Implementation Experience with AES Candidate Algorithms

by Dr Brian Gladman, UK

Introduction

This paper presents experience gained during the implementation of each of the 15 AES candidate algorithms and seeks to provide fair and accurate comparisons in respect of implementation and performance issues.

This paper considers the following topics:

- the effectiveness of each of the specifications from an implementation perspective
- the feasibility of implementing the algorithms using these specifications alone
- the effort involved in implementing each algorithm to a reasonable level of efficiency
- The comparative performance of the AES candidate algorithms when coded in C for Pentium Pro and Pentium II processors.

The Algorithm Specifications

The Character of the Specifications

The specifications of the 15 AES candidates vary widely in form, with some using a formal mathematical style while others rely on a combination of text, diagrams and pseudo code. While each of these approaches can support correct implementation, they are significantly different in their ease of use from an implementation perspective. For example, although formality is often valuable in security critical code, it is surprising how difficult it is to avoid semantic ambiguities that can undermine precision and lead to implementation errors. On the other hand, it can also be extremely difficult to describe some features textually in an unambiguous way.

Given these factors the most helpful approaches are those that involve descriptions using more than one form. Although descriptive redundancy introduces the opportunity for inconsistency, more importantly it reduces the risk that errors will persist and provide a basis for erroneous implementation. Consequently, specifications that employ a mixture of text, diagrams and pseudo code will generally be preferable to those that rely on one form of description alone.

Provision of Guidance on Implementation

The AES algorithm specifications also vary widely is in their coverage of implementation options and optimisation opportunities. Some design teams have clearly taken the view that such guidance is not necessary whereas others have gone to considerable lengths to explain how their algorithm can be implemented efficiently within a range of processor environments.

In some specifications, the way in which an algorithm is described is quite different to the way in which it is most efficiently implemented. Moreover, there are AES specifications that omit important details of the mathematical constructions that they use. Whilst such omissions do not prevent implementation, they lead to significant extra work that could easily be avoided by providing the details concerned.

Byte Order

An area of general difficulty in a number of the specifications is in the conventions used for byte order within multiple byte values (this will be referred to here as ‘endianness’).

Several of the specifications contain errors caused by confusion about byte order whilst others switch between different byte order conventions in a way that seems certain to lead to confusion.

Byte order on input and output is a particular area of uncertainty. Quite a few AES candidates avoid the endian issue by defining their inputs and outputs as 32-bit (or 64-bit) quantities so that byte order and any associated conversion costs are external to the algorithm. Some algorithms don’t specify their endianness and hence force prospective implementers to discover this using test vectors. Still others do specify an endian convention but then proceed to use the opposite convention in some or all of their specifications.

In the authors experience this has been by far the most troublesome issue in implementing and testing the 15 AES candidate algorithms. In fact the development process is compounded because the standard test vectors for variable text and variable keys do not contain any ‘endian neutral’ vectors of the kind that are useful in resolving such ambiguities.

Although in an ideal world the specifications would be precise and unambiguous on their byte order conventions, experience suggests that this is unlikely to be achieved in practice. Consequently it is recommended that the standard sets of test vectors for variable text and variable keys should be augmented with (at least) an ‘all 0’ vector as an aid in resolving such difficulties.

It is not clear whether byte order on input and output is an internal or external issue from the viewpoint of

the AES algorithms. However, in the following commentary it will be assumed that the AES specifications are intended to provide a basis for implementations that produce results that are the same on processors with different byte order conventions.

Comments on Specific Specifications

CAST-256

With one exception, this specification fully describes the algorithm and hence allows implementation without reference to source code. The exception is byte order on input/output, which is big-endian but does not appear to be specified.

No implementation guidance is provided but the algorithm is largely conventional and this makes this omission a relatively minor one.

CRYPTON

The CRYPTON specifications are all well presented and provide the details needed for implementation from scratch. The algorithm defines and uses little endian byte order. Rounds are numbered from 1, not 0, and when this is combined with reference to 'even' and 'odd' rounds, there is a small amount of room for confusion.

There is limited implementation guidance. The 'version 1' algorithm is an improvement on earlier versions from an implementation viewpoint because the key-schedule is easier to understand.

DEAL

The DEAL specification is sound but relies heavily on the separate specification of DES. Input/output byte order is not specified but appears to be little endian. There is almost no implementation guidance.

