.

The fact remains that the best Serpent software implementation will be 3–4 times slower than other candidates, including Mars, Twofish, RC6, and Rijndael. Thus, even a non-conservative version of Serpent with only sixteen rounds would be considerably slower than these other algorithms. The nature of the algorithm, which uses basic RISC operations exclusively, strongly suggests that its relative performance will not be negatively affected on any platform.

3.2.15 Twofish

Twofish is the fastest AES candidate on a Pentium (and other 32-bit CPUs), and is second-fastest on the Pentium Pro/II. Its key setup is about average (eight are faster, and six are slower); however, for applications where key setup is a large factor in encryption speed (for small amounts of text), there are alternate implementations of Twofish that trade off encryption speed for key setup speed. As a hash function, for example, Twofish is much more efficient using one of these alternative implementations. See [SKW+98a] for details.

Twofish uses only basic RISC operations, and Twofish's performance on other 32-bit platforms will be comparable.

3.3 Comparing Performance of "Minimal Secure Variants"

In [Bih98], Biham introduced the notion of comparing the algorithms based on their "minimal secure variants." Different design teams were more or less conservative than each other; the number of rounds in their final submissions was not a fixed percentage of the number of rounds they could successfully cryptanalyze. Biham tried to normalize the algorithms by determining the minimal number of rounds that is secure (either as described by the designers or other cryptographers, or Biham's "best guess"), and then added a standard two cycles.

We do not believe that this measurement is a fair way to compare algorithms. Two examples illustrate this. One, Biham maintains that the minimal number of rounds for DFC is 7 and for Serpent is 15. It seems unreasonable to add two rounds to each, when that is a 29% increase for DFC and only a 13% increase for Serpent. The comparative analysis clearly shows that one round of DFC is stronger than one round of Serpent. Two, Biham's measures don't take into account initial and final transformations for some of the ciphers: e.g., the whitening in Twofish and the initial/final transformations in E2. At least in Twofish's case, the transformations have the effect of adding a round to the cipher without the performance penalty.

Aside from the fairness issue, we also question the usefulness of this measure: we feel that each algorithm should be judged on its own merits, and that changing the number of rounds is no different than changing the structure of the rounds. Also, NIST has not indicated that they would allow algorithms to be selected with modifications.

And finally, in some cases we even question Biham's choice of values. Being about to break ten rounds of Twofish, for example, seems overly optimistic. If such a metric seems useful, we recommend that several cryptanalysts be polled for their opinions.

Regardless of these concerns, we include Table 6, based on Biham's numbers. We urge caution in using this data. Biham also presented a table of relative performance, but he used speeds for compiled C code [Bih98], which makes no sense for any application where speed is important.

4 Performance on 64-bit CPUs

We have some data regarding performance on the Alpha in Table 7. Aside from DFC and HPC, these

Algorithm Name	Rounds	Minimal Secure Rounds	MSR Encrypt Pentium Pro ASM (clocks)	MSR Encrypt Pentium ASM (clocks)
CAST-256	48	40	500*	500*
Crypton	12	11	316	358
DEAL	6	9	3300	3300
DFC	8	9	844	?
E2	12	10	342*	342*
Frog	8	?	?	?
HPC	8	?	?	?
Loki97	16	> 36	?	?
Magenta	6	> 10	?	?
Mars	32	20	200*	344*
RC6	20	20	250	700*
Rijndael	10	8	233	256
SAFER+	8	7	700*	963*
Serpent	32	17	478*	584*
Twofish	16	12	194	218

Table 6: Minimal Secure Round Performance of AES Candidates with 128-bit Keys on Pentium-class CPUs.

Algorithm Name	Cycles
CAST-256	600
Crypton	408
DEAL	2528*
DFC	304
E2	471
Frog	?
HPC	376
Loki97	?
Magenta	?
Mars	478
RC6	467*
Rijndael	340*
SAFER+	656
Serpent	915
Twofish	360*

Table 7: AES Candidate Performance on the DEC Alpha.

numbers are based on analyzing the algorithms and studying data sheets and not on actual implementations [Alm99]. Perhaps the most interesting thing is that, for many of the AES candidates, the Alpha is actually slower, in terms of clocks per block, than the Pentium II. DFC is the fastest algorithm on the Alpha, followed by Rijndael, Twofish, HPC, and Crypton. All the other submisions are considerably slower.

4.1 RC6

RC6's performance on the DEC Alpha processor is also hurt by its reliance on multiplications and rotations. The 21164 can run up to 700 MHz (or faster) and has a 4-way superscalar architecture. The Alpha can perform a signed 32-bit multiply in 8 clocks, and a second multiply operation can be started 4 clocks after the first, so two of them take 12 clocks total. This multiply opcode should work for RC6's unsigned multiplies, assuming that the signed multiply overflow is properly handled; otherwise, we would need to use the slower 64-by-64-bit multiply, adding 8 more clocks. The remainder of the RC6 round logic takes an additional 11 clocks, due in no small part to the lack of rotation opcodes, which must be synthesized out of shifts. Thus, an RC6 round requires a total of 23 clocks on the Alpha. At twenty rounds, this is 460 clocks per block, about twice the clock count on a Pentium II. In other words, a 600 MHz Alpha runs RC6 at a slower absolute speed (in Mbits/s) than a 400 MHz Pentium II!

