The RC6™ Block Cipher

Ronald L. Rivest[1], M.J.B. Robshaw[2], R. Sidney[2], and Y.L. Yin[2]

[1] M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA
rivest@theory.lcs.mit.edu
[2] RSA Laboratories, 2955 Campus Drive, Suite 400, San Mateo, CA 94403, USA
{matt,ray,yiqun}@rsa.com

Abstract. We introduce the RC6™ block cipher. RC6 is an evolutionary improvement of RC5, designed to meet the requirements of the Advanced Encryption Standard (AES). Like RC5, RC6 makes essential use of data-dependent rotations. New features of RC6 include the use of four working registers instead of two, and the inclusion of integer multiplication as an additional primitive operation. The use of multiplication greatly increases the diffusion achieved per round, allowing for greater security, fewer rounds, and increased throughput.

1 Introduction

RC6™ is a new block cipher submitted to NIST for consideration as the new Advanced Encryption Standard (AES).

The design of RC6 began with a consideration of RC5 [17] as a potential candidate for an AES submission. Modifications were then made to meet the AES requirements, to increase security, and to improve performance. The inner loop, however, is based around the same "half-round" found in RC5.

RC5 was intentionally designed to be extremely simple, to invite analysis shedding light on the security provided by extensive use of data-dependent rotations. Since RC5 was proposed in 1995, various studies [2, 4, 7, 10, 14, 18] have provided a greater understanding of how RC5's structure and operations contribute to its security. While no practical attack on RC5 has been found, the studies provide some interesting theoretical attacks, generally based on the fact that the "rotation amounts" in RC5 do not depend on all of the bits in a register. RC6 was designed to thwart such attacks, and indeed to thwart all known attacks, providing a cipher that can offer the security required for the lifespan of the AES.

To meet the requirements of the AES, a block cipher must handle 128-bit input/output blocks. While RC5 is an exceptionally fast block cipher, extending it to act on 128-bit blocks in the most natural manner would result in using two 64-bit working registers. The specified target architecture and languages for AES do not yet support 64-bit operations in an efficient and clean manner. Thus we have modified the design to use four 32-bit registers rather than two 64-bit registers. This has the advantage that we are doing two rotations per round rather than the one found in a half-round of RC5, and we are using more bits of data to determine rotation amounts in each round.

The philosophy of RC5 is to exploit operations (such as rotations) that are efficiently implemented on modern processors. RC6 continues this trend, and takes advantage of the fact that 32-bit integer multiplication is now efficiently implemented on most processors. Integer multiplication is a very effective "diffusion" primitive, and is used in RC6 to compute rotation amounts, so that the rotation amounts are dependent on *all* of the bits of another register, rather than just the low-order bits (as in RC5). As a result the new RC6 has much faster diffusion than RC5. This also allows RC6 to run with fewer rounds at increased security and with increased throughput.

We believe that RC6 is well-suited to meet all of the requirements of the Advanced Encryption Standard.

2 Details of RC6

Like RC5, RC6 is a fully parameterized family of encryption algorithms. A version of RC6 is more accurately specified as RC6-$w/r/b$ where the word size is w bits, encryption consists of a nonnegative number of rounds r, and b denotes the length of the encryption key in bytes. Since the AES submission is targeted at $w = 32$ and $r = 20$, we shall use RC6 as shorthand to refer to such versions. When any other value of w or r is intended in the text, the parameter values will be specified as RC6-w/r. Of particular relevance to the AES effort will be the versions of RC6 with 16-, 24-, and 32-byte keys.

For all variants, RC6-$w/r/b$ operates on units of four w-bit words using the following six basic operations. The base-two logarithm of w will be denoted by $\lg w$.

$a + b$	integer addition modulo 2^w
$a - b$	integer subtraction modulo 2^w
$a \oplus b$	bitwise exclusive-or of w-bit words
$a \times b$	integer multiplication modulo 2^w
$a \lll b$	rotate the w-bit word a to the left by the amount given by the least significant $\lg w$ bits of b
$a \ggg b$	rotate the w-bit word a to the right by the amount given by the least significant $\lg w$ bits of b

Note that in the description of RC6 the term "round" is somewhat analogous to the usual DES-like idea of a round: half of the data is updated by the other half; and the two are then swapped. In RC5, the term "half-round" was used to describe this style of action, and an RC5 round was deemed to consist of two half-rounds. This seems to have become a potential cause of confusion, and so RC6 reverts to using the term "round" in the more established way.

2.1 Key schedule

The key schedule of RC6-$w/r/b$ is practically identical to the key schedule of RC5-$w/r/b$. Indeed, the only difference is that for RC6-$w/r/b$, more words are derived from the user-supplied key for use during encryption and decryption. The key schedule algorithm is presented in full detail in the Appendix.

The user supplies a key of b bytes, where $0 \le b \le 255$. From this key, $2r+4$ words (w bits each) are derived and stored in the array $S[0,\ldots,2r+3]$. This array is used in both encryption and decryption.

2.2 Encryption and decryption

RC6 works with four w-bit registers A, B, C, D which contain the initial input plaintext as well as the output ciphertext at the end of encryption. The first byte of plaintext or ciphertext is placed in the least-significant byte of A; the last byte of plaintext or ciphertext is placed into the most-significant byte of D. We use $(A, B, C, D) = (B, C, D, A)$ to mean the parallel assignment of values on the right to registers on the left. Test vectors for encryption using RC6 are provided in the Appendix.

