Some Comments on the First Round AES Evaluation of RC6

Scott Contini[1], Ronald L. Rivest[2], M.J.B. Robshaw[1], and Yiqun Lisa Yin[1]

[1] RSA Laboratories, 2955 Campus Drive, San Mateo, CA 94403, USA
{scontini,matt,yiqun}@rsa.com
[2] M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA
rivest@theory.lcs.mit.edu

1 Introduction

The first round of the AES process is coming to an end. Since August of 1998, the cryptographic community has had the opportunity to consider each of the fifteen proposed AES candidates. In this note, we take the opportunity to answer some of the questions and to respond to some of the issues that have been raised about the suitability of RC6 as an AES candidate.

2 Encryption/Decryption Performance of RC6

Since the publication of RC6 a variety of researchers and implementors have examined RC6 and considered its performance in a wide range of environments.

2.1 32-bit architectures

RC6 is one of the fastest AES proposals on 32-bit architectures, particularly so on the NIST reference platform of a 200 MHz Pentium Pro. It is argued by many that the most reasonable way to assess the performance of an algorithm is to consider its speed in an assembly language implementation. We agree that this is the case. However it is also interesting to consider how the performance of RC6 compares when using different compilers. It is important to note that to take advantage of compiler-specific optimizations the source code provided to the compiler might need to be changed. When we do this a wide range of performance speeds on the NIST reference platform become possible as is illustrated by the following table.

Speed in Mbits/sec on 200 MHz Pentium Pro		
Compiler	*Encrypt*	*Decrypt*
Borland (writing rotate as (i))	41.5	45.2
MSVC (writing rotate as (i))	53.4	57.0
MSVC (using `rotl` and `rotr`)	97.8	82.3
GCC (writing rotate as (i))	61.2	41.5
GCC (writing rotate as (ii))	69.2	65.3

Examples of suitable code are given for left rotate where | is used to denote the bitwise inclusive-or. Right rotate requires a similar form.

(i): `((a) >> (32-(b))) | ((a) << (b))`.
(ii): `(((a) << (int)(b)) | ((a) >> (32 - (int)(b)))`.

The generally superior performance of RC6 on the NIST reference platform, the Pentium Pro, is a very positive attribute of the algorithm. On other 32-bit architectures however, such as the Pentium, it seems that the performance advantage of RC6 is somewhat reduced [25]. Despite this RC6 still seems to be consistently ranked among the better-performing algorithms.

2.2 8-bit architectures and smart cards

During the performance assessment in the first round, several researchers implemented AES candidates in 8-bit environments. At this point it becomes exceedingly difficult to compare the different algorithms since few algorithms have been implemented in the same environment in a consistent manner.

We have analyzed the number of cycles required to encrypt with RC6 using an Intel MCS51 and our estimate is that encryption would require around $12,700$ cycles. This is a little faster than some other estimates given, such as those provided by Hacez et al. [14] which quotes $14,500$ cycles for encryption.

The key scheduling with RC6 has been the subject of some discussion. This revolves around the two issues of performance and RAM requirements.

The key-scheduling process is quite time-consuming requiring between $27,000$ cycles (RSA Laboratories estimate) and $43,000$ cycles [14]. This is equivalent to the cost of encrypting three to four blocks of data. In addition, the key schedule provided in the algorithm description of RC6 [24] does not allow the subkeys to be computed on the fly. At first sight it appears that the entire array of subkeys has to be generated before encryption and/or decryption can start requiring at least 176 bytes of RAM. At the Second AES Conference, however, Keating [16] demonstrated how the RC6 key scheduling could be accomplished with less RAM though in fairness, the performance hit when doing this would likely be prohibitive.

It is really unclear at the moment how important the performance of the AES algorithm is going to be on such severely memory-restricted 8-bit processors. We are unconvinced that performance on 8-bit processors is likely to be a pressing issue in the near future, much less so when we look even ten years on. Given the trend towards system-on-chip solutions and hybrid architectures using coprocessors, perhaps a more useful measure of the performance of RC6 in a constrained environment is given by Hachez et al. [14] in their implementation of four different AES candidates on the ARM processor. In Section 6.2 of [14] it can be seen that RC6 performs very well in this environment. With regards to the amount of RAM available, many predictions suggest that even on the cheapest smartcards there will routinely be more than enough for our purposes.