DFC

The DFC specification is complete but originally contained errors that prevented correct implementation from scratch. The corrected version fully supports this.

Care is taken to specify big endian byte order. Although the main specification document gives very little guidance on efficient implementation, an ancillary document giving some help has since become available.

E2

The E2 specification uses an effective combination of formal text and diagrams to describe this algorithm. Nevertheless, byte order conventions are confusing since section 1.1 of the document sets out what seem to be 'little-endian' conventions when, in fact, the algorithm is 'big-endian'.

This situation arises because section 1.1 numbers entities from 'right to left' whereas the main specification uses 'left to right' numbering. This notational inconsistency is unfortunate and seems certain to cause confusion.

A supporting document provides very helpful implementation guidance.

FROG

A combination of text, diagrams and pseudo code is used to describe FROG. This fully supports implementation and the provision of extensive pseudo code makes implementation guidance largely unnecessary.

However, the pseudo code is confusing in parts because it specifies redundant code (in the makePermutation procedure the line 'if index > last then index <= 0').

Byte order conventions are given.

HPC–Medium

HPC is an algorithm that involves many constituent sub-algorithms, only one of which is needed to meet the AES requirement. The comments here only cover HPC-Medium, the AES compliant component.

The HPC specification relies heavily on actual C code sequences to describe its operation and this makes its implementation relatively easy.

The input to HPC is defined in terms of 64 bit words and care is taken to define character order within these as 'little-endian'. However input and output byte order seems to be big-endian in practice (byte order changes were needed to match the test vectors on a little-endian processor).

LOKI97

The LOKI specification supports implementation except for input and output byte order. Internal byte order appears to be little endian but input and output seem to use big-endian conventions. No implementation guidance is provided.

MAGENTA

The specification is accurate and complete but is very compact and quite difficult to follow. There is no implementation guidance. Byte ordering is implied to be little endian.

MARS

The MARS specification is excellent form an implementation viewpoint since it uses text, diagrams

and pseudo code to give a very clear overall description of the algorithm. The input/output byte order convention used is little endian and clearly specified as such.

With the exception of an ambiguity in the 'key fixing' step (now corrected) it was possible to fully implement MARS from its specification.

The extensive pseudo code provided for MARS makes implementation relatively easy. Although this reduces the need for implementation guidance, some aspects of the key-schedule are not easy to implement efficiently and hence deserve coverage in this respect.

RC6

The RC6 specification is excellent. It defines and uses little endian conventions and provides full pseudo code that makes it quite difficult to make mistakes in its implementation. The simplicity of RC6 makes implementation guidance unnecessary.

Rijndael

The Rijndael specification is generally good but there are a number of discrepancies that make it impossible to implement the algorithm without reference to the supplied source code.

Byte ordering conventions are described but parts of the specification appear to use different conventions.

Good implementation guidance is provided.

SAFER+

This specification is complete and fully supports implementation without reference to source code. It uses big-endian byte ordering conventions on input and output.

The SAFER+ specification does not provide any implementation guidance. Surprisingly the PHT that forms the core of SAFER+ is only specified in matrix form without the decomposition that is needed for its efficient coding.

Serpent

This specification provides an accurate and precise description of the algorithm that is sufficient to allow implementation in its 'non-bitslice' mode except for input/output byte order. Internally Serpent is specified as little endian but its byte order on input and output is big-endian (but not clearly specified as such).

There is some implementation guidance provided but this would not be sufficient for implementers who were not already familiar with the concepts of 'bitslice' operation. In practice it is not possible to implement Serpent efficiently without reference to supplied source code since the specification does not provide any details of how the S boxes can be implemented as Boolean functions. However, since this algorithm is of non-US origin the header files containing these definitions are freely available.

Twofish

The Twofish specification is very comprehensive and contains all the information needed to implement the algorithm. Its byte order conventions are clearly defined.

High level guidance is provided on the ways in which Twofish can be implemented efficiently but parts of the algorithm – for example, the key-schedule – are described in a way that is likely to encourage inefficient implementation approaches. Although the information needed for efficient coding is available elsewhere in the document, it is not easy to find and is hence not 'user friendly' from an implementation perspective.

Conclusions in Respect of Specifications

In general the specifications of 15 AES candidate algorithms are provided to a good standard. Byte order remains as a significant problem that is illustrated by the following table. This shows the byte order changes have to be implemented by the author's source code to match the supplied variable text and variable key test vectors when running on a Pentium Pro/II processor.