Encryption with RC6-$w/r/b$

Input: Plaintext stored in four w-bit input registers A, B, C, D
Number r of rounds
w-bit round keys $S[0,\ldots,2r+3]$

Output: Ciphertext stored in A, B, C, D

Procedure:
$B = B + S[0]$
$D = D + S[1]$
for $i = 1$ **to** r **do**
{
$t = (B \times (2B+1)) \lll \lg w$
$u = (D \times (2D+1)) \lll \lg w$
$A = ((A \oplus t) \lll u) + S[2i]$
$C = ((C \oplus u) \lll t) + S[2i+1]$
$(A, B, C, D) = (B, C, D, A)$
}
$A = A + S[2r+2]$
$C = C + S[2r+3]$

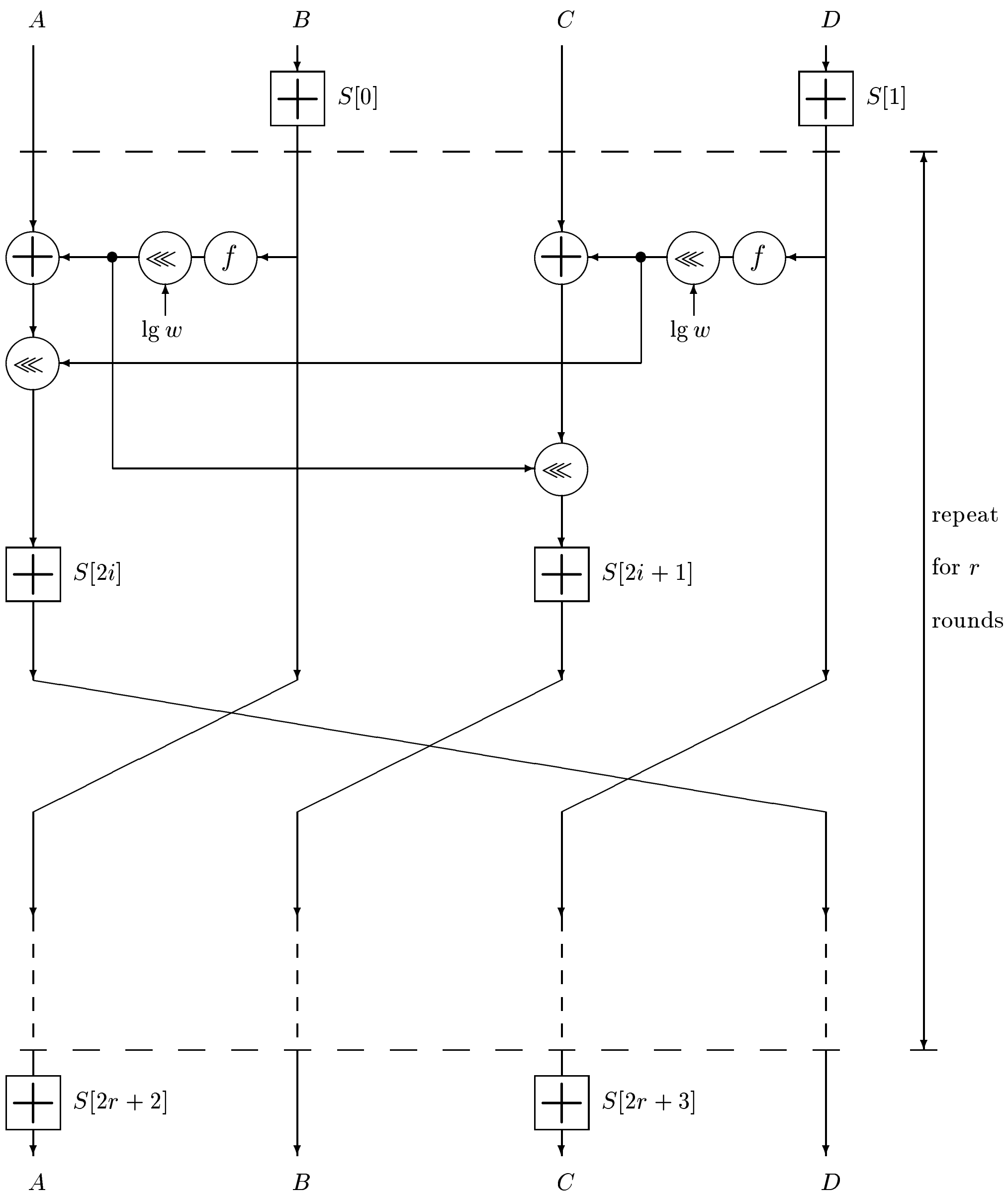

Fig. 1. Encryption with RC6-$w/r/b$. Here $f(x) = x \times (2x + 1)$.

Decryption with RC6-$w/r/b$

Input:	Ciphertext stored in four w-bit input registers A, B, C, D Number r of rounds w-bit round keys $S[0, \ldots, 2r+3]$
Output:	Plaintext stored in A, B, C, D

Procedure:	$C = C - S[2r+3]$
	$A = A - S[2r+2]$
	for $i = r$ **downto** 1 **do**
	{
	$(A, B, C, D) = (D, A, B, C)$
	$u = (D \times (2D+1)) \lll \lg w$
	$t = (B \times (2B+1)) \lll \lg w$
	$C = ((C - S[2i+1]) \ggg t) \oplus u$
	$A = ((A - S[2i]) \ggg u) \oplus t$
	}
	$D = D - S[1]$
	$B = B - S[0]$

3 Performance

In this section we provide some measurements of the encryption and decryption time of RC6 and also the time required for key setup.