5 Smart-Card Performance

NIST also requires AES to be efficient on smart cards.

Smart cards with 8-bit CPUs tend to be used in applications that are not speed-critical. It is usually the I/O, or other system components, that is the limiting factor in performance and not the cryptography. This having been said, the speed of almost all the candidates on common 8-bit CPUs seems to be acceptably fast, given certain assumptions. We hope that some other group implements the AES front-runners on a single common smart-card CPU, so that their performance can be better compared.

There are some applications that require fast encryption operations on an 8-bit CPU, such as automated toll collection and public transport tickets. As a rule of thumb, a cipher that is used in these applications should be as least as fast as DES on an 8-bit CPU.

For 32-bit smart cards, the Pentium performance numbers give a reasonable relative performance metric (although Mars and RC6 are likely to be relatively slower, due to their use of 32-bit multiplies).

We feel that it is more important to compare how well each algorithm fits on a smart card CPU than to measure exactly how fast it runs. One major concern is how much RAM the algorithm requires. Most commodity 8-bit smart-card CPUs today include from 128 to 256 bytes of on-board RAM. Each CPU family typically contains members with more RAM and a correspondingly higher cost. Several of the algorithms have no way to compute subkeys on the fly, thus requiring that all the subkeys be precomputed whenever a key is changed. The subkeys consume from 150 to 256 bytes of "extra" RAM; more than is available on many commonly used CPUs. Although some CPUs include enough RAM to hold the subkeys, it is often invalid to assume that such a large fraction of the RAM can be dedicated solely to the encryption function. Obviously, if an algorithm does not fit on the desired CPU, with its particular RAM/ROM configuration, its performance on that CPU family is irrelevant.

For some of the candidates, the performance or RAM requirements can depend on whether encryption or decryption is being performed. It is tempting to consider only one of the two operations, using the argument that the smart card can perform the more efficient side of the operation, and the (larger) terminal can perform the less efficient side. Experience shows that this does not work. Many smart card terminals contain a secure module; in many cases this secure module is itself a smart card chip. In several applications it is a requirement that two smart cards execute a protocol together, and many existing protocols use both encryption and decryption on the same smart card.

5.1 Comparing Performance on Smart Cards

Each submission discussed smart-card performance, but it is fairly difficult to compare the speeds given in the documentation. Most of them involved different CPUs, different clock rates, and different assumptions about subkey precomputation vs. on-the-fly computation.

Table 8 compares the RAM requirements for the different AES submissions. Again, the starred values are estimates. These numbers assume that the key must be stored in RAM, and that the key must still be available after encrypting a single block of text. The results seem to fall into four categories:

Algorithm Name	Smart Card RAM (bytes)
CAST-256	60*
Crypton	52*
DEAL	50*
DFC	200
E2	300
Frog	2300+
HPC	?
Loki97	?
Magenta	?
Mars	195*
RC6	210*
Rijndael	52
SAFER+	50*
Serpent	50*
Twofish	60

Table 8: AES Candidates Smart-Card RAM Requirements.

algorithms that can fit on any smart card (less than 128 bytes of RAM required), algorithms that can fit on higher-end smart cards (between 128 and 256 bytes of RAM required), algorithms that can only fit on high-end smart cards (more than 256 bytes of RAM required), and algorithms that can't fit on smart cards (Frog)[4].

Even if an algorithm fits onto a smart card, it should be noted that the card functionality is much more than block encryption, and the encryption application will not have the card's entire RAM capacity available to it. So, while an algorithm that requires about 200 bytes of RAM can theoretically fit on a 256-byte smart card, it probably won't be possible to run the smart-card application that calls the encryption. For many applications, a RAM requirement of more than 64 bytes just isn't practical.

Given all these considerations, the only algorithms that seem to be suitable for widespread smart-card implementation are CAST-256, Crypton, DEAL, Rijndael, SAFER+, Serpent, and Twofish.

5.2 Algorithm-Specific Comments on Smart-Card Performance

5.2.1 CAST-256

CAST-256 has an on-the-fly key schedule, meaning that it can fit well into the RAM of inexpensive CPUs. However, it should be noted that the subkey generation on 32-bit CPUs takes about four times as long as a block encryption, meaning that the overall performance suffers significantly when on-the-fly subkey generation is used. Given that the algorithm is already fairly slow, this additional overhead may push the smart card performance of CAST-256 out of the range of acceptability.

To return the key to a usable state, the algorithm will either have to make a copy of the key (at the cost of extra RAM), or "unwind" the key schedule (which takes as long as computing it in the first place). To perform a decryption, the key schedule has to be performed in a forward direction, and then unwound during the decryption operation.