Action	**Algorithms**
no action	CRYPTON DEAL FROG MAGENTA MARS RC6 Rijndael Twofish
invert byte order in 32 bit words	CAST-256 DFC E2 LOKI97
invert byte order in 64 bit words	HPC
invert byte order in 128 bit words	SAFER+ Serpent

The mapping of test vectors to algorithm input, output and key blocks used to compile this table is as follows. The vectors are read as hexadecimal numbers with consecutive pairs of hexadecimal digits representing single bytes. The left and right digits of each pair give the most and least significant four bits of each byte respectively. The sequence of ***digit pairs*** within each test vector is scanned from left to right and the resulting bytes are placed in consecutive

memory locations with increasing addresses. This matches the NIST convention on 'big-endian' processors but should require an inversion of byte order within input and output blocks on 'little-endian' processors.

In practice, the table shows that the byte order actually used varies widely among the 15 AES candidates. Whether this matters depends on AES policy: should byte order be specified by the encryption algorithm or is this an external issue?

However, any need to change byte order on input and output will involve processing costs and these can have a significant impact on algorithm performance. This is especially significant when an algorithm is fast and it is hence not surprising to find that all the higher speed AES candidates implement a byte order that avoids such overheads when running on the reference architecture.

Since these issues have a major impact on the portability of encrypted data between different processors, they will need to be resolved if this is an AES algorithm requirement.

Implementation Experience

Although the AES teams have provided reference and optimised implementations of their algorithms, it is evident that quite different approaches have been adopted in these respects. Thus, while some have invested substantial effort to demonstrate algorithm performance, others have left such efforts to be pursued by the wider community.

In consequence, comparison of the performance of the supplied implementations is more a comparison of the approach of the different design teams than it is an indication of the implementation properties of the algorithms themselves.

The author's aim has been to implement the algorithms in a more consistent way in order to provide a more equitable basis for their assessment. Accordingly, all 15 AES candidate algorithms have been implemented from scratch without reference to the code provided by the original design teams[1].

Choice of Implementation Approach

The work described here compares AES algorithm implementations and performance when written in C for the Pentium II machine. The choice of the Pentium II is simply the result of its availability but C was consciously chosen instead of the alternative of using an assembler for several reasons.

Firstly, writing good assembler code for modern processor architectures is far from easy and implementing all 15 AES candidates from scratch in assembler in the limited time available would almost certainly have been impossible.

Secondly, with modern C compilers it will normally be possible to achieve speeds that are within 30% of those achievable with hand coded assembler and this is close enough for the assessments that are needed at this stage in the AES process. At present, knowledge of the ultimate performance of the AES candidates on specific current generation processors is less important than understanding how well the algorithms map onto a wide range of different processor architectures. Developing and making C source code widely available was hence considered to be the most effective way of providing the sort of information that is most needed at this stage in the AES selection process.

Before comparing the relative performance of the AES candidate algorithms, the following paragraphs provide comments, where appropriate, on aspects of their implementation.

CAST-256

CAST-256 is a fairly conventional algorithm that is straightforward to implement. The cost of implementation is low and there appears to be limited opportunities for optimisation.

CRYPTON

CRYPTON is a novel algorithm that allows the same routine to be used for both encryption and decryption. It is quite intricate and hence takes some time to implement well. Optimisation opportunities are explained in the specification and are straightforward to implement. The key-schedule is much faster for encryption than for decryption.

DEAL

DEAL is easy to implement provided that DES source code is already available. There is limited room for optimisation in DEAL and the efficiency achieved is largely determined by that of the DES implementation on which it depends.

DFC

DFC is quite time consuming to implement efficiently on a 32-bit machine because it involves 64-bit arithmetic. There is considerable scope for optimisation, especially in the modular division step.

Since DFC is based on 64-bit arithmetic, it makes more sense to judge its performance using processor and compiler combinations that support such

[1] For some algorithms limited inspection of the provided source code was needed because of specification errors.

capabilities (which will be the norm in AES time scales).

E2

E2 is quite intricate and hence proved relatively costly to implement and optimise. However, at the time E2 was coded the absence of test vectors and uncertainty about byte order had a big impact on implementation cost.

There is considerable scope for optimisation in E2 and the author's experience suggests that the best approach is likely to vary from one processor family to another.

