The RC5 Encryption Algorithm*

Ronald L. Rivest

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, Mass. 02139
`rivest@theory.lcs.mit.edu`

Abstract. This document describes the RC5 encryption algorithm, a fast symmetric block cipher suitable for hardware or software implementations. A novel feature of RC5 is the heavy use of *data-dependent rotations*. RC5 has a variable word size, a variable number of rounds, and a variable-length secret key. The encryption and decryption algorithms are exceptionally simple.

1 Introduction

RC5 was designed with the following objectives in mind.

- RC5 should be a *symmetric block cipher*. The same secret cryptographic key is used for encryption and for decryption. The plaintext and ciphertext are fixed-length bit sequences (blocks).
- RC5 should be *suitable for hardware or software*. This means that RC5 should use only computational primitive operations commonly found on typical microprocessors.
- RC5 should be *fast*. This more-or-less implies that RC5 be *word-oriented*: the basic computational operations should be operators that work on full words of data at a time.
- RC5 should be *adaptable to processors of different word-lengths*. For example, as 64-bit processors become available, it should be possible for RC5 to exploit their longer word length. Therefore, the number w of bits in a word is a *parameter* of RC5; different choices of this parameter result in different RC5 algorithms.
- RC5 should be iterative in structure, with a *variable number of rounds*. The user can explicitly manipulate the trade-off between higher speed and higher security. The number of rounds r is a second parameter of RC5.
- RC5 should have *a variable-length cryptographic key*. The user can choose the level of security appropriate for his application, or as required by external considerations such as export restrictions. The key length b (in bytes) is thus a third parameter of RC5.
- RC5 should be *simple*. It should be easy to implement. More importantly, a simpler structure is perhaps more interesting to analyze and evaluate, so that the cryptographic strength of RC5 can be more rapidly determined.

---

* RC5 is a trademark of RSA Data Security. Patent pending.

- RC5 should have a *low memory requirement*, so that it may be easily implemented on smart cards or other devices with restricted memory.
- (Last but not least!) RC5 should provide *high security* when suitable parameter values are chosen.

In addition, during the development of RC5, we began to focus our attention on a intriguing new cryptographic primitive: *data-dependent rotations*, in which one word of intermediate results is cyclically rotated by an amount determined by the low-order bits of another intermediate result. We thus developed an additional goal.

- RC5 should *highlight the use of data-dependent rotations*, and encourage the assessment of the cryptographic strength data-dependent rotations can provide.

The RC5 encryption algorithm presented here hopefully meets all of the above goals. Our use of "hopefully" refers of course to the fact that this is still a new proposal, and the cryptographic strength of RC5 is still being determined.

2 A Parameterized Family of Encryption Algorithms

In this section we discuss in somewhat greater detail the parameters of RC5, and the tradeoffs involved in choosing various parameters.

As noted above, RC5 is *word*-oriented: all of the basic computational operations have w-bit words as inputs and outputs. RC5 is a block-cipher with a two-word input (plaintext) block size and a two-word (ciphertext) output block size. The nominal choice for w is 32 bits, for which RC5 has 64-bit plaintext and ciphertext block sizes. RC5 is well-defined for any $w > 0$, although for simplicity it is proposed here that only the values 16, 32, and 64 be "allowable."

The number r of rounds is the second parameter of RC5. Choosing a larger number of rounds presumably provides an increased level of security. We note here that RC5 uses an "expanded key table," S, that is derived from the user's supplied secret key. The size t of table S also depends on the number r of rounds: S has $t = 2(r + 1)$ words. Choosing a larger number of rounds therefore also implies a need for somewhat more memory.

There are thus several distinct "RC5" algorithms, depending on the choice of parameters w and r. We summarize these parameters below:

w This is the *word size*, in bits; each word contains $u = (w/8)$ 8-bit bytes. The nominal value of w is 32 bits; allowable values of w are 16, 32, and 64. RC5 encrypts two-word blocks: plaintext and ciphertext blocks are each $2w$ bits long.

r This is the number of rounds. Also, the expanded key table S contains $t = 2(r + 1)$ words. Allowable values of r are 0, 1, ..., 255.

In addition to w and r, RC5 has a variable-length secret cryptographic key, specified by parameters b and K:

b The number of bytes in the secret key K. Allowable values of b are 0, 1, ..., 255.

K The b-byte secret key: $K[0]$, $K[1]$, ..., $K[b-1]$.