The ROM usage for CAST-256 is a point of concern; the S-boxes alone require 4K bytes of ROM. Adding the code probably will push the total close to 6K bytes of ROM, which rules out the least expensive members of almost all CPU families.

5.2.2 Crypton

Crypton should also fit on smart cards. The code size, including two full 8-by-8-bit S-box tables, should be easily under 2K bytes. Additionally, if ROM size is important, the S-boxes can be computed using 4-by-4-bit S-boxes at a considerable cost in speed, saving 300–400 bytes. Unlike Rijndael,

[4] We do not consider the solution of using EEPROM to store the expanded key, as this is not practical in many smart card protocols that require the use of a session key. Some algorithms, like Twofish and Rijndael, have significantly reduced RAM requirements if this solution is used.

these estimates include both encryption and decryption. The RAM usage of Crypton can also be quite small (52 bytes), or it can precompute subkeys and use about 230 bytes of RAM to achieve higher speed.

5.2.3 DEAL

DEAL fits on smart cards as well as DES does; i.e., reasonably well. The on-the-fly key schedule of DES means that RAM usage is minimal, and ROM should be comfortably under 2K bytes. However, the speed of DEAL will be basically that of triple-DES, which is quite a bit slower than that of the other AES candidates.

For decryption, DEAL suffers from the key schedule, which can be computed in a forward direction only. Thus DEAL either has to use an expanded key at the cost of extra RAM, or compute the key schedule forward first, and than backward during the actual decryption. Of course it should be noted that calculating DEAL keys requires invocations of DES, and hence is very slow to compute on the fly.

5.2.4 Decorrelated Fast Cipher

As with RC6 and Mars, performance will be poor on 8-bit CPUs without a multiply instruction, but for most CPUs of interest this seems not to be a problem. The 64-bit multiply used by DFC is not very fast on most smart cards. Even if they have a multiply instruction, it typically takes around 10 clock cycles for an 8-by-8-bit multiply. The overhead of adding the results of the small multiplies together is even larger than this in many situations. Because DFC uses only 8 rounds, the performance with an expanded key is not as slow as one would expect. The penalty for generating the key on the fly is unfortunately fairly large.

The code and table for DFC fit in less than 2K bytes of ROM. Using on-the-fly subkey generation, less than 100 bytes of RAM is required, but the speed is one-sixth the speed of using precomputed subkeys (200 bytes of RAM).

As with many of the other candidates, the DFC key schedule can only be generated in a forward direction. Decryption without an expanded key thus requires the key schedule to be computed forward and then backward again during the actual decryption.

Further information about DFC and smart cards can be found in [PV98]

5.2.5 E2

On-the-fly subkey generation was explicitly precluded as part of the design criteria of E2, and the subkeys alone require 256 bytes of RAM. Thus, the algorithm takes about 300 bytes of RAM, ruling out the vast majority of commonly used smart card CPUs. This fact strongly argues against E2's acceptability as a widespread standard for smart card use.

Unfortunately, no ROM size estimates are given in the submission documents, but we estimate that the code and tables for E2 should fit comfortably in less than 2K of ROM.

5.2.6 Frog

Frog's round structure is very suitable for implementation on smart cards. Unfortunately, the key schedule requires over 2300 bytes of RAM. For almost all smart card applications used and proposed today, Frog would not be a suitable block cipher.

5.2.7 Hasty Pudding Cipher

Hasty Pudding was optimized for 64-bit CPUs. Implementation on a smart card looks difficult. The expanded key table is 256 entries of 64 bits each, which requires 2K bytes of RAM. As with Frog, this seems to make HPC very unsuitable for practical smart card applications.

5.2.8 Loki97

The Loki97 round uses two S-boxes, one of which has 13 input bits. A full table of this S-box requires 8K bytes of ROM. Most implementations will probably choose not to store the S-boxes but instead generate the required entries on the fly using the definition. This is very inefficient, since it involves multiplication over a finite field. This can be implemented with tables, but then it would be just as big as the S-box table the implementation is trying to avoid. Doing the multiply using shift-and-XOR would slow down the algorithm by a factor of five to ten.

An expanded key requires nearly 400 bytes of RAM. As this is impractical in most situations, we will assume that the key is generated on the fly. As with many other ciphers, the key schedule can only be computed in the forward direction. Decryption thus requires the key schedule to be computed twice.

We do not have any reasonable numbers for the speed of Loki97 on a smart card. With an expanded key, it looks to be slower than DES (per byte). Without an expanded key, it is about four times as slow.

5.2.9 Magenta

Magenta's structure is well-suited to implementation on smart cards. The byte structure provides a natural implementation on 8-bit CPUs. The memory requirements are very limited, and there is no key schedule to compute. However, due to the very large number of operations in Magenta, the resulting performance is still relatively low.