It is interesting to note that the key schedule of RC6 is a separate component from the encryption routine and is, in principle, replaceable. We show in the

Appendix a possible alternative way of providing the key scheduling for RC6 that might be of some independent interest.

2.3 Future architectures

An important consideration for the AES, given its intended lifetime of 20 to 30 years, is the suitability of the chosen algorithm on future architectures. While it is difficult to make predictions on how different algorithms will perform, there has been some initial assessment. In this section we will consider the performance of RC6 in 64-bit environments and also the impact of future levels of parallelism.

Chief among current estimates is the work by Baudron et al. [1] where RC6 is timed on existing 64-bit platforms such as the DEC Alpha. Since the design of RC6 is oriented to 32-bit platforms there are no substantial problems in implementing RC6 in such environments. However, it may be the case that any performance gains that other algorithms attain, most notably those of the DFC algorithm [12], are not mimicked by similar gains for RC6. Much of this is due to the instruction set supported on these 64-bit architectures. Despite this, we would expect that some improvements could be made by adopting some simple programming tricks. For instance, left-rotating a 32-bit word `x` by some value `y` can be accomplished as follows. First define a 64-bit quantity `z` as `z = (x << 32) | x` where | is used to denote the bitwise inclusive-or. Then the least 32 bits of `z >> (32 - y)` is equal to `x` left-rotated by `y` bit positions. This could be useful in certain sitations. We also note that if an operation to perform the 64-bit rotate of `a` by `b` bit positions were available, say `ROTL(a,b)`, then we could derive $x \lll y$ by taking the least 32 bits of `ROTL(z, y)` where `z` is also formed as above.

However, it is clear that such tricks can give only a partial improvement. Nevertheless without any additional attempts at optimization, the relative drop in performance of RC6 as measured and estimated by Baudron et al. is not as significant as it might first appear. The estimated encryption speed for DFC on the Alpha 21164a (the fastest of the AES submissions in this environment) is 304 cycles per block, whereas RC6 is estimated at 467 cycles per block. This is around 50% slower. Note however that this is a crude estimate that may or may not take account of assembly hand-optimization and appears to be using platforms that could themselves be just as obsolete in a few years as current 32-bit platforms are expected to be among the 64-bit enthusiasts. As a result it does not necessarily give a good indication of future performance.

Instead it would be better to look to future 64-bit designs to consider how the different algorithms will fare. As a first step, during the rump session at the Second AES Conference Doug Whiting presented some preliminary results on the implementation of four AES candidate algorithms on sample Merced and McKinley chips. These are 64-bit architectures, and might appear to offer some indication of the expected performance gains of different algorithms.

Algorithm	*Cycles on Pentium Pro*	*Factor change with Merced*	*Factor change with McKinley*
RC6	250	2.5	2.1
Rijndael	283	0.6	0.5
Serpent	900	0.8	0.8
Twofish	258	0.8	0.7

The fact that it appears that RC6 will require so many more cycles on a new-generation 64-bit Intel chip than on an existing 32-bit chip is surprising to say the least! Unfortunately the results obtained were obtained under conditions of non-disclosure and there is no way of looking at the code or examining the details of the implementation. We suspect that the figures given are for a preliminary version of the chip, perhaps using microcoded routines instead of single instructions. We would be very surprised if such a situation were to remain the case in future generations of full production chips. However this will clearly be an important issue in any future comparison between the AES candidates.

Finally we discuss the issue of parallelism and cite the work of Craig Clapp [6]. In this paper, the effect of increasing levels of parallelism is investigated for seven of the AES submissions, including RC6. There it is shown that for a limited increase in the amount of parallelism available in the processor, RC6 would provide suitably increased performance speed. With very high degrees of parallelism available, some other submissions have the potential for a greater improvement in performance than RC6 and accomplish an encryption rate comparable to, and even surpassing, that of RC6. However the advantage in performance over RC6 for even the fastest algorithm when such extensive parallelism is available (Rijndael) is estimated to be by a factor of around 30%.

2.4 Java performance

At the second AES conference, NIST provided information on the performance of the different AES algorithms in Java [22]. There were several interesting aspects to this chart including the surprisingly poor reported performance of RC6.