For notational convenience, we designate a particular (parameterized) RC5 algorithm as *RC5-w/r/b*. For example, RC5-32/16/10 has 32-bit words, 16 rounds, a 10-byte (80-bit) secret key variable, and an expanded key table of $2(16{+}1) = 34$ words. Parameters may be dropped, from last to first, to talk about RC5 with the dropped parameters unspecified. For example, one may ask: How many rounds should one use in RC5-32?

We propose RC5-32/12/16 as providing a "nominal" choice of parameters. That is, the nominal values of the parameters provide for $w = 32$ bit words, 12 rounds, and 16 bytes of key. Further analysis is needed to analyze the security of this choice. For RC5-64, we suggest increasing the number of rounds to $r = 16$.

We suggest that in an implementation, all of the parameters given above may be packaged together to form an *RC5 control block*, containing the following fields:

v 1 byte version number; `10` (hex) for version 1.0 here.

w 1 byte.

r 1 byte.

b 1 byte.

K b bytes.

A control block is thus represented using $b{+}4$ bytes. For example, the control block

```
10 20 0C 0A 20 33 7D 83 05 5F 62 51 BB 09 (in hexadecimal)
```

specifies an RC5 algorithm (version 1.0) with 32-bit words, 12 rounds, and a 10-byte (80-bit) key "`20 33 ... 09`". RC5 "key-management" schemes would then typically manage and transmit entire RC5 control blocks, containing all of the relevant parameters in addition to the usual secret cryptographic key variable.

2.1 Discussion of Parameterization

In this section we discuss the extensive parameterization that RC5 provides.

We should first note that it is not intended that RC5 be secure for all possible parameter values. For example, $r = 0$ provides essentially no encryption, and $r = 1$ is easily broken. And choosing $b = 0$ clearly gives no security.

On the other hand, choosing the maximum allowable parameter values would be overkill for most applications.

We allow a range of parameter values so that users may select an encryption algorithm whose security and speed are optimized for their application, while providing an evolutionary path for adjusting their parameters as necessary in the future.

As an example, consider the problem of replacing DES with an "equivalent" RC5 algorithm. One might reasonable choose RC5-32/16/7 as such a replacement. The input/output blocks are $2w = 64$ bits long, just as in DES. The number of rounds is also the same, although each RC5 round is more like two DES rounds since all data registers, rather than just half of them, are updated in one RC5 round. Finally, DES and RC5-32/16/7 each have 56-bit (7-byte) secret keys.

Unlike DES, which has no parameterization and hence no flexibility, RC5 permits upgrades as necessary. For example, one can upgrade the above choice for a DES replacement to an 80-bit key by moving to RC5-32/16/10. As technology improves, and as the true strength of RC5 algorithms becomes better understood through analysis, the most appropriate parameter values can be chosen.

The choice of r affects both encryption speed and security. For some applications, high speed may be the most critical requirement—one wishes for the best security obtainable within a given encryption time requirement. Choosing a small value of r (say $r = 6$) may provide some security, albeit modest, within the given speed constraint.

In other applications, such as key management, security is the primary concern, and speed is relatively unimportant. Choosing $r = 32$ rounds might be appropriate for such applications. Since RC5 is a new design, further study is required to determine the security provided by various values of r; RC5 users may wish to adjust the values of r they use based on the results of such studies.

Similarly, the word size w also affects speed and security. For example, choosing a value of w larger than the register size of the CPU can degrade encryption speed. The word size $w = 16$ is primarily for researchers who wish to examine the security properties of a natural "scaled-down" RC5. As 64-bit processors become common, one can move to RC5-64 as a natural extension of RC5-32. It may also be convenient to specify $w = 64$ (or larger) if RC5 is to be used as the basis for a hash function, in order to have 128-bit (or larger) input/output blocks.

It may be considered unusual and risky to specify an encryption algorithm that permits insecure parameter choices. We have two responses to this criticism:

1. A fixed set of parameters may be at least as dangerous, since the parameters can not be increased when necessary. Consider the problem DES has now: its key size is too short, and there is no easy way to increase it.
2. It is expected that implementors will provide implementations that ensure that suitably large parameters are chosen. While unsafe choices might be usable in principle, they would be forbidden in practice.

It is not expected that a typical RC5 implementation will work with any RC5 control block. Rather, it may only work for certain fixed parameter values, or parameters in a certain range. The parameters w, r, and b in a received or transmitted RC5 control block are then merely used for *type-checking*—values other than those supported by the implementation will be disallowed. The flexibility of RC5 is thus utilized at the system design stage, when the appropriate parameters are chosen, rather than at run time, when unsuitable parameters might be chosen by an unwary user.