5.2.10 Mars

The ROM size for Mars is of some concern, particularly since it requires a 2K byte ROM just for the S-box. The size of this table alone is larger than the total ROM usage of most other AES candidates. No estimates of code/table size are given in the Mars submission paper [BCD+98], but certainly at least 3K bytes would be required (and probably more). This size is not absolutely fatal, but it certainly rules out some cost-sensitive applications.

We find the discussion on smart card performance in the Mars submission paper to be basically useless, since only a very hypothetical CPU is discussed (unlike any actual 8-bit CPU we've ever seen). The comment in the Mars paper that smart cards are "typically equipped with a dedicated [i.e., hardware] crypto unit" seems to evade the issue. Any algorithm can run in dedicated hardware; the whole point is to allow the AES algorithm to run in software on the smart card without requiring custom hardware design to be added to an otherwise off-the-shelf CPU. Furthermore, the assertion is not correct. Although there are many smart cards with a crypto unit, the great majority of these are only designed for public-key algorithms. There are very few smart cards that have hardware support for DES or a similar block cipher.

As with RC6, a minor concern is that performance will be poor on 8-bit CPUs without a multiply instruction, but for most CPUs of interest this seems not to be a problem. Even with a multiply instruction, the multiply will be one of the slow parts of Mars, as each 32-by-32-bit multiply has to be built up from 10 separate 8-by-8-bit multiplies.

Mars has no on-the-fly subkey generation and requires 160 bytes of subkey RAM. When added to the 16 bytes of plaintext and 16 bytes of key, plus other scratch variables, it appears that about 200 bytes of RAM are required. This amount of RAM rules out most commodity smart card CPUs. The Mars team has pointed out that it is theoretically possible to compute subkeys on the fly, but the performance of this approach is so poor as to rule it out as an effective alternative.

The use of multiplications and variable rotations in Mars raises a concern about possible timing attacks, because it is almost guaranteed that one of these two operations will have variable timing on 8-bit CPUs. Avoiding these variabilities in the timing will result in a speed and/or size penalty.

5.2.11 RC6

RC6's small code size should carry over very nicely into the smart card world. A minor concern is that performance will be poor on 8-bit CPUs without a multiply instruction, but for most CPUs of interest this seems not to be a problem. Still, even on 8-bit CPUs that do have a multiply instruction, RC6 is not among the fastest candidates.

RC6 has no on-the-fly subkey generation and requires 176 bytes of subkey RAM. When added to the 16 bytes of plaintext and 16 bytes of key, plus other scratch variables, it appears that over 210 bytes of RAM are required. This amount of required RAM means that RC6 cannot fit on most commodity smart card CPUs.

The use of multiplications and variable rotations in RC6 raises a concern about possible timing attacks, because it is almost guaranteed that one or the other of these two operations will have variable timing on 8-bit CPUs.

5.2.12 Rijndael

Rijndael was clearly designed with smart cards in mind [DR98b]. It fits in a very small ROM footprint (under 1K bytes) and has one of the fastest smart card speeds of any of the candidates. On-the-fly subkey generation allows a minimal RAM footprint of only 36 bytes. If enough RAM was available to hold 160 bytes of precomputed subkeys, the speed of Rijndael would be even faster. This 36-byte RAM footprint assumes that the key itself is stored in EPROM; otherwise, an additional 16 bytes of RAM is required. Alternately, the subkey computation must be "unwound" after each block is encrypted, lowering throughput considerably.

Perhaps the only negative comment about Rijndael on smart cards is that the performance numbers in the paper do not include a decryption function. Including decryption would increase the code size somewhat (probably between 512 and 1024 bytes), and decryption speed would be slower (possibly twice as slow) than encryption speed due to the coefficients of the inverse MDS matrix. As we pointed out before, there are many applications that require both encryption and decryption in the smart card, but even in decryption mode Rijndael is very competitive.

5.2.13 SAFER+

Given the 8-bit nature of SAFER+, it is not surprising that the algorithm fits well and performs well on smart cards. The code and table size should be easily less than 2K bytes of ROM. The RAM usage should be quite reasonable as well because of the simple on-the-fly key schedule. Unlike most other candidates, the round keys can be generated in any order, making the key-schedule aspects of the encryption and decryption very similar.

5.2.14 Serpent

Serpent was also clearly designed with smart cards in mind [ABK98b]. It can have a very small ROM footprint (under 1K bytes) or a somewhat larger bitslice version (2K) for higher performance. Serpent's on-the-fly key schedule allows for fairly minimal RAM usage. The decryption routine has to compute a simple linear pre-key recursion forwards to the end, adding a very small overhead to the first decryption, and then step backwards through the key schedule. Similar to many on-the-fly key schedules, Serpent usually requires an extra copy of 256 bits of key material in RAM (in addition to the key itself); alternately, this RAM could be saved by undoing the linear recursion after processing each block at the cost of a minor performance degradation.