The performance figures shown here for an optimized ANSI C implementation of RC6 were obtained using the compiler in Borland C^{++} Development Suite 5.0 as specified in the AES submission requirements. Performance was measured on a 266 MHz Pentium II with 32 Mbytes of RAM running Windows 95. To improve the accuracy of our timing measurements, maskable interrupts on the processor were disabled while our timing tests were executed; in addition, each set of the timing tests described in Sections 3.1 and 3.2 was executed 10 times, and we report the average of the times thereby obtained.

The figures shown for an assembly language implementation of RC6 were obtained on the same computer under identical conditions.

The performance figures given for an optimized Java implementation of RC6 were measured on a 180 MHz Pentium Pro with 64 Mbytes of RAM running Windows NT 4.0. This implementation was compiled with JavaSoft's JDK 1.1.6 compiler, and the performance of the resulting byte code was measured both with JavaSoft's JDK 1.1.6 interpreter (with JIT compilation disabled) and with Symantec Corporation's Java! JustInTime Compiler Version 210.054 for JDK 1.1.2. To improve the accuracy of our timing measurements, each set of the timing tests described in Sections 3.1 and 3.2 was executed 10 times, and we report the average of the times thereby obtained.

	scheme	cycles/block	blocks/sec at 200 MHz	Mbytes/sec at 200 MHz
ANSI C	RC6 encrypt (any size key)	616	325,000	5.19
ANSI C	RC6 decrypt (any size key)	566	353,000	5.65
Java (JDK)	RC6 encrypt (any size key)	16,200	12,300	0.197
Java (JDK)	RC6 decrypt (any size key)	16,500	12,100	0.194
Java (JIT)	RC6 encrypt (any size key)	1010	197,000	3.15
Java (JIT)	RC6 decrypt (any size key)	955	209,000	3.35
assembly	RC6 encrypt (any size key)	254	787,000	12.6
assembly	RC6 decrypt (any size key)	254	788,000	12.6
By way of comparison: RC5-32/16/16				
ANSI C	RC5-32/16/16 encrypt	328	610,000	4.9
Java (JIT)	RC5-32/16/16 encrypt	1,140	175,000	1.4
assembly	RC5-32/16/16 encrypt	148	1,350,000	10.8

Table 1. Speed of RC6 encryption and decryption in ANSI C, Java, and assembly. RC6 figures have been rounded to three significant digits.

Our timing figures have been scaled to 200 MHz, and it is expected that the tests NIST performs on the NIST reference platform will in general produce figures which are comparable to ours (as explained in Section 3.2, a possible exception to this expectation is key setup for ANSI C). Later in this section we give estimates for the performance of RC6 on 8-bit platforms (as might be found in smart cards) and some estimates for the requirements of a hardware implementation of the algorithm.

3.1 Encryption/decryption in ANSI C, Java, and assembly

The encryption figures given for RC6 in Table 1 do not include key setup, and are independent of the length of the user-supplied key. Timings in ANSI C

	scheme	cycles	μsecs at 200 MHz	key setups/sec at 200 MHz
ANSI C	RC6-32/20/16	$4,710$	23.5	$42,500$
Java (JDK)	RC6-32/20/16	$107,000$	537	$1,860$
Java (JIT)	RC6-32/20/16	$14,300$	71.4	$14,000$
ANSI C	RC6-32/20/24	$4,710$	23.6	$42,400$
Java (JDK)	RC6-32/20/24	$108,000$	542	$1,840$
Java (JIT)	RC6-32/20/24	$14,300$	71.5	$14,000$
ANSI C	RC6-32/20/32	$4,720$	23.6	$42,400$
Java (JDK)	RC6-32/20/32	$110,000$	548	$1,820$
Java (JIT)	RC6-32/20/32	$15,000$	75.1	$13,300$

Table 2. Key setup times for RC6 in ANSI C and Java. All figures have been rounded to three significant digits.

were obtained by encrypting and decrypting a single $3,000$-block piece of data in ECB mode (since this seemed to be a reasonable real-life use of the NIST-specified API); timings in Java were obtained by encrypting and decrypting a single $100,000$-block piece of data in ECB mode; and timings in assembly language were obtained by iteratively encrypting and decrypting a single block $3,000$ times.

Faster implementations may well be possible.

Slightly different methodologies may have been used to obtain the figures in Table 1 for RC5-32/16/16 which are provided for the purpose of comparison.

3.2 Key setup in ANSI C and Java

The time required for key setup with RC6-32/20/b when using keys of length $b = 128$, $b = 192$, and $b = 256$ bits is shown in Table 2. Timings for ANSI C were obtained by computing $3,000$ key schedules; timings for Java were obtained by computing $10,000$ key schedules. Because the NIST-specified C API for AES submissions inputs keys as hexadecimal ASCII strings (rather than as simple byte-strings), it was not used for timing key setup operations. Instead, an auxiliary key setup routine (which inputs keys as simple byte-strings) was used to obtain these figures.

Faster implementations may well be possible.

3.3 Estimates of performance on 8-bit platforms

Here we give some crude estimates for the performance of RC6 on an 8-bit processor. In particular, we consider estimates for Intel's popular MCS 51 Microcontroller family [5]. The estimates we shall make can be considered to hold for Phillips' 80C51 family of processors [16] as well, since these two families have very similar instruction sets and timings.

Encryption/decryption. We first consider the round function of RC6 (see Section 2.2). It consists of six additions, two exclusive-ors, two squarings, two left-rotates by five bits, and two left-rotates by a variable quantity r. Note that we have counted $B \times (2B + 1) = 2B^2 + B$ as a squaring and two additions.