This is completely at odds with our own measurements. NIST estimates the performance of RC6 using the JIT compiler to be around 1.4 Mbits/sec. However internal measurements at RSA Laboratories give a speed of around 25.2 Mbits/sec. Indeed, the timing given by NIST is roughly equal to the timing we obtain without the use of the JIT compiler. Currently we have no explanation for why this might be the case. Our early thoughts on a known bug in JDK 1.1.6 with JIT (see bugParade ID 4171185 at `http://developer.java.sun.com`) which causes some, but not all, programs to drop out of JIT compilation might, or might not, be relevant.

However we can point to several third party timings of RC6 and other AES algorithms in Java. These are provided by Alan Folmsbee of Sun Microsystems [11], NTT Laboratories [21], and Baudron et al. [1] and all three sources show that from among the AES submissions, RC6 offers the fastest encryption and

decryption performance in Java. When we also consider the memory requirements for the different algorithms as measured by NIST [22] then the suitability of RC6 for Java applications becomes clear.

3 Minimal-Rounds Performance

Biham [2] made a very interesting presentation at the Second AES Conference and descibed a fairer way to compare the performance of algorithms. There Biham made the case that performance should only be measured when the algorithms are considered to be offering similar resistance to cryptanalytic attack.

This is certainly the best way to compare algorithms, but unfortunately, establishing the appropriate number of rounds for each algorithm to give comparable security is rather subjective. There hasn't been sufficient time to assess the security of the different ciphers to make such a comparison any more meaningful than trusting the designers intuition and measuring the performance of the ciphers on a standard platform. We would agree with Adi Shamir's suggestion made during the subsequent discussion that such a comparison might be better suited to the second round assessment of algorithms.

Nevertheless it is interesting to see how the different algorithms compare using Biham's current estimates of the number of rounds required to give comparable security. In the table that follows, we do just this, using two sources of figures for the performance of the algorithm. One source [2] is that given by Biham for the Pentium with MMX and the second source [22] are the figures for an assembly implementation of the different algorithms provided by Schneier and quoted in [22].

Pentium Pro [22]			*Pentium (MMX) [2]*			*Pentium Pro [22]*		
proposed rounds			*minimal rounds*			*minimal rounds*		
algorithm	*rounds*	*cycles*	*algorithm*	*rounds*	*cycles*	*algorithm*	*rounds*	*cycles*
RC6	20	260	*Serpent*	17	956	*Mars*	20	244
Mars	32	390	*Mars*	20	1000	*RC6*	21	273
Twofish	16	400	*Rijndael*	8	1021	*Rijndael*	8	293
Rijndael	10	440	*Twofish*	14	1097	*Twofish*	14	350
Crypton	12	476	*Crypton*	11	1175	*Crypton*	11	436
CAST-256	48	660	*E2*	10	1507	*Serpent*	17	547
E2	12	720	*RC6*	21	1508	*CAST-256*	40	550
Serpent	32	1030	*CAST-256*	40	1740	*E2*	10	600

It seems that by making a preliminary adjustment for the minimal number of rounds, the performance difference between the top and the bottom algorithms from among the eight highlighted is reduced. (These eight were chosen as the top eight performers according to [2].) However when we look at the third column we see that the same algorithms appear in pretty much the same groupings as they did in the first column. For the most part the designers recommended number of rounds isn't too distant from the minimal number of rounds and it

doesn't appear that a minimal-round assessment really changes things dramatically. Certainly, however, it is clear that *Serpent* in particular has been designed with a considerable margin for safety.

4 Secure Implementation

At the Second AES Conference a number of papers discussed the relevance of timing attacks [17] and simple and differential power analysis [18] when applied to either the encryption routine or the key scheduling routine of the different AES submissions [3, 5, 10].

The first lesson to be learnt was that it is very difficult to implement any of the AES submissions so that they are absolutely resistant to such system attacks. Certainly though one might be able to take some precautions in the way different algorithms are implemented. Usually, however, such algorithmic fixes are at the expense of performance.