The assistance given by Kazumaro Aoki of NTT during implementation is gratefully acknowledged.

FROG

FROG is easy to implement since pseudo code is provided for its constituent parts. However, its key-schedule is painfully slow and offers little room for any obvious improvements in efficiency. For this reason alone FROG is not a realistic AES candidate.

HPC-128

The full HPC algorithm involves five sub-ciphers and this makes implementation from scratch very costly. It is hard to believe that this is necessary and the author has chosen to implement only HPC-128, the AES compliant element of the specification.

HPC-128 is relatively easy to implement since C source code fragments are provided in the specification. As with DFC, HPC uses 64-bit arithmetic, which means that its performance is relatively poor on 32-bit processors.

The key-schedule appears very costly compared to the encryption and decryption routines and this seems likely to count against it as a strong AES candidate.

Loki97

Loki97 is quite intricate and uses indices that have to be computed by masking out parts of words that are either 11 or 13 bits long. It was not particularly difficult to implement but it proved quite time consuming.

MAGENTA

The MAGENTA specification is very compact and is not designed to ease the implementation task. However, it proved relatively easy to implement although the resulting performance is very disappointing. Moreover, it seems unlikely that there are any optimisations that would provide the very significant gains needed to make it worth considering as a continuing AES candidate.

MARS

The extensive use of pseudo code to describe MARS makes implementation easy. It can also be optimised in a relatively straightforward way.

The only area that caused any difficulty with MARS was the 'key fixing' process in the key-schedule, where the behaviour of bit 31 in 32-bit words proved difficult to describe without reference to code fragments.

It seems likely that this aspect of the specification can be simplified without compromising security and the author feels that this would be worthwhile.

RC6

RC6 is by far the easiest of the AES candidates to implement. It takes very little time and the simplicity of the algorithm makes it quite difficult to make mistakes in its implementation.

It also performs well on the Pentium II and is easily the fastest of the candidates on this processor. It also optimises well in C where performance is within 10% of that achievable with hand coded assembly language.

Rijndael

Rijndael is a variant of square with a neat structure that allows very good optimisation on 32 bit processors. Its performance is very good and seems likely to remain so on many processors since it uses only efficient and commonly available instructions. Its key-schedule is asymmetric and is much faster for encryption than for decryption.

SAFER+

SAFER+ is a byte-oriented algorithm that does not take full advantage of the 32-bit operations available on the Pentium II. In consequence, its performance is unspectacular on this processor. However little time was spent on optimisation so there is likely to be room for significant improvement (this has been confirmed by a recent Cylink announcement on the NIST AES forum).

Serpent

Serpent is an innovative algorithm that exploits the 'bit-slice' approach to algorithm implementation. However, its performance is relatively poor compared to many AES candidates, in part because it employs an unusually large number of rounds.

The bit-slice version of the algorithm depends on finding Boolean functions to represent S boxes that can be computed in a minimum number of processor cycles. Such optimisations were undertaken as a part of the implementation process.

Twofish

Twofish is a quite complex algorithm that combines many different techniques. It is quite expensive to implement from scratch, especially so if optimum performance is needed.

The resulting benefit is that the algorithm can be implemented in many different ways that allow it to be optimised for a wide range of applications scenarios.

Comparative Performance

The performance of the 15 AES algorithms has been compared by timing encryption, decryption and key-schedule computation on the Pentium Pro reference platform. The results are presented in Table 1.

Timing

The timing was undertaken using the Pentium time stamp counter in a code sequence of the following general form:

```
cpuid
rdtsc
save counter - value 1
cpuid
timed subroutine call ) – one call
cpuid
rdtsc
save counter - value 2
cpuid
timed subroutine call ) – two
timed subroutine call ) - calls
cpuid
rdtsc
save counter - value 3
```

```
cpuid
```

where the “rdtsc” instruction reads the time stamp counter and the “cpuid” instruction forces the processor to complete all previous instructions before it continues. This is needed to avoid erroneous timings resulting from ‘out-of-order’ execution of the cycle count reading instructions.

The minimum values of:

```
Time for 2 = value 3 – value 2
time for 1 = value 2 – value 1
```

were then determined over 100 runs of the above sequence and the difference between these values was then reported as the number of cycles required for the subroutine in question. Before each timing sequence, the routine being timed was run at least once in order to remove cache-filling effects.