Finally, we note that RC5 might be used in some applications that do not require cryptographic security. For example, one might consider using RC5-32/8/0 (with no secret key) applied to inputs 0, 1, 2, ... to generate a sequence of pseudo-random numbers to be used in a randomized computation.

3 Notation and RC5 Primitive Operations

We use $\lg(x)$ to denote the base-two logarithm of x.

RC5 uses only the following three primitive operations (and their inverses).

1. Two's complement addition of words, denoted by "+". This is modulo-2^w addition. The inverse operation, subtraction, is denoted "$-$".
2. Bit-wise exclusive-OR of words, denoted by $\oplus$.
3. A left-rotation (or "left-spin") of words: the cyclic rotation of word x left by y bits is denoted $x \lll y$. Here y is interpreted modulo w, so that when w is a power of two, only the $\lg(w)$ low-order bits of y are used to determine the rotation amount. The inverse operation, right-rotation, is denoted $x \ggg y$.

These operations are directly and efficiently supported by most processors.

A distinguishing feature of RC5 is that the rotations are rotations by "variable" (plaintext-dependent) amounts. We note that on modern microprocessors, a variable-rotation $x \lll y$ takes *constant time*: the time is independent of the rotation amount y. We also note that rotations are the only non-linear operator in RC5; there are no nonlinear substitution tables or other nonlinear operators. The strength of RC5 depends heavily on the cryptographic properties of data-dependent rotations.

4 The RC5 Algorithm

In this section we describe the RC5 algorithm, which consists of three components: a *key expansion* algorithm, an *encryption* algorithm, and a *decryption* algorithm. We present the encryption and decryption algorithms first.

Recall that the plaintext input to RC5 consists of two w-bit words, which we denote A and B. Recall also that RC5 uses an *expanded key table*, $S[0...t-1]$, consisting of $t = 2(r+1)$ w-bit words. The key-expansion algorithm initializes S from the user's given secret key parameter K. (We note that the S table in RC5 encryption is not an "S-box" such as is used by DES; RC5 uses the entries in S sequentially, one at a time.)

We assume standard *little-endian* conventions for packing bytes into input/output blocks: the first byte occupies the low-order bit positions of register A, and so on, so that the fourth byte occupies the high-order bit positions in A, the fifth byte occupies the low-order bit positions in B, and the eighth (last) byte occupies the high-order bit positions in B.

4.1 Encryption

We assume that the input block is given in two w-bit registers A and B. We also assume that key-expansion has already been performed, so that the array $S[0...t-1]$ has been computed. Here is the encryption algorithm in pseudo-code:

$A = A + S[0];$
$B = B + S[1];$
for $i = 1$ **to** r **do**
  $A = ((A \oplus B) \lll B) + S[2*i];$
  $B = ((B \oplus A) \lll A) + S[2*i+1];$

The output is in the registers A and B.

We note the exceptional simplicity of this 5-line algorithm.

We also note that each RC5 round updates *both* registers A and B, whereas a "round" in DES updates only half of its registers. An RC5 "half-round" (one of the assignment statements updating A or B in the body of the loop above) is thus perhaps more analogous to a DES round.

4.2 Decryption

The decryption routine is easily derived from the encryption routine.

for $i = r$ **downto** 1 **do**
  $B = ((B - S[2*i+1]) \ggg A) \oplus A;$
  $A = ((A - S[2*i]) \ggg B) \oplus B;$
$B = B - S[1];$
$A = A - S[0];$

4.3 Key Expansion

The key-expansion routine expands the user's secret key K to fill the expanded key array S, so that S resembles an array of $t = 2(r+1)$ random binary words determined by K. The key expansion algorithm uses two "magic constants," and consists of three simple algorithmic parts.

Definition of the Magic Constants The key-expansion algorithm uses two word-sized binary constants P_w and Q_w. They are defined for arbitrary w as follows:

$$P_w = \mathrm{Odd}((e-2)2^w) \quad (1)$$

$$Q_w = \mathrm{Odd}((\phi-1)2^w) \quad (2)$$

where

$$e = 2.718281828459\ldots \text{ (base of natural logarithms)}$$

$$\phi = 1.618033988749\ldots \text{ (golden ratio) },$$

and where $\mathrm{Odd}(x)$ is the odd integer nearest to x (rounded up if x is an even integer, although this won't happen here). For $w = 16$, 32, and 64, these constants are given below in binary and in hexadecimal.

```
P16 = 1011011111100001 = b7e1
Q16 = 1001111000110111 = 9e37

P32 = 10110111111000010101000101100011 = b7e15163
Q32 = 10011110001101110111100110111001 = 9e3779b9

P64 = 1011011111100001010100010110001010001010111011010010101001101011
    = b7e151628aed2a6b
Q64 = 1001111000110111011110011011100101111111010010100111110000010101
    = 9e3779b97f4a7c15
```