4.1 Timing attacks

The two operations in RC6 that seem to offer the most exposure to timing attacks are the data-dependent rotation and the multiplication. Several commentators [15, 17] have previously mentioned that RC5 could in theory be vulnerable to timing attacks if we make certain assumptions about how the data-dependent rotation unit is implemented in practice. The most typical assumption for such an attack to succeed is that the time required to rotate a w-bit word by t bit positions is directly proportional to t. Certainly, if this rather strong assumption holds then information about the subkeys used during encryption can be readily derived. Indeed, Handschuh [13] gives a thorough analysis of such a timing attack on RC5 under exactly this assumption.

Clearly, such considerations also extend to RC6 since it too depends heavily on data-dependent rotations. However, for most modern processors data-dependent rotation seems to be implemented as a constant time operation and as a consequence for both RC5 and RC6, any concerns about timing attacks involving the data-dependent rotation itself are immediately nullified.

Other processors, however, may have a rotation or shift time that depends linearly with the amount of rotation. This might apply particularly to processors found in smart cards and other computationally limited environments. In such situations, it is typically easy to arrange the work so that the total computation time is independent of the data, even though the rotate and/or shift time might not be. As an example, we compute a rotation by t bits using a left-shift of t bits and a right-shift of $w - t$ bits (for a wordsize of w bits). The sum total of the number of shifts is w bit positions irrespective of the value of t provided.

Whether or not the data-dependent rotation is constant time or not, the encryption and decryption time for both RC5 and RC6 can be considered to be data-independent, thereby causing any potential timing attacks that attempt to exploit the presence of the data-dependent rotation to fail.

The introduction of the integer multiplication might cause some observers to think about timing attacks on RC6 because of what might be termed an "early-out" strategy. To optimize performance, it is conceivable that a multiplication operation $A \times B$ might be implemented with a look ahead strategy. In particular, a processor might examine the bits of B (say) and if the high-order bits of B are not set, then the multiplication might be aborted early since the product has already been computed without the need to consider the role of the upper bits of B. In such situations the time required to perform a multiplication would be dependent on the data being multiplied and we believe that in such circumstances a timing attack on RC6 would be successful.

Our own tests on Pentium, Pentium II and Pentium Pro processors show that there is no such effect and this is consistent with the view that modern processors do not have an "early-out" strategy. With earlier bit serial multipliers it would be possible to skip over zeros in the multiplier (and, if done the right way, strings of ones as well). However, modern array multipliers take the same time regardless of the form of the inputs [19].

As with the operation of data-dependent rotations, however, we can also illustrate an implementation fix if this is thought to be an issue. One possible remedy is to use a blinding technique. Suppose we wish to compute x^2 for some x. Using an integer r picked at random we compute $s = x - r$. Then x^2 can be computed as $r^2 + 2rs + s^2$ the computation of which is independent of the form of x. This however is a moderately costly alternative.

A more efficient alternative is the following. Suppose we wish to compute x^2 for some x. Let $y = x \,|\, \texttt{0x80000000}$ where $|$ denotes the operation of bitwise inclusive-or. We thus have that y is equal to x with the top bit set. We then compute y^2 which will thwart any "early-out" strategy since the top bit is always set. We also have that $x^2 = y^2 \bmod 2^{32}$ which is clear if $x = y$ since immediately we have that $x^2 = y^2$. Note also that if $y = x + 2^{31}$ then $y^2 = x^2 + x2^{32} + 2^{62}$ which is congruent to x^2 modulo 2^{32}. The rest of the computation of $f(x) = 2x^2 + x$ can be completed in constant time.

4.2 Power analysis

Protection against power analysis is a much more difficult proposition. It seems that advanced analysis of nearly any primitive operation [10] can be used to deduce key-dependent information!

In this section we restrict ourselves to a consideration of Simple Power Analysis which might allow the constituent operations in the cipher to be identified, thereby potentially leaking key-dependent information. Protection against the more advanced Differential Power Analysis will be the subject of ongoing research.

For the data-dependent rotation there appears to be the following solution that provides resistance to both timing attacks and simple power analysis. This solution can be tweaked for different data block sizes of two bits, four bits or more, giving different trade-offs in terms of speed and memory requirements. The example given here is for an 8-bit implementation.