These basic operations can be implemented on an 8-bit processor in the following way (ignoring addressing instructions):

1. A 32-bit addition can be computed using four 8-bit additions with carry (`ADDC`).
2. A 32-bit exclusive-or can be computed using four 8-bit exclusive-ors (`XRL`).
3. A 32-bit squaring can be computed using six 8-bit by 8-bit multiplications (`MUL`) and eleven additions with carry (`ADDC`). Note that six multiplications are enough since we only need the lower 32 bits of the 64-bit product.
4. Rotating a 32-bit word left by five bit positions can be computed by rotating the word right by one bit position three times and then permuting the four bytes. Note that rotating the word right by one bit position can be done using four byte rotations with carry (`RRC`).
5. Rotating a 32-bit word left by r can be computed by rotating the word left or right by one bit position r' times ($r' \leq 4$, with average two) and then permuting the four bytes appropriately. The five least-significant bits of r are used to determine r' and the permutation which can be controlled using jumps (`JB`).
6. Most instructions take one cycle except that `MUL` takes four cycles and `JB` takes two cycles.

Putting things together, we can estimate the total number of cycles needed for one round of RC6.

operation	instructions	cycles per operation	contributing cycles
add	4 `ADDC`	4	$4 \times 6 = 24$
exclusive-or	4 `XRL`	4	$4 \times 2 = 8$
squaring	6 `MUL`, 11 `ADDC`	35	$35 \times 2 = 70$
rotate left by 5	12 `RRC`	12	$12 \times 2 = 24$
rotate left by r (average over r)	8 `RRC` or `RLC`, 8 `JB`	24	$24 \times 2 = 48$
total			174

Taking conservative account of the addressing instructions, the pre-whitening, post-whitening and any additional overheads, we estimate that encrypting one block of data with RC6 requires around $(174 \times 20) \times 4 = 13,920$ cycles. According to Intel [5] each cycle takes one microsecond on a typical MCS 51 processor, and so an estimate for the encryption speed of RC6 on this particular processor is around $(1,000,000/13,920) \times 128 = 9.2$ Kbits/second.

An implementation of RC6 for the Intel 8051 was recently completed and gave a count of $13,535$ cycles for encrypting one block of data. This gives encouraging confirmation for the estimates we have just derived.

Key setup. The dominant loop in RC6 key setup is the last **for** loop (see Appendix). For $b = 16$, 24, 32 and $r = 20$, the number of iterations in this loop is $v = 3 \times \max\{20 \times 2 + 4, b/4\} = 132$, which is independent of b. So the estimates we make will be suitable for all key lengths of particular interest in the AES submission.

Each iteration in the loop uses four 32-bit additions, one rotate to the left by three, and one variable rotate to the left by r. In addition there are some 8-bit operations which we will just include as overheads. Following similar analysis for encryption, we obtain that the total number of cycles for each iteration (ignoring addressing instructions) is 52. Again, making a conservative estimate for the additional overheads we get $(52 \times 132) \times 4 = 27,456$ cycles to setup a 128-, 192- and 256-bit key, requiring about 27 milliseconds on an Intel MCS 51.

3.4 Hardware estimates

For most applications, an implementation of RC6 in software is probably the best choice. RC6's primitive operations (add, subtract, multiply, exclusive-or, and rotate) are very well-supported on modern microprocessors, and one therefore benefits from the exceptional effort and care that has gone into the design of such processors. Furthermore, using such an implementation technique allows one to easily ride the technology curve that we are all familiar with that results in Moore's Law (a factor of two improvement every 18 months).

However, in certain cases it might be desirable to have a custom integrated circuit implementing RC6. For example, one might wish to have the highest attainable speed, or to integrate other functions around the RC6 algorithm.

Because RC6 uses the familiar primitive operations noted above, one can take advantage of existing expertise in designing circuit modules for implementing these primitives. For example, while one could implement RC6 using standard gate-array technology, one would not benefit from the tremendous effort that has been put into designing efficient multiplication circuitry. Indeed, a gate-array implementation might very well perform more poorly than a processor-based implementation. But this is not an atypical situation, and one can easily design circuits that incorporate the best available multiplication circuitry as submodules.

For a custom or semi-custom implementation, the most relevant parameters are the silicon area, speed, and power consumption of a 32×32 integer multiplication. We have investigated this issue and find the following figures for this operation:

- 120×100 microns in area with a standard 0.25 micron CMOS process,
- around three nanoseconds required for each multiply, and
- a power consumption of five milliwatts.

We conservatively estimate that a 32-bit variable rotate (a "barrel shifter") would take half of the area of the multiplier and one nanosecond to do. Also we might estimate that a 32-bit full adder would take one quarter the multiplier area and around one nanosecond. We further observe that the function $f(x) = x(2x+1)(\bmod 2^w)$ can be computed by using only a multiplier that returns the bottom 32 bits of the 64-bit product rather than implementing a full 32×32 multiplier. We estimate that such a "partial" multiplier would take around 60% of the area of the full multiplier and three nannoseconds to do.

Considering a critical path for RC6, we can add up the relevant estimates.

operation	time (ns)	area (mm^2)
32×32 "partial" multiplication	3	0.007
32-bit xor	0	0.000
32-bit barrel-shifter	1	0.006
32-bit carry-propagate add	1	0.003
total	5	0.016

For an efficient implementation, one would want to have two such circuits on one chip. As a result, the total silicon area would be about 0.032 mm^2 for those parts directly relevant to RC6; we allow another 0.018 mm^2 (say) for control, I/O, etc. We therefore obtain an estimate that the computational area would be around 0.05 mm^2. Assuming that power consumption is proportional to area, we have a total power budget of about 21 milliwatts.