5.2.15 Twofish

Twofish was designed with smart cards in mind. Its subkeys can be computed on the fly, and can encrypt and decrypt in 60 bytes of RAM (this includes space for the key, text, and working registers). Encryption and decryption are the same speed, and both efficient, in this implementation. If more RAM is available, Twofish can encrypt and decrypt at faster speeds. Several levels of space/speed are available; see [WS98, SKW+98b, SKW+99] for details.

6 Hardware Performance

For ultimate performance, only hardware implementations will do, so NIST has correctly required that AES be efficient in silicon. While most of the algorithms have several possible tradeoffs of silicon area versus performance, the most interesting figures here are the highest possible throughput and the silicon area required to achieve it. This metric has some associated ambiguity, because, for example, different interleaving modes may achieve dramatically different speeds. Nonetheless, our analysis here gives emphasis to the fully parallel hardware versions, since, if the cost of such an implementation is reasonably low, none of the other tradeoffs needs to be considered.

Critical is the requirement for fast context switching. Many applications, such as IPSec, require a hardware encryptor to switch contexts efficiently. Algorithms requiring subkey precomputation have a lower key agility due to the precomputation time, and they also require extra RAM to hold the precomputed subkeys, as compared to algorithms with true on-the-fly subkey generation. The amount of RAM required to hold a key context is shown below in Table 9. For many applications, the time required to switch in even 128 bytes is prohibitive.

6.1 Comparing Performance in Hardware

The speed of almost all the candidate algorithms in hardware seems to be acceptably fast, given certain assumptions. That having been said, it is fairly difficult to compare the results given in the AES submission papers. Most estimates involve different process technologies (e.g., 0.35 micron vs. 0.25 micron), different size metrics (e.g., gates vs. silicon area), different design methodologies (e.g., FPGA vs. gate array vs. custom layout), different assumptions about subkey precomputation (precomputed vs. on-the-fly generation), and different operating conditions (e.g., best case vs. worst case). Thus, our comments here are based on estimates, and we try to stay away from precise comparisons. Instead we focus on high-level issues that affect relative efficiency and speed.

6.2 Algorithm-Specific Comments on Hardware Performance

6.2.1 CAST-256

CAST-256 is large in hardware because the 4K bytes of ROM used for S-boxes is monolithic and cannot be reduced in size by serializing, as with most other algorithms. This means the algorithm has a much smaller range of speed–area tradeoffs. As in software, the algorithm is relatively slow because of the large number of rounds compared to most other candidates, implying more clocks per block.

In practice, the key agility is effectively quite low, despite the fact that the subkeys can be generated

Algorithm Name	Key Context RAM (bytes)
CAST-256	0
Crypton	0
DEAL	0
DFC	0
E2	256
Frog	2300+
HPC	?
Loki97	?
Magenta	?
Mars	160
RC6	176
Rijndael	0
SAFER+	0
Serpent	0
Twofish	0

Table 9: Hardware Key-Context RAM Requirements.

on the fly. This is because the key schedule uses the same large S-box multiple times, meaning that the subkey circuitry has to include another large ROM, or the ROM must be time-multiplexed with encryption, slowing the algorithm down even further.

6.2.2 Crypton

Crypton is probably the most hardware-friendly AES candidate. It has all the hardware benefits of Rijndael, without any of the major concerns. In particular, encryption and decryption use the identical circuitry, and there are no large ROMs for S-boxes. In fact, a fully instantiated version (i.e., all twelve rounds), with performance approaching 2 Gbit/s, can be realized in under 50,000 gates! It would be simple to build significantly smaller versions without affecting performance by iterating at a much higher clock rate; for example, a four-round instantiation should take under 15,000 gates and still provide over 1.5 Gbit/s.

6.2.3 DEAL

DEAL is just like triple-DES in speed and size. The size numbers are actually fairly reasonable, but it is difficult to achieve 1 Gbit/s using triple-DES in today's technology (without interleaving) because of the sheer number of rounds required.

6.2.4 Decorrelated Fast Cipher

The largest element in DFC is clearly the 64-bit modular multiplier. This element can easily be serialized to perform the operation in multiple clocks to trade off area for speed. The ROM size required is quite modest at sixty-four 32-bit words. Unfortunately, the DFC submission paper gives no hardware estimates. It appears that a fully parallel DFC should have a reasonable area: not too large, but not too small. Assuming a 20 nsec modular multiply (extrapolated from Alpha multiply times), with a 30 MHz clock rate for one round, a speed of about 500 Mbit/s is achievable.

Key agility is not very high, since subkey generation takes significantly longer than a block encryption. On-the-fly subkey generation is possible but even slower.

6.2.5 E2

E2 is large in hardware. A fully parallel version requires sixteen 256-byte ROMs. One particularly annoying feature of the design is the use of a multiplication in only the initial and final permutation; this requires the inclusion of a complete multiplier circuit that gets very little use. It is possible to serialize the circuit to various levels and trade off area for space, but a careful study of the algorithm leaves the overall impression that hardware considerations were not paramount during the design.

In addition, the key agility is quite low by design, and requiring 256 bytes of RAM to hold subkeys significantly increases the hardware size.