First we store two arrays

$$\texttt{shiftedbit[0][i]} = 0 \text{ for } 0 \le i \le 7 \text{ and}$$
$$\texttt{shiftedbit[1][i]} = 1 \ll \texttt{i} \text{ for } 0 \le i \le 7.$$

The following pseudocode shows how to provide a timing and simple power analysis resistant implementation of a left rotation of x by y bit positions. A right rotation can be implemented in a similar manner. Let $z = x \lll y$ and let $x[0], \ldots, x[3]$ be the constituent bytes of x with $x[0]$ being the least significant. Similar notation is adopted for the constituent bytes of z while y is a single byte value.

$\texttt{z}[0] = \texttt{z}[1] = \texttt{z}[2] = \texttt{z}[3] = 0$
for $i = 0$ **to** 31 **do**
    `bit =` $(x[i \gg 3] \gg (i \& 7))$;
    `byte = shiftedbit[bit][`$(i + y) \& 7$`]`;
    $z[((i + y) \gg 3) \& 3]$ `|= byte;`

While providing algorithmic protection against Simple Power Analysis might not be prohibitive, it's worth remembering that protection against the more advanced types of power analysis is particularly difficult for all of the AES submissions to achieve. This is certain to become an important area of future research.

5 Cryptanalysis of RC6

Over the months since RC6 was first presented to the public there has been substantial interest from the cryptographic community. We suggest that some of this interest has translated into an initial investigation into the security offered by RC6. Yet there have been few observations that cast doubt on the security offered. The most comprehensive and thorough investigation into the security of RC6 remains that given in [8].

Indeed most of the work that has taken place since the publication of RC6 has been on investigating the differences between RC5 and RC6 and some of the simplified variants of RC6 [4, 7, 9]. This body of work all seems to provide evidence that the changes made in moving from RC5 to RC6 were well-founded. In particular the use of integer multiplication to provide good diffusive properties and the fixed rotation have been highlighted several times as being particularly important to the security of the cipher.

The only potentially negative observations that we are aware of were presented in Baudron et al. [1]. They are the following. The first observation is that for the quadratic function $f(x) = 2x^2 + x$ there are many fixed points. Such fixed points arise when $x = 0 \bmod 2^{16}$ so that $f(x) = x$. The most interesting way we think that something like this might be useful to the cryptanalyst is in observing a similar effect when forming truncated differentials [20] across the quadratic function that hold with probability one. As an example, if we define

some exclusive-or difference Δ to have the form $\Delta = xy00$ where x and y are arbitrary byte values, then given some input pair a and b such that $b = a \oplus \Delta$ we have that $f(b) = f(a) \oplus \Delta'$ where $\Delta' = x'y'00$. The output difference has the same general form as the input difference though the values of the top two bytes in the output difference may well be different. Unfortunately for the attacker it appears to be difficult to use these in an attack because of the presence of the fixed rotation by five bit positions hinders the construction of good truncated differentials over sufficiently many rounds.

The second observation considers when each of the data-dependent rotations in every round of RC6 takes the value zero. In such a situation the exclusive-or of the least significant five bits of two of the plaintext input words and the least significant five bits of two of the ciphertext inputs is a constant that depends on the values of the subkeys. This might potentially allow for the recovery of some key material. However because of the probabilities involved this seems to be restricted to attacks on at most ten rounds of RC6, and indeed existing attacks using linear cryptanalysis [8] appear to offer better avenues for the attacker. Interestingly, even though the amount of data available to the analyst is constant since the block size is fixed, it could be that a theoretical attack could be mounted with work-load less than that expected for a particular key length. It could be that for 192- and 256-bit keys more than 20 rounds might be more appropriate to prevent such theoretical attacks. However this issue will become clearer as more detailed analysis of RC6 takes place.

6 RSA DSI's Position on Intellectual Property

The position of RSA Data Security on its policy with regards to any possible intellectual property coverage of the final AES algorithm is clear and unambiguous.

> RSA will not require licensing or royalty payments for the manufacture, use, or sale of products utilizing the algorithm selected as the AES, which conform with the AES, on the basis of any patents that RSA may hold that could be deemed to cover the selected algorithm. However, RSA may require appropriate notices acknowledging RSA's ownership of such patents.

This position is intended to promote our belief that the community is best served by having an advanced encryption standard that is chosen on purely technical grounds.