With 20 rounds per block, we have a total encryption time of approximately $5 \times 20 = 100$ nanoseconds for each block, giving an estimated data rate of around 1.3 Gbits/second. We would expect the decryption time to be similar to that required for encryption, and for both encryption and decryption time to be independent of the length of the user-supplied key.

We observe that these estimates are somewhat crude and also conservative. It would also be possible to unwind the main encryption loop 20 times in some modes of use which would allow for greatly improved performance at the cost of additional area and power consumption.

4 Implementation Issues

As might be expected from the description of the algorithm, RC6 is remarkably compact. Indeed, we estimate that for Intel's Pentium Pro microprocessor, a fast assembly language implementation of RC6 (one which runs significantly faster than the optimized C implementation we have provided with this submission) could easily be written with well under 256 bytes of code each for the tasks of key setup, block encryption, and block decryption.

Unlike many other encryption algorithms, RC6 does not use look-up tables during encryption. This means that RC6 code and data can readily fit within today's on-chip cache memory, and typically do so with room to spare. RC6 encryption and decryption make use of a 176-byte key schedule and a bare minimum of additional memory; to compute that 176-byte key schedule, the RC6 key setup process requires little more than an auxiliary array of approximately the same size as the user's supplied key. In addition, since the key schedule is only 176 bytes, it is possible to precompute and store the key schedules for hundreds of keys. Then switching to one of these keys only requires switching the pointer to the relevant key schedule, thereby providing key agility.

Given that the family of RC6-like algorithms is fully parameterized and that RC6 can be efficiently and compactly implemented, the cipher appears to be particularly versatile.

5 Design and Motivation

During the design of RC6 the following considerations were uppermost.

1. Security.
2. Simplicity.
3. Good performance.

5.1 Security and simplicity

The simplicity of RC5 has made it an attractive object for research. By being readily accessible to both crude and sophisticated analysis many people have been encouraged to look at the cipher and to assess the security it offers. RC6 was designed to build on the experience gained in using RC5 and to build on the security offered by a remarkably simple cipher.

One can view the design of RC6 as progressing through the following steps:

1. Start with the basic half-round loop of RC5:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad A = ((A \oplus B) \lll B) + S[i] \\
\qquad (A, B) = (B, A) \\
\quad \}
\end{array}
$$

2. Run two copies of RC5 in parallel: one on registers A, B and one on registers C, D.

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad A = ((A \oplus B) \lll B) + S[2i] \\
\qquad C = ((C \oplus D) \lll D) + S[2i+1] \\
\qquad (A, B) = (B, A) \\
\qquad (C, D) = (D, C) \\
\quad \}
\end{array}
$$

3. At the swap stage, instead of swapping A with B and C with D, permute the registers by $(A, B, C, D) = (B, C, D, A)$, so that the AB computation is mixed with the CD computation. At this stage the inner loop looks like:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad A = ((A \oplus B) \lll B) + S[2i] \\
\qquad C = ((C \oplus D) \lll D) + S[2i+1] \\
\qquad (A, B, C, D) = (B, C, D, A) \\
\quad \}
\end{array}
$$

4. Mix up the AB computation with the CD computation further, by switching where the rotation amounts come from between the two computations:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad A = ((A \oplus B) \lll D) + S[2i] \\
\qquad C = ((C \oplus D) \lll B) + S[2i+1] \\
\qquad (A, B, C, D) = (B, C, D, A) \\
\quad \}
\end{array}
$$

5. Instead of using B and D in a straightforward manner as above, we use transformed versions of these registers, for some suitable transformation. Our security goals are that the data-dependent rotation amount that will be derived from the output of this transformation should depend on all bits of the input word and that the transformation should provide good mixing within the word. The particular choice of this transformation for

RC6 is the function $f(x) = x(2x+1)(\bmod 2^w)$ followed by a left rotation by five bit positions. This transformation appears to meet our security goals while taking advantage of simple primitives that are efficiently implemented on most modern processors. Note that $f(x)$ is one-to-one modulo 2^w, and that the high-order bits of $f(x)$, which determine the rotation amount used, depend heavily on all the bits of x.
This gives us:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad t = (B \times (2B+1)) \lll 5 \\
\qquad u = (D \times (2D+1)) \lll 5 \\
\qquad A = ((A \oplus t) \lll u) + S[2i] \\
\qquad C = ((C \oplus u) \lll t) + S[2i+1] \\
\qquad (A, B, C, D) = (B, C, D, A) \\
\quad \}
\end{array}
$$

6. At the beginning and end of the r rounds, add pre-whitening and post-whitening steps. Without these steps, the plaintext reveals part of the input to the first round of encryption and the ciphertext reveals part of the input to the last round of encryption. The pre- and post-whitening steps help to disguise this and leaves us with RC6:

$$
\begin{array}{l}
B = B + S[0] \\
D = D + S[1] \\
\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
\quad \{ \\
\qquad t = (B \times (2B+1)) \lll 5 \\
\qquad u = (D \times (2D+1)) \lll 5 \\
\qquad A = ((A \oplus t) \lll u) + S[2i] \\
\qquad C = ((C \oplus u) \lll t) + S[2i+1] \\
\qquad (A, B, C, D) = (B, C, D, A) \\
\quad \} \\
A = A + S[2r+2] \\
C = C + S[2r+3]
\end{array}
$$

While it might appear that the evolution from RC5 to RC6 was straightforward, it in fact involved the design and analysis of literally dozens of alternatives. RC6 is the design that captures the spirit of our three goals of security, simplicity and performance the most effectively.