Converting the Secret Key from Bytes to Words. The first algorithmic step of key expansion is to copy the secret key $K[0 \ldots b-1]$ into an array $L[0 \ldots c-1]$ of $c = \lceil b/u \rceil$ words, where $u = w/8$ is the number of bytes/word. This operation is done in a natural manner, using u consecutive key bytes of K to fill up each successive word in L, low-order byte to high-order byte. Any unfilled byte positions of L are zeroed.

On "little-endian" machines such as an Intel '486, the above task can be accomplished merely by zeroing the array L, and then copying the string K directly into the memory positions representing L. The following pseudo-code achieves the same effect, assuming that all bytes are "unsigned" and that array L is initially zeroed.

> **for** $i = b - 1$ **downto** 0 **do**
> $\quad L[i/u] = (L[i/u] \lll 8) + K[i];$

Initializing the Array S. The second algorithmic step of key expansion is to initialize array S to a particular fixed (key-independent) pseudo-random bit pattern, using an arithmetic progression modulo 2^w determined by the "magic constants" P_w and Q_w. Since Q_w is odd, the arithmetic progression has period 2^w.

> $S[0] = P_w;$
> **for** $i = 1$ **to** $t - 1$ **do**
> $\quad S[i] = S[i-1] + Q_w;$

Mixing in the Secret Key. The third algorithmic step of key expansion is to mix in the user's secret key in three passes over the arrays S and L. More precisely, due to the potentially different sizes of S and L, the larger array will be processed three times, and the other may be handled more times.

$$
\begin{array}{l}
i = j = 0; \\
A = B = 0; \\
\textbf{do } 3 * \max(t, c) \textbf{ times:} \\
\qquad A = S[i] = (S[i] + A + B) \lll 3; \\
\qquad B = L[j] = (L[j] + A + B) \lll (A + B); \\
\qquad i = (i + 1) \bmod(t); \\
\qquad j = (j + 1) \bmod(c);
\end{array}
$$

The key-expansion function has a certain amount of "one-wayness": it is not so easy to determine K from S.

5 Discussion

A distinguishing feature of RC5 is its heavy use of *data-dependent rotations*—the amount of rotation performed is dependent on the input data, and is not predetermined.

The encryption/decryption routines are very simple. While other operations (such as substitution operations) could have been included in the basic round operations, our objective is to focus on the data-dependent rotations as a source of cryptographic strength.

Some of the expanded key table S is initially added to the plaintext, and each round ends by adding expanded key from S to the intermediate values just computed. This assures that each round acts in a potentially different manner, in terms of the rotation amounts used.

The xor operations back and forth between A and B provide some avalanche properties, causing a single-bit change in an input block to cause multiple-bit changes in following rounds.

6 Implementation

The encryption algorithm is very compact, and can be coded efficiently in assembly language on most processors. The table S is both quite small and accessed sequentially, minimizing issues of cache size.

A reference implementation of RC5-32/12/16, together with some sample input/output pairs, is provided in the Appendix.

This reference implementation, which is not optimized for high speed at all, encrypts at 100K bytes/second on a 50Mhz '486 laptop compiled with the Borland C++ compiler in 16-bit mode, and at 2.4M bytes/second on a Sparc 5 when compiled by gcc.

The fastest RC5 encryption speeds obtainable are yet to be fully determined. The above timing results can presumably be improved substantially. Assembly language can help greatly by making the rotation operator directly accessible. For example, we know that an assembly-language routine for the '486 can perform each half-round with just four instructions; an initial assembly-language implementation runs at 1.2M bytes/sec on a 50MHz '486 SLC. One should be able to encrypt at several megabytes/second on a Pentium.

7 Analysis

This section contains some preliminary results on the strength of RC5. Much more work remains to be done. Here we report the results of two experiments studying how changing the number of rounds affects properties of RC5.

The first test involved examining the uniformity of correlation between input and output bits. We found that four rounds sufficed to get very uniform correlations between individual input and output bits in RC5-32.

The second test checked to see if the data-dependent rotation amounts depended on every plaintext bit, in 100 million trials with random plaintext and keys. That is, we checked whether flipping a plaintext bit caused some intermediate rotation to be a rotation by a different amount. We found that eight rounds in RC5-32 were sufficient to cause each message bit to affect some rotation amount.