7 Conclusions

In view of the performance, security and exceptional simplicity of RC6, we believe that it should be considered for inclusion in the second round of AES evaluation.

Acknowledgements

We would like to thank David Young for his thoughts on the suitability of RC6 for 8-bit and 64-bit environments.

References

1. O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Poupard, J. Stern and S. Vaudenay. Report on the AES candidates. In Proceedings of *The Second AES Candidate Conference*, pages 53–67. March 22-23, 1999.
2. E. Biham. A note on comparing the AES candidates. In Proceedings of *The Second AES Candidate Conference*, pages 85–94. March 22-23, 1999.
3. E. Biham and A. Shamir. Power analysis of the key scheduling of the AES candidates. In Proceedings of *The Second AES Candidate Conference*, pages 115–121. March 22-23, 1999.
4. J. Borst, B. Preneel and J. Vandewalle. Linear cryptanalysis of RC5. In L. Knudsen, editor, *Fast Software Encryption*, *Lecture Notes in Computer Science*, to appear. Springer Verlag.
5. S. Chari, C. Jutla, J. Rao, and P. Rohatgi. A cautionary note regarding the evaluation of AES candidates on smart-cards. In Proceedings of *The Second AES Candidate Conference*, pages 133–150. March 22-23, 1999.
6. C. Clapp. Instruction-level parallelism in AES candidates. In Proceedings of *The Second AES Candidate Conference*, pages 68–84. March 22-23, 1999.
7. S. Contini and Y.L. Yin. On differential properties of data-dependent rotations and their use in MARS and RC6. In Proceedings of *The Second AES Candidate Conference*, pages 230–239. March 22-23, 1999.
8. S. Contini, R.L. Rivest, M.J.B. Robshaw and Y.L. Yin. The Security of the RC6 Block Cipher. v1.0, August 20, 1998. Available at `www.rsa.com/rsalabs/aes/`.
9. S. Contini, R.L. Rivest, M.J.B. Robshaw and Y.L. Yin. Improved analysis of some simplified variants of RC6. In L. Knudsen, editor, *Fast Software Encryption*, *Lecture Notes in Computer Science*, to appear. Springer Verlag.
10. J. Daemen and V. Rijmen. Resistance against implementation attacks: A comparative study of the AES proposals. In Proceedings of *The Second AES Candidate Conference*, pages 122–132. March 22-23, 1999.
11. A. Folmsbee. AES Java™ technology comparison. In Proceedings of *The Second AES Candidate Conference*, pages 35–52. March 22-23, 1999.
12. H. Gilbert, M. Girault, P. Hoogborst, F. Noilhan, T. Pornin, G. Poupard, J. Stern and S. Vaudenay. Decorrelated fast cipher: an AES candidate. Submitted to the Advanced Encryption Standard process. In *AES CD-1: Documentation*, National Institute of Standards and Technology (NIST), August 1998.
13. H. Handschuh and H. Heys. A timing attack on RC5. In pre-proceedings of *SAC'98 - Fifth Annual Workshop on Selected Areas in Cryptography*, pages 318-343, 1998.
14. G. Hachez, F. Koeune, and J.J. Quisquater. cAESar results: Implementation of four AES candidates on two smart cards. In Proceedings of *The Second AES Candidate Conference*, pages 95–108. March 22-23, 1999.

15. B.S. Kaliski and Y.L. Yin. On the Security of the RC5 Encryption Algorithm. RSA Laboratories Technical Report TR-602. Available at `www.rsa.com/rsalabs/aes/`.
16. G. Keating. Performance analysis of AES candidates on the 6805 CPU core. In Proceedings of *The Second AES Candidate Conference*, pages 109–114. March 22-23, 1999.
17. P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology — Crypto '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, 1996. Springer Verlag.
18. P.C. Kocher. Power analysis. Manuscript in preparation.
19. T. Knight. Personal communication. September 3, 1998
20. L.R. Knudsen. Applications of higher order differentials and partial differentials. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211, 1995. Springer Verlag.
21. NTT Laboratories. Java performance of the AES candidates. March 18, 1999. Available at `info.isl.ntt.co.jp/e2/`
22. National Institute of Standards and Technology. NIST's Efficiency testing for round 1 AES candidates. Available at `www.nist.gov/aes/`
23. R.L. Rivest. The RC5 encryption algorithm. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96, 1995. Springer Verlag.
24. R.L. Rivest, M.J.B. Robshaw, R. Sidney and Y.L. Yin. The RC6 Block Cipher. v1.1, August 20, 1998. Available at `www.rsa.com/rsalabs/aes/`
25. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson. Performance comparion of the AES submissions. In Proceedings of *The Second AES Candidate Conference*, pages 15–34. March 22-23, 1999.