Byte Order

It has been noted earlier that the AES algorithms use different conventions for byte order, with some candidates needing byte order changes on input and output in order to match the test vectors provided.

It is not surprising that all the fastest algorithms avoid byte order changes by using appropriate ordering conventions for the ‘little-endian’ reference platform. However, if these algorithms were run on big-endian machines, they would require byte order changes and their performance would suffer accordingly.

The faster an algorithm is the more impact this will have. For example on the 200MHz Pentium Pro

	RC6	Rijndael	MARS	Twofish	CRYPTON	CRYPTON v1	CAST	E2
Key Setup –128	1632	305:1389	4316	8414	531:1369	744:1270	4333	9473
-192	1885	277:1595	4377	11628	539:1381	748:1284	4342	9540
-256	1877	374:1960	4340	15457	552:1392	784:1323	4325	9913
Encrypt –128	270	374	369	376	474	476	633	687
-192	267	439	373	376	473	469	633	696
-256	270	502	369	381	469	470	639	691
Decrypt –128	226	352	376	374	474	470	634	691
-192	235	425	379	374	470	470	633	693
-256	227	500	376	374	483	469	638	706
Encrypt -128	94.8	68.4	69.4	68.1	54.1	53.8	40.4	37.3
Decrypt -128	113.3	72.7	68.1	68.4	54.1	54.5	40.4	37.0
Average -128	103.2	70.2	68.7	68.3	54.1	54.1	40.4	37.2

	Serpent	DFC	HPC	SAFER+	LOKI97	FROG	DEAL	MAGENTA
Key Setup –128	2402	5222	120749	4278	7430	1416182	8635	30
-192	2449	5203	120754	7426	7303	1422837	8653	25
-256	2349	5177	120731	11313	7166	1423613	11698	37
Encrypt –128	952	1203	1429	1722	2134	2417	2339	6539
-192	952	1288	1477	2555	2138	2433	2358	6531
-256	952	1178	1462	3391	2131	2440	3115	8711
Decrypt –128	914	1244	1599	1709	2192	2227	2365	6534
-192	914	1235	1599	2530	2189	2255	2363	6528
-256	914	1226	1526	3338	2184	2240	3102	8705
Encrypt -128	26.9	21.3	17.9	14.9	12.0	10.6	10.9	3.9
Decrypt -128	28.0	20.6	16.0	15.0	11.7	11.5	10.8	3.9
Average -128	27.4	20.9	16.9	14.9	11.8	11.0	10.9	3.9

The values are in clock cycles for Pentium Pro/II. The two key set-up values for Rijndael and CRYPTON are those for encryption and decryption respectively. The speeds in the last three rows are megabits/second for the 200MHz Pentium Pro reference platform.

Table 1

reference platform, an algorithm that achieves 25 megabits/second will suffer a penalty of around 2 megabits/second whereas one that is capable of 100 megabits/second suffers a much larger penalty of about 15 megabits/second.

In order to fairly compare algorithm performance the figures given in Table 1 therefore exclude any processing costs for changing byte order on input and output. The figures are thus a measure of the 'core' performance of the algorithms – the speed they can achieve on processors where input and output byte order changes are not needed.

Implementation Focus

The implementations on which Table 1 is based place emphasis on encryption/decryption speed rather than on limiting memory use or key-schedule cost.

Results

Table 1 shows that RC6 is the fastest algorithm on the Pentium Pro/II processor. Rijndael, MARS and Twofish follow and achieve effectively the same performance. Somewhat surprisingly, the speed of candidates varies over a large range (25:1).

For reference purposes DES coded in C can achieve a speed of over 27 megabits/second on the reference platform, considerably faster than many of the AES candidates.

Practical Encryption Speeds

For many cryptographic applications, encryption speed is more important than that for decryption. This applies, for example, when an algorithm is used in cipher block chaining mode or when it is used as a hash function. In addition, when a small number of blocks are encrypted the cost of computing the key-schedule will have a significant impact on algorithm performance.

Using the above figures (for 128 bit keys) the AES algorithms can be ranked in respect of the overall performance they achieve in encrypting blocks of different length. The resulting rankings for small, medium and large numbers of blocks are shown in the following table:

16 bytes	4096 bytes	$>10^6$ bytes
Rijndael	RC6	RC6
CRYPTON	Rijndael	MARS
RC6	MARS	Rijndael
Serpent	Twofish	Twofish
MARS	CRYPTON	CRYPTON
CAST	CAST	CAST

This shows that both Rijndael and CRYPTON are very effective for small numbers of blocks because they both have fast encryption key-schedules.