Note that in the preceding development, the decision to expand to four 32-bit registers was made first (for performance reasons), and then the decision to use the quadratic function $f(x) = x(2x+1)(\bmod 2^w)$ was made later. If we had decided to stick with a two register version of RC6 then we might have had the following encryption scheme as an intermediate:

$$
\begin{aligned}
&B = B + S[0] \\
&\textbf{for } i = 1 \textbf{ to } r \textbf{ do} \\
&\quad \{ \\
&\qquad t = B \times (2B + 1) \lll 5 \\
&\qquad A = ((A \oplus t) \lll t) + S[i] \\
&\qquad (A, B) = (B, A) \\
&\quad \} \\
&A = A + S[r + 1]
\end{aligned}
$$

This variant of RC6 may be of independent interest, particularly when support for 64-bit arithmetic in C improves. However we merely mention this as an aside here.

5.2 Good performance for a given level of security

Since the publication of RC5 there have been several notable papers providing substantive progress in the analysis of RC5 [2, 7, 10, 18]. While the latest techniques demonstrate that RC5-32/12/b, i.e. a 12-round version of RC5, might not be suitable for longer-term security needs, these attacks currently fall short of providing any real avenue for practical attack against a 16-round version.

Most existing cryptanalytic results on RC5 depend on what might be viewed as a relatively slow avalanche of change between rounds. The integer addition helps to provide a reasonable amount of change due to the effect of the carry, but the most dramatic changes take place when two different rotation amounts are used at a similar point during the encryption of two related plaintexts. Typically an attacker would aim to control the evolution of the differences from round to round and, in versions of RC5 with fewer rounds, this can allow an attack to be mounted.

The incremental changes in arriving at RC6 from RC5 have already been outlined. Two significant changes are the introduction of the quadratic function $B \times (2B + 1)$ (resp. $D \times (2D + 1)$) and the fixed rotation by five bits.

The quadratic function is aimed at providing a faster rate of diffusion thereby improving the chances that simple differentials will spoil rotation amounts much sooner than is accomplished with RC5. The quadratically transformed values of B and D are used in place of B and D to modify the registers A and C, increasing the nonlinearity of the scheme while not losing any entropy (since the transformation is a permutation). The fixed rotation by five bits plays a simple yet important role in complicating both linear and differential cryptanalysis.

6 Security

We conjecture that to attack RC6 the best approach[3] available to the cryptanalyst is that of exhaustive search for the b-byte encryption key (or the expanded

[3] Note that we are using RC6 to refer to the AES specific variant with $w = 32$ and $r = 20$.

key array $S[0, \ldots, 43]$ when the user-supplied encryption key is particularly long). The work effort required for this is $\min\{2^{8b}, 2^{1408}\}$ operations.

The more advanced attacks of differential [1] and linear cryptanalysis [13], while being feasible on small-round versions of the cipher, do not extend well to attacking the full 20-round RC6 cipher. The main difficulty is that it is hard to find good iterative characteristics or linear approximations with which an attack might be mounted.

It is an interesting challenge to establish the most appropriate goals for security against these more advanced attacks. To succeed, these attacks typically require large amounts of data, and obtaining 2^a blocks of known or chosen plaintext-ciphertext pairs is a very different task from trying to recover one key from among 2^a possibilities (this latter task can be readily parallelized). It is worth observing that with a cipher running at the rate of one terabit per second (that is, encrypting data at the rate of 10^{12} bits/second), the time required for 50 computers working in parallel to encrypt 2^{64} blocks of data is more than a year; to encrypt 2^{80} blocks of data is more than $98,000$ years; and to encrypt 2^{128} blocks of data is more than 10^{19} years.

While having a data requirement of 2^{64} blocks of data for a successful attack might be viewed as sufficient in practical terms, we have aimed to provide a much greater level of security. The community as a whole will decide which level of security a cipher, in particular an AES candidate, should satisfy. Should this be less than a data requirement of 2^{128} blocks of data then the number of rounds of RC6 could potentially be reduced from our initial suggestion of 20 rounds, thereby providing an improvement in performance.

For attacking an eight-round version of the cipher, RC6-32/8/b, one can construct six-round characteristics or linear approximations. Assuming that these could be used to attack the eight-round version of the cipher (an assumption that, while reasonable, overlooks a vast number of practical details) the estimated data required to mount a differential cryptanalytic attack on RC6-32/8/b would be around 2^{76} chosen plaintext pairs, and to mount a linear cryptanalytic attack would be around 2^{60} known plaintexts. Consideration of more sophisticated phenomena such as *differentials* [12] and *linear hulls* [15], together with more customized techniques will reduce these figures by a moderate amount, but they provide a reasonable illustration of the security that might be offered by a version of RC6 with a few rounds.

Currently, it seems that a differential attack on the full 20-round RC6 cipher appears to be most easily accomplished by using a six-round iterative characteristic (although we have identified useful three- and four-round characteristics). Considering a variety of options, the probability of one of the best 18-round characteristics we are aware of in attacking RC6 is around 2^{-264} and uses integer subtraction as the notion of difference. (For some technical reasons, using exclusive-or as the notion of difference can be more problematical.) To use this characteristic in an attack would require more than the total number of available chosen plaintext/ciphertext pairs. While we expect the amount of data required for an attack to drop as more detailed analysis takes place we do not believe that differential cryptanalysis can be successfully applied to RC6.