Appendix - An Alternative Key Schedule

The RC6 key schedule algorithm is well studied and believed to be highly secure. Why bother to make a revised proposal?

First, it could be possible to significantly improve the running time of the key schedule algorithm, while preserving the desired level of security. This might be particularly important when the block cipher is used for hashing and it could be desirable to have a key setup algorithm that takes around the same time to run as it takes to encrypt a single block of data. While such "key agility" is useful in some applications, it is not so clear that key setup time is a significant feature for typical applications. For example, if a typical Internet packet contains 512 bytes (that is, 32 AES blocks), and if the key setup overhead is the same as encrypting three blocks, then the overhead for doing a key setup before encrypting the packet is under ten percent. In addition, if an application can cache the results of a key setup for later use, the key setup overhead can be amortized over the encryption of many packets. So, the key setup performance is not likely to be a significant issue for most applications. On the other hand, if this aspect of RC6 can be further optimized, why not explore such optimizations?

Second, the key setup algorithm RC6 may be difficult to implement on devices that are extremely memory (RAM) limited. Devices with under 128 bytes of RAM, for example, may not be able to use RC6 efficiently if they need to do key setup on the device (as opposed to doing the key setup off-device). Given Moore's Law, we feel that running AES on a device with such severely limited memory is unlikely to be a real concern. Nonetheless, it is interesting to explore what one might do to adapt RC6 to run on such limited devices.

We will outline a potential new key schedule for RC6, and for ease of reference let us refer to this new proposal as RC6a. The RC6a algorithm is identical to RC6 for encryption and decryption, but has a new key schedule. The RC6a key schedule is an "on the fly" round key generation algorithm and is capable of generating the round keys needed for RC6 using an amount of working storage that is essentially no larger than the encryption key itself (that is, no larger than the 16, 24, or 32-byte supplied key).

Since RC6 utilizes integer multiplication as a new primitive in the encryption algorithm it might be nice to exploit the strong cryptographic properties of multiplication in the key schedule algorithm. The RC6a key schedule algorithm does this, utilizing some of RC6's computational elements, such as the $f(x) = x(2x+1)$ operation.

Description of RC6a

Let c denote the number of words in the supplied encryption key, let w denote the word-size in bits and let r denote the number of rounds. For AES we have $r = 20$, $w = 32$, and $c = 4$, 6, or 8 32-bit words[3]. We require that c be at least two but less than 32. If fewer than two words of key are supplied, we pad with zeros out to two words, just as RC6 now pads out to four bytes (one word).

The RC6a key schedule is defined in terms of the array $S[0], \ldots, S[2r+c+3]$. For AES, $(2r+4) = 44$, so the array contains $44+c$ 32-bit words. Each of the 20 rounds of RC6 uses two round key words, and the pre- and post-whitening use two words of round key each. These are taken in order from the last 44 entries in the array starting with $S[c]$.

The first step is to pack the given c words of key into the first c words of the array S, so the given key occupies positions $S[0], \ldots, S[c-1]$. Given any consecutive c words of round key, the RC6a key schedule uses a nonlinear recurrence to define successive words of the round key array. More precisely, given c consecutive round keys, one can determine the next round key in the sequence. Thus, one can implement RC6a with only slightly more than c words of memory; one only needs to keep the latest c round keys at any time. The recurrence can also be run in the reverse direction, which allows decryption to be performed in a memory-compact manner as well.

The RC6a key schedule uses an order-c nonlinear recurrence of the following

[3] Note that the key schedule algorithm is well defined (as is RC6) for other word sizes, and for other values of c.

form. (It is now clear why we need c to be equal to at least two.)

$$S[j] = ((S[j-c] \oplus F_j) \lll F_j) \oplus R_j$$

where

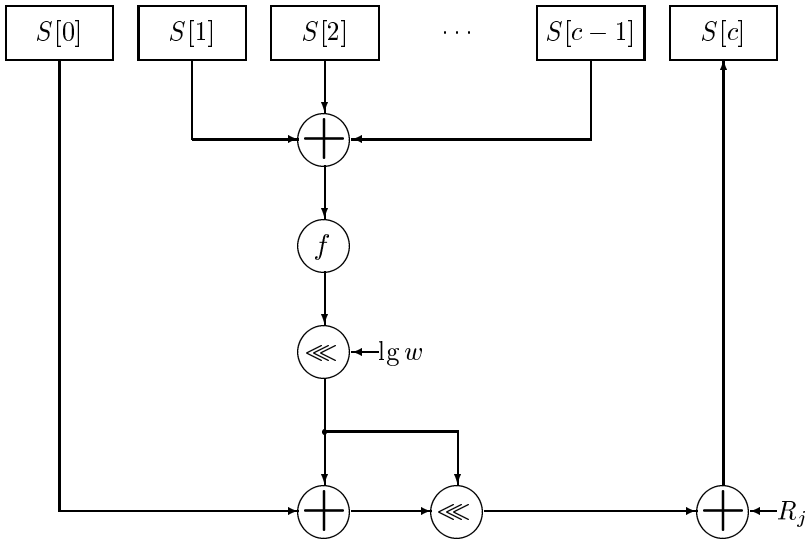
$$F_j = (X_j \times (2X_j + 1)) \lll \lg w$$

and

$$X_j = S[j-c+1] \oplus S[j-c+2] \oplus \ldots \oplus S[j-1]$$

We set $R_j = P_w + (j-c)Q_w + r$ where r is the number of rounds (for the AES submission $r = 20$) as a sequence of magic constants similar to those used in the RC6 key setup.

We note that as a result of this generation process the first c words of round key are only a mildly-mixed version of the c words of the supplied encryption key. One could potentially be concerned about this choice: the user might conceivably choose a "weak" key when used in this way, or this might give an avenue for attack against related keys. The first concern is not really relevant, since the RC6 key schedule could itself generate any conceivable pattern for the first c words of round key, however the issue of related key attacks against RC6a deserves further exploration[4]. Our analysis of this alternative key schedule is at a very preliminary stage and so additional research will likely highlight whether additional "mixing" is required before extracting the subkeys used during the encryption/decryption process.



Depending on the environment and the amount of optimization taking place the RC6a key schedule is around two to three times faster than the current key schedule for RC6. The timings given below are for a 128-bit key and the RAM requirements for the implementation on the Intel MCS51 were 25 bytes.

[4] One could reasonably argue that protection against related-key attacks should best be left to a pre-processing step, such as is commonly used for RC4.

	Cycles on Pentium Pro			*Cycles on Intel MCS51*
	Borland	*MSVC++ 4.0*	*Assembly*	
RC6 key setup	4710	2400	1100	27000
RC6a key setup	1260	840	540	13600

We note that the computation of X_j is simple in practice, since it is a "running XOR". For efficiency we can use the recurrence $X_j = X_{(j-1)} \oplus S[j-1] \oplus S[j-c]$ once we have computed X_c to get started. Furthermore, the memory requirements are quite modest; one only needs c consecutive values $S[j-c]$, ..., $S[j-1]$ in order to compute $S[j]$ (plus a small amount of working storage). Even for $c = 8$ (a 32-byte supplied key), we probably need less than 60 bytes of RAM to work in.

Note that the recurrence is invertible. Thus, for very tight memory situations, one can run the generation backwards to restore the original key $S[0]$, ... , $S[c-1]$. Separate storage to hold the initial key isn't required. Similarly, for decryption, the recurrence can be run backwards from the final state $S[44]$, ..., $S[43+c]$ during decryption. Thus for encryption or decryption, you can compute the round keys "on the fly" though to do this for decryption, you have to compute the final state first. With a little extra memory (that is with c extra words of memory) the final state can be saved too in preparation for either encryption or decryption.

This article was processed using the LaTeX macro package with LLNCS style