For a text length of about 4000 bytes the better encryption speed of RC6 puts it in first place and other algorithms such as MARS and Twofish with good encryption speeds also improve their rankings.

For bulk encryption, RC6 is ahead of the other algorithms, followed by MARS, Rijndael and Twofish, all of which provide very similar levels of performance.

Note, however, that this table is based on a version of Twofish that is optimised for bulk encryption. Its performance for small numbers of blocks could be considerably improved by using a different version (although bulk encryption speed would then suffer unless both versions were available).

Serpent enters the table for the encryption of one block but its lower encryption speed quickly reduces its ranking as the number of blocks increases.

Conclusions

The AES winning candidate will need to perform well in a wide range of different environments – on high-end and low-end processors, on smart cards and in hardware. On this basis, it would be wrong to eliminate candidates solely on the performance that they provide on the reference platform as reported here. Quite apart from this, the primary concern must be security and until the list of secure candidates is known, it is premature to discuss the elimination of candidates with any certainty.

However, the algorithms vary across a very large range in performance terms on the Pentium Pro/II processor and this does allow some general conclusions to be reached.

First, it seems unlikely that candidates that provide less than 15 megabits/second on the reference platform should be carried forward into the next AES round. On this basis MAGENTA, DEAL, FROG and LOKI97 could reasonably be eliminated. This criterion would also make SAFER+ and HPC marginal although care is required in considering the latter since it will perform a great deal better on 64-bit processors (as will DFC). Moreover, a recent announcement by Cylink suggests that SAFER+ can achieve a much better performance than the author's code offers and this suggests that it would be wrong to rule this candidate out on the basis of the results given here.

A major reason for the lower performance of Serpent is its unusually large number of rounds. It seems certain that Serpent is very conservative when compared with other AES candidates and this suggests that its number of rounds could be significantly reduced to improve its performance. It

might hence remain as a candidate with such a change.

RC6 has to be a strong candidate for the next round if it is secure. It is simple, elegant, easy to implement and easy for C compilers to optimise. Moreover, its simplicity is likely to make implementation assurance much easier than for other candidate algorithms. Its use of a multiply instruction may hurt it on some processors but, despite this, its many attractive features combine to make it a strong contender.

Rijndael is an especially strong candidate because it is simple to implement and provides a very good performance across small, medium and large numbers of encrypted blocks. It maps well in C and seems likely to maintain its performance on a wide range of processor architectures.

MARS also achieves a very good level of performance on the reference platform, although its use of multiply instructions may again reduce its performance on some processors.

Twofish is an 'engineers cipher' in that it can be put together in a variety of different ways to achieve a good balance between performance and resource costs in many different operating contexts. The downside is that it is a relatively complex algorithm and this may make implementation assurance more difficult than for other candidates. Nevertheless, it would be surprising if it did not continue into the next round.

Of the candidates with lower 32-bit performance, HPC and DFC need careful consideration because they use 64-bit arithmetic and this will be highly efficient by the time an AES winner is chosen. However, HPC's key schedule is time consuming.

CRYPTON, E2 and CAST provide good 'mid-range' performance on the reference platform and their status is hence likely to be determined how well they perform in other contexts.

Additional Information

Further information on the work reported here is available on the author's web site at:

http://www.seven77.demon.co.uk/aes.htm

Acknowledgements

Many people have made helpful comments on aspects of the work reported here. The helpful contributions made by the following people are gratefully acknowledged:

- Russell Bradford
- Shai Halevi (IBM)
- Niels Ferguson and Doug Whiting (Twofish)
- Marcus Watts
- Eli Biham and Ross Anderson (Serpent),
- Richard Schroeppel (HPC)
- Richard Outerbridge (DEAL)
- Kazumaro Aoki (NTT)
- Sam Simpson
- Helger Lipmaa
- Louis Granboulan
- Vincent Rijmen (Rijndael)
- David Hearn

I would also like to thank the Intel Corporation for providing support for this work by providing a copy of their VTune™ performance tuning application that proved invaluable in the optimisation of the author's implementations of the AES candidate algorithms.