The number of rounds chosen in practice should always be at least as great (if not substantially greater) than these simple tests would suggest. As noted above, we suggest 12 rounds as a "nominal" choice for RC5-32, and 16 rounds for RC5-64.

RC5's data-dependent rotations may help frustrate differential cryptanalysis (Biham/Shamir [1]) and linear cryptanalysis (Matsui [2]), since bits are rotated to "random" positions in each round.

There is no obvious way in which an RC5 key can be "weak," other than by being too short.

I invite the reader to help determine the strength of RC5.

8 Acknowledgements

I'd like to thank Burt Kaliski, Yiqun Lisa Yin, Paul Kocher, and everyone else at RSA Laboratories for their comments and constructive criticisms.

References

1. E. Biham and A. Shamir. *A Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, 1993.
2. Mitsuru Matsui. The first experimental cryptanalysis of the Data Encryption Standard. In Yvo G. Desmedt, editor, *Proceedings CRYPTO 94*, pages 1–11. Springer, 1994. Lecture Notes in Computer Science No. 839.

9 Appendix

```
/* RC5REF.C -- Reference implementation of RC5-32/12/16 in C.        */
/* Copyright (C) 1995 RSA Data Security, Inc.                        */
#include <stdio.h>
typedef unsigned long int WORD; /* Should be 32-bit = 4 bytes        */
#define w        32             /* word size in bits                 */
#define r        12             /* number of rounds                  */
#define b        16             /* number of bytes in key            */
#define c         4             /* number  words in key = ceil(8*b/w)*/
#define t        26             /* size of table S = 2*(r+1) words   */
WORD S[t];                      /* expanded key table                */
WORD P = 0xb7e15163, Q = 0x9e3779b9;  /* magic constants            */
/* Rotation operators. x must be unsigned, to get logical right shift*/
#define ROTL(x,y) (((x)<<(y&(w-1))) | ((x)>>(w-(y&(w-1)))))
#define ROTR(x,y) (((x)>>(y&(w-1))) | ((x)<<(w-(y&(w-1)))))

void RC5_ENCRYPT(WORD *pt, WORD *ct) /* 2 WORD input pt/output ct    */
{ WORD i, A=pt[0]+S[0], B=pt[1]+S[1];
  for (i=1; i<=r; i++)
    { A = ROTL(A^B,B)+S[2*i];
      B = ROTL(B^A,A)+S[2*i+1];
    }
  ct[0] = A; ct[1] = B;
}

void RC5_DECRYPT(WORD *ct, WORD *pt) /* 2 WORD input ct/output pt    */
{ WORD i, B=ct[1], A=ct[0];
  for (i=r; i>0; i--)
    { B = ROTR(B-S[2*i+1],A)^A;
      A = ROTR(A-S[2*i],B)^B;
    }
  pt[1] = B-S[1]; pt[0] = A-S[0];
}

void RC5_SETUP(unsigned char *K) /* secret input key K[0...b-1]      */
{  WORD i, j, k, u=w/8, A, B, L[c];
   /* Initialize L, then S, then mix key into S */
   for (i=b-1,L[c-1]=0; i!=-1; i--) L[i/u] = (L[i/u]<<8)+K[i];
   for (S[0]=P,i=1; i<t; i++) S[i] = S[i-1]+Q;
   for (A=B=i=j=k=0; k<3*t; k++,i=(i+1)%t,j=(j+1)%c)   /* 3*t > 3*c */
     { A = S[i] = ROTL(S[i]+(A+B),3);
       B = L[j] = ROTL(L[j]+(A+B),(A+B));
     }
}
```

```
void main()
{ WORD i, j, pt1[2], pt2[2], ct[2] = {0,0};
  unsigned char key[b];
  if (sizeof(WORD)!=4)
    printf("RC5 error: WORD has %d bytes.\n",sizeof(WORD));
  printf("RC5-32/12/16 examples:\n");
  for (i=1;i<6;i++)
    { /* Initialize pt1 and key pseudorandomly based on previous ct */
      pt1[0]=ct[0]; pt1[1]=ct[1];
      for (j=0;j<b;j++) key[j] = ct[0]%(255-j);
      /* Setup, encrypt, and decrypt */
      RC5_SETUP(key);
      RC5_ENCRYPT(pt1,ct);
      RC5_DECRYPT(ct,pt2);
      /* Print out results, checking for decryption failure