To mount a linear cryptanalytic attack, there appear to be two different options. The first might be to find a linear approximation over several rounds that uses a linear approximation across the quadratic function. Since there appear to be some very suitable linear approximations using the least significant bits of this function, this might be an appealing strategy. Indeed, one can establish useful six-round iterative linear approximations that can be used, at least in principle, to attack reduced-round versions of RC6. However, the bias of these approximations drops rapidly as more rounds are added, and soon the amount of data required for a successful attack exceeds the amount of data available. Instead, we note that an attacker might well pursue an alternative approach.

It is possible to find a two-round iterative linear approximation that does not use an approximation across the combination of the quadratic function and fixed rotation by five bit positions. Using basic but established techniques to predict the bias of such an approximation, we observe that the data requirements to exploit this approximation over a version of RC6 with 16 rounds is about 2^{142} known plaintexts. Further analysis demonstrated that additional techniques could be used to bring the data requirements down to a little under 2^{128} known plaintexts. This provided our rationale for choosing 20 rounds for RC6.

With our current knowledge, the most successful avenue for a linear cryptanalytic attack on RC6 would be to use the two-round iterative approximation we have just mentioned to build up an 18-round linear approximation with which to attack the cipher. Using the same techniques as before to predict the data requirements to use this approximation at first sight we might need 2^{182} known plaintexts, an amount which exceeds the available data. Enhanced techniques can be used to reduce this figure by a moderate amount, but in the final assessment we believe that the number of known plaintexts needed to exploit this approximation readily exceeds the maximum number of plaintexts available. We conclude that a linear cryptanalytic attack against RC6 is not possible using these techniques. Further, we believe that the use of more sophisticated linear approximations and the use of other more advanced linear approximation techniques are exceptionally unlikely to provide sufficient gains as to offer an attack requiring less than 2^{128} known plaintexts.

We are aware of several potential enhancements to the essential attacks we have described (in particular, the use of truncated and higher-order differentials [9]), and we are also aware of some alternative approaches. However, all these techniques have so far failed to improve on the attacks outlined here, and we believe that all currently available sophisticated cryptanalytic attacks will require more data than there is available. A report on our work and findings is in preparation.

RC6 can easily be implemented in such a way as to be invulnerable to "timing attacks" [11]. Many modern processors have constant-time rotation and multiplication instructions. Other processors may have a rotation or shift time that depends linearly with the amount of rotation, but in this case it is usually easy to arrange the work so that the total compute time is data-independent (for example, by computing a rotate of t bits using a left-shift of t bits and a right-shift of $w-t$ bits). In either case, the RC6 encrypt/decrypt time is data-independent, causing any potential timing attacks to fail.

Studies of RC5 have failed to reveal any weakness in the key setup. This provided one of the motivations for using the same key setup in RC6 as was used in RC5. The process of transforming the supplied key to the table of round keys appears to be well-modeled by a pseudo-random process. Thus, while there is no proof that no two keys yield the same table of round keys, it appears to be highly unlikely. It can be estimated that the chance that there exist two 256-bit keys yielding the same table of 44 32-bit round keys is approximately $2^{2\times 256-44\times 32} = 2^{-896} = 10^{-270}$ (approximately). We feel that there is value in the "one-way" structure of the key-setup routine that is more important than the (infinitesimal) chance that there might be two keys that yield the same table of round keys. One such value is the protection it provides against related-key attacks, for example.

We can summarize our claims on the security of RC6 as follows:

– The best attack on RC6 appears to be exhaustive search for the user-supplied encryption key.
– The data requirements to mount more sophisticated attacks on RC6 such as differential and linear cryptanalysis exceed the available data.
– There are no known examples of what might be termed "weak" keys.

7 Flexibility and Future Directions

As we have already observed RC6 provides the user with a great amount of flexibility with regards to the size of the encryption key, the number of rounds and the word size of the basic computational unit.

While the submission of RC6 for consideration as the forthcoming AES is based around the use of 32-bit words (giving a block size of 128 bits), future developments and market demand might encourage an extension of RC6 to other block sizes. Of most importance may be block sizes of 256 bits which would take advantage of a word size of 64 bits and the performance offered by the next generation of system architectures (see for example [3, 6]).

We note further that the structure of RC6 allows one to exploit a certain degree of parallelism in the encryption and decryption routines. For example, the computation of t and u at each round can be computed in parallel as can the updates of A and C. As processors move to include an increasingly amount of internal parallelism (e.g., with the move to superscalar architectures), implementations of RC6 should show increased throughput.

8 Conclusions

RC6 is a secure, compact and simple block cipher. It offers good performance and considerable flexibility. Furthermore its simplicity will allow analysts to quickly refine and improve our estimates of its security.

9 Acknowledgements

Many people have been extremely helpful during the design of RC6. In particular we would like to thank Burt Kaliski, Scott Contini, and Jeff Ylvisaker of RSA Laboratories, Tom Knight of M.I.T., and Phil Rogaway of UC Davis.