6.2.6 Frog

Frog's bomb permutation addressing is an operation that is not directly supported in hardware. Still, it is simple to create a logical circuit that is equivalent. The key schedule and S-box lookups, on the other hand, are problematic. The key schedule requires 2300 bytes of RAM, making it one of the most costly AES submissions from a hardware point of view. A straightforward implementation has to perform 128 S-box lookups in the RAM. It seems virtually impossible to do this in less than 128 RAM cycle times. If we assume a 10 nsec RAM access time (corresponding to a 100 MHz clock) we get a throughput of only 100 Mbit/s. The only way to increase the speed seems to be to use a faster RAM.

Overall, Frog is very costly to implement in hardware, and creating an implementation capable of anything more 100 Mbit/s seems a challenge. The key agility of Frog is also exceptionally low.

6.2.7 Hasty Pudding Cipher

We have not studied the different ways of implementing HPC in hardware in any detail. For our discussion, we only look at the 128-bit case. Each round consists of a long sequence of boolean operations with 4 table lookups into the KX table. The KX table alone requires 2 KB RAM. The boolean operations can be done very efficiently in hardware, so the table lookups seem to be the hardware bottleneck.

Filling the KX table from the key is a very expensive operation. It requires 4 table accesses for each of the 256 steps for each of the 3 passes for a total of over 3000 table accesses. A straightforward implementation requires at least 3000 clock cycles for a key setup. Thus the key agility of HPC is very low.

6.2.8 Loki97

Of the primitive operations in Loki97, the S-boxes are the most complex. Due to the particular choice of the S-boxes, they can probably be implemented as a direct boolean function instead of a table lookup. Once that has been done, the round function is very fast. The S-box lookups can either be done all in parallel (if there are enough copies of the S-box logic) or in serial fashion, providing a nice tradeoff between speed and area.

Loki's key schedule is about three times as expensive as the encryption itself. Furthermore, it doesn't support decryption without a large number of precomputations. This suggests that the key should be expanded in hardware, at the cost of 192 bytes of RAM. Key agility is reasonable, as a straightforward key setup requires 48 clock cycles.

6.2.9 Magenta

Magenta provides a lot of flexibility in the hardware implementation. The implementor can choose how many f-circuits to implement. Each round requires 384 evaluations of the f-function, and natural choices would be 1, 2, and 32 f-circuits. Even with 32 copies of the circuit, each round still requires 12 clock cycles. At six rounds for a full encryption, a "natural" implementation requires 72 clock cycles per block. Assuming a 100 MHz clock rate, this translates into a speed of about 180 Mbit/s. It seems hard to make implementations that are significantly faster than this.

Magenta's key agility is excellent: there is no key setup time required.

6.2.10 Mars

The kitchen-sink nature of Mars makes hardware implementation unwieldy. Note that an implementation requires a full multiplier, a full rotator, and a large (2K byte) S-box ROM. Each of these blocks is large and limits the ability to build a small hardware module with reasonable performance. The Mars paper [BCD+98] estimates a size of 70,000 "cells," which is certainly considerably larger than many of the other candidates.

Because there is no on-the-fly subkey generation, the key agility of Mars in hardware is fairly low. For example, precomputing all the subkeys apparently requires 280 clocks, while an entire block can be normally processed in about 50 clocks. Even if the precomputation for the "next" key is performed in parallel with an encryption, the time to change a key still dominates.

6.2.11 RC6

The simplicity of this algorithm lends itself nicely to hardware. There are many very useful speed–size tradeoffs possible, such as time-multiplexing only a single multiplier and rotator, using a bit-serial (or multi-bit) multiplier/rotator, etc.

The size and performance numbers in the submission [RRS+98] are clearly derived from a full custom layout, and are thus difficult to compare directly to the estimates from almost all the other candidates. It should be noted that every other candidate would also benefit dramatically from a full custom layout, but the vast majority of hardware implementations today are built with gate arrays or standard cells. Also, the speed numbers seem incredibly optimistic; performing a 32-by-32-bit multiply in 3 nsec and a 32-bit rotation in 1 nsec in a 0.25 micron process is not conservative.[5] The estimates in the RC6 paper might possibly be for nominal or best-case conditions (process, temperature, voltage), but certainly not the average-to-worst-case estimates required in any manufacturable design.

This is not to say that RC6 cannot be built in a reasonable size and at a reasonable performance, but the estimates seem highly optimistic. In other words, it is our belief that the size and speed of RC6 in hardware are good, but not as stellar as represented in the paper.

Because there is no on-the-fly subkey generation, the key agility of RC6 in hardware is fairly low. For example, precomputing all the subkeys requires 132 clocks, while an entire block can be normally processed in 20–40 clocks. Even if the precomputation for the next key is performed in parallel with an encryption, the time to change a key still dominates.

6.2.12 Rijndael

Rijndael fits and performs reasonably well in hardware. The use of only XOR operations (instead of additions) means that carry propagation times are avoided, although conventional carry look-ahead circuits make this issue much less important than the authors suggest [DR98a]. The simple on-the-fly key schedule for Rijndael means that key agility is very high, with a key change effectively taking zero time.

The major concern with Rijndael in hardware is that a fully parallel version requires sixteen 256-byte ROMs, for a total of 4K bytes of ROM. This is not terribly unreasonable for a very high-performance design, but a gate-level S-box (e.g., Crypton, Twofish) would have cut the hardware size significantly, although it might have affected the cryptanalysis. Fortunately, it is fairly easy to build lower performance versions of Rijndael with fewer S-box ROMs, accepting a slowdown inversely proportional to the number of ROMs.

It should also be noted that building decryption in hardware will probably double the chip size, since neither the S-box nor the MDS matrix of encryption can be used at all for decryption. This total asymmetry between encryption and decryption building blocks of Rijndael seems to be unique among the AES candidates, and its most annoying impact is on the hardware implementation. Also, the key agility for decryption is lower (about one block time), since the key schedule must be run forward for one entire block before decryption can begin.

6.2.13 SAFER+

SAFER fits reasonably well in hardware. A fully parallel version requires sixteen 256-byte ROMs and can run at acceptable speeds, but it is possible to serialize the S-box lookups to various levels to trade off speed for area. Decryption is not identical to encryption, but fortunately the same ROMs can be used, although the mixing transforms are quite distinct and thus increase area. The key agility is high, thanks to the simple key schedule.

6.2.14 Serpent

Serpent fits reasonably well in hardware and has a nice set of performance tradeoffs. A fully parallel version would require a total of 256 S-boxes, each a 4-bit permutation, for an equivalent of 2K bytes of "table," although the S-boxes can be built with direct logic. The round function should run quite fast, giving very nice performance at one clock per round. Note that decryption requires the inverse permutations and linear transformations, thus doubling this portion of the logic. Fortunately, there are many possible serialized versions of Serpent; for example, a bit-serial version performs one S-box lookup per clock, cutting the gate count (and the throughput) dramatically. The ability to perform almost a linear tradeoff of area versus speed is very attractive.

The key agility of Serpent in hardware is very good, allowing on-the-fly subkey computation, or a precomputation time equal to a single block encryption time.

6.2.15 Twofish

Twofish was designed from the beginning with hardware implementations in mind [SKW+98a]. There are many possible space–time tradeoffs that make the algorithm efficient both for high-speed and for low gate-count implementations.

The key agility is high, both for encryption and decryption, because the key schedule can be run on the fly in either direction. This feature is rare among the candidates. Hence, the additional RAM required

[5] To validate this impression, we spoke extensively with experienced commercial multiplier designers and took the Intel Pentium II 0.25 micron CPU multiplier (4 clocks at 400 MHz = 10 nsec) as a case in point.

for the expanded key is 0; however, hardware implementations can save approximately 10,000 gates with a key context of 50% of the key size.

7 Conclusions

The most obvious conclusion that can be drawn from this exercise is that it is very difficult to compare cipher designs for efficiency, and even more difficult to design ciphers that are efficient across all platforms and all uses. It's far easier to design a cipher to be efficient on one platform, and then let the other platforms come out as they may. Most of the AES candidates seem to have done this.

In the previous sections, we have tried to summarize the efficiencies of the AES candidates against a variety of metrics. The next thing to do is to assign a numerical score to each metric and each algorithm, then weights to each of the metrics, and finally to calculate an overall score for the different algorithms. While appearing objective, this would be more subjective than we want to be; we leave it as an exercise to the reader.

The performance comparisons will most likely leave NIST in a bit of a quandary. The easiest thing for them to do would be to decide that certain platforms are important and others are unimportant, and to choose an AES candidate that is efficient on only the important platforms. Unfortunately, AES will become a standard. This means AES will have to work in a variety of current and future applications, doing all sorts of different encryption tasks. Specifically:

- AES will have to be able to encrypt bulk data quickly on top-end 32-bit and 64-bit CPUs. The algorithm will be used to encrypt streaming video and audio to the desktop in real time.

- AES will have to be able to fit on small 8-bit CPUs in smart cards. To a first approximation, all DES implementations in the world are on small CPUs with very little RAM. They are in burglar alarms, electricity meters, pay-TV devices, and smart cards. Sure, some of these applications will get 32-bit CPUs as those get cheaper, but that just means that there will be another set of even smaller 8-bit applications. These CPUs will not go away; they will only become smaller and more pervasive.

- AES will have to be efficient on the smaller, weaker 32-bit CPUs. Smart cards won't be getting Pentium-class CPUs for a long time. The first 32-bit smart cards will have simple CPUs with a simple instruction set. 16-bit CPUs will be used in embedded systems that need more power than an 8-bit CPU, but can't afford a 32-bit CPU.

- AES will have to be efficient in hardware, in not very many gates. There are lots of encryption applications in dedicated hardware: contactless cards for fare payment, for example.

- AES will have to be key agile. There are many applications where small amounts of text are encrypted with each key, and the key changes frequently. IPsec is an excellent example of this kind of application. This is a very different optimization problem than encrypting a lot of data with a single key.

- AES will have to be able to be parallelized. Sometimes you have a lot of gates in hardware, and raw speed is all you care about.

- AES will have to work on DSPs. Sooner or later, your cell phone will have proper encryption built in. So will your digital camera and your digital video recorder.

- AES will need to work as a hash function. There are many applications where DES is used both for encryption and authentication; there just isn't enough room for a second cryptographic primitive. AES will have to serve these same two roles.

Choosing a single algorithm for all these applications is not easy, but that's what we have to do. It might make more sense to have a family of algorithms, each tuned to a particular application, but NIST's current plan is that there will be only one AES. And when AES becomes a standard, customers will want their encryption products to be "buzzword compliant." They'll demand it in hardware, in desktop computer software, on smart cards, in electronic-commerce terminals, and in other places we never thought it would be used. Anything chosen as AES has to work in all those applications.

8 Authors' Biases

As authors of the Twofish algorithm, we cannot claim to be unbiased commentators on the AES submissions. However, we have tried to be evenhanded and fair. If Twofish comes out looking like one of the best candidates, it is because we realized early on in the Twofish design process that our submission had to be efficient in a wide variety of applications and a wide variety of platforms, and took pains to make sure that is true. We were surprised to discover that most of the other submitters did not take the same efficiency requirements into account.

Many of the performance numbers in this paper are estimates; we simply did not have time to code each submission in optimized assembly language. As more accurate performance numbers appeared, we have updated the tables in this paper. We will continue to do so.

References

[ABK98a] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 98.

[ABK98b] R. Anderson, E. Biham, and L. Knudsen, "Serpent and Smart Cards," *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998, to appear.

[Ada98] C. Adams, "The CAST-256 Encryption Algorithm," NIST AES Proposal, Jun 98.

[Alm99] K. Almquist, "AES Candidate Performance on the Alpha 21164 Processor," version 2, posted to sci.crypt, 4 Jan 1999.

[Bih98] E. Biham, "Design Tradeoffs of the AES Candidates," invited talk presented at ASIACRYPT '98, Beijing, 1998.

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "MARS — A Candidate Cipher for AES," NIST AES Proposal, Jun 98.

[BP98] L. Brown and J. Pieprzyk, "Introducing the new LOKI97 Block Cipher," NIST AES Proposal, Jun 98.

[CMK+98] L. Chen, J.L. Massey, G.H. Khachatrian, and M.K. Kuregian, "Nomination of SAFER+ as Candidate Algorithm for the Advanced Encryption Standard (AES)," NIST AES Proposal, Jun 98.

[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.

[DR98a] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 98.

[DR98b] J. Daemen and V. Rijmen, "The Block Cipher Rijndael," *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998, to appear.

[GGH+98] H. Gilbert, M. Girault, P. Hoogvorst, F. Noilhan, T. Pornin, G. Poupard, J. Stern, and S. Vaudenay, "Decorrelated Fast Cipher: an AES Candidate," NIST AES Proposal, Jun 98.

[GLC98] D. Georgoudis, D. Lerous, and B.S. Chaves, "The 'Frog' Encryption Algorithm," NIST AES Proposal, Jun 98.

[Gla98] B. Gladman, "AES Algorithm Efficiency," `http://www.seven77.demon.co.uk/aes.htm`, 1 Dec 98.

[JH98] M.J. Jacobson and K. Huber, "The MAGENTA Block Cipher Algorithm," NIST AES Proposal, Jun 98.

[Knu98] L Knudsen, "DEAL — A 128-bit Block Cipher," NIST AES Proposal, Jun 98.

[Lim98] C.H. Lim, "CRYPTON: A New 128-bit Block Cipher," NIST AES Proposal, Jun 98.

[NTT98] Nippon Telephone and Telegraph, "Specification of E2 — a 128-bit Block Cipher," NIST AES Proposal, Jun 98.

[PRB98] P. Preneel, V. Rijmen, and A. Bosselaers, "Principles and Performance of Cryptographic Algorithms," *Dr. Dobb's Journal*, v. 23, n. 12, 1998, pp. 126–131.

[PV98] G. Poupard and S. Vaudenay, "Decorrelated Fast Cipher: an AES Candidate well suited for low cost smart card applications," *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998, to appear.

[RRS+98] R. Rivest, M. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 98.

[Sch98] R. Schroeppel "Hasty Pudding Cipher Specification," NIST AES Proposal, Jun 98.

[SKW+98a] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, Jun 98.

[SKW+98b] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish on Smart Cards," *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998, to appear.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, *The Twofish Encryption Algorithm: A 128-bit Block Cipher*, John Wiley & Sons, 1999.

[WS98] D. Whiting and B. Schneier, "Improved Twofish Implementations," Twofish Technical Report #3, Counterpane Systems, 2 Dec 98.