References

1. E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, New York, 1993.
2. A. Biryukov and E. Kushilevitz. Improved cryptanalysis of RC5. To appear in proceedings of *Advances in Cryptology — Eurocrypt '98*, *Lecture Notes in Computer Science*, 1998. Springer Verlag.
3. Hewlett Packard. Strategy description, May 22, 1997. Available at `http://www.hp.com/gsy/software/64bit/64bitwp.html`.
4. M.H. Heys. Linearly weak keys of RC5. *IEE Electronic Letters*, Vol. 33, pages 836–838, 1997.
5. Intel Corporation. MCS 51 Microcontroller Family User's Manual. February 1994.
6. Intel Corporation. *The Next Generation of Microprocessor Architecture.* October, 1997. Available at `http://www.intel.com/pressroom/archive/backgrnd/sp101497.HTM`.
7. B.S. Kaliski and Y.L. Yin. On differential and linear cryptanalysis of the RC5 encryption algorithm. In D. Coppersmith, editor, *Advances in Cryptology — Crypto '95*, volume 963 of *Lecture Notes in Computer Science*, pages 171–184, 1995. Springer Verlag.
8. B.S. Kaliski and Y.L. Yin. On the Security of the RC5 Encryption Algorithm. RSA Laboratories Technical Report TR-602. To appear.
9. L.R. Knudsen. Truncated and higher order differentials. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211, 1994. Springer Verlag.
10. L.R. Knudsen and W. Meier. Improved differential attacks on RC5. In N. Koblitz, editor, *Advances in Cryptology — Crypto '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 216–228, 1996. Springer Verlag.
11. 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.
12. X. Lai, J.L. Massey and S. Murphy. Markov ciphers and differential cryptanalysis. In D.W. Davies, editor, *Advances in Cryptology — Eurocrypt '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38, 1991. Springer-Verlag.
13. M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397, 1994. Springer-Verlag.
14. S. Moriai, K. Aoki, and K. Ohta. Key-dependency of linear probability of RC5. March 1996. To appear in *IEICE Trans. Fundamentals.*
15. K. Nyberg. Linear approximation of block ciphers. In A.D. Santis, editor, *Advances in Cryptology — Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pages 439–444, 1994. Springer-Verlag.

16. Phillips Semiconductors. 80C51 Family Programmer's Guide and Instruction Set. November, 1996.
17. 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.
18. A. A. Selcuk. New results in linear cryptanalysis of RC5. In S. Vaudenay, editor, *Fast Software Encryption*, volume 1372 of *Lecture Notes in Computer Science*, pages 1–16, 1998, Springer-Verlag.

Appendix

Test vectors

Test vectors for encryption with RC6

plaintext	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
user key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ciphertext	8f c3 a5 36 56 b1 f7 78 c1 29 df 4e 98 48 a4 1e
plaintext	02 13 24 35 46 57 68 79 8a 9b ac bd ce df e0 f1
user key	01 23 45 67 89 ab cd ef 01 12 23 34 45 56 67 78
ciphertext	52 4e 19 2f 47 15 c6 23 1f 51 f6 36 7e a4 3f 18
plaintext	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
user key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	00 00 00 00 00 00 00 00
ciphertext	6c d6 1b cb 19 0b 30 38 4e 8a 3f 16 86 90 ae 82
plaintext	02 13 24 35 46 57 68 79 8a 9b ac bd ce df e0 f1
user key	01 23 45 67 89 ab cd ef 01 12 23 34 45 56 67 78
	89 9a ab bc cd de ef f0
ciphertext	68 83 29 d0 19 e5 05 04 1e 52 e9 2a f9 52 91 d4
plaintext	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
user key	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ciphertext	8f 5f bd 05 10 d1 5f a8 93 fa 3f da 6e 85 7e c2
plaintext	02 13 24 35 46 57 68 79 8a 9b ac bd ce df e0 f1
user key	01 23 45 67 89 ab cd ef 01 12 23 34 45 56 67 78
	89 9a ab bc cd de ef f0 10 32 54 76 98 ba dc fe
ciphertext	c8 24 18 16 f0 d7 e4 89 20 ad 16 a1 67 4e 5d 48

Key schedule for RC6

The key schedule of RC6-$w/r/b$ is practically identical to the key schedule of RC5-$w/r/b$ and is presented here. The only difference is that more words are derived from the user-supplied key for use during encryption and decryption. The user supplies a key of b bytes. Sufficient zero bytes are appended to give a key length equal to a non-zero integral number of words; these key bytes are then loaded in little-endian fashion into an array of c w-bit words $L[0], \ldots, L[c-1]$. Thus the first byte of key is stored as the low-order byte of $L[0]$, etc., and $L[c-1]$ is padded with high-order zero bytes if necessary. (Note that if $b = 0$ then $c = 1$ and $L[0] = 0$.) The number of w-bit words that will be generated for the additive round keys is $2r + 4$ and these are stored in the array $S[0, ..., 2r + 3]$.

The constants $P_{32} =$ `B7E15163` and $Q_{32} =$ `9E3779B9` (hexadecimal) are the same "magic constants" as used in the RC5 key schedule. The value of P_{32} is derived from the binary expansion of $e - 2$, where e is the base of the natural logarithm function. The value of Q_{32} is derived from the binary expansion of $\phi - 1$, where ϕ is the Golden Ratio. Similar definitions from RC5 for P_{64} etc. can be used for versions of RC6 with other word sizes. These values are somewhat arbitrary, and other values could be chosen to give "custom" or proprietary versions of RC6.

Key schedule for RC6-$w/r/b$

Input:	User-supplied b byte key preloaded into the c-word array $L[0, \ldots, c-1]$ Number r of rounds
Output:	w-bit round keys $S[0, \ldots, 2r+3]$
Procedure:	$S[0] = P_w$

$S[0] = P_w$

for $i = 1$ **to** $2r + 3$ **do**
  $S[i] = S[i-1] + Q_w$

$A = B = i = j = 0$

$v = 3 \times \max\{c, 2r + 4\}$
for $s = 1$ **to** v **do**
  {
    $A = S[i] = (S[i] + A + B) \lll 3$
    $B = L[j] = (L[j] + A + B) \lll (A + B)$
    $i = (i + 1) \bmod (2r + 4)$
    $j = (j + 1) \bmod c$
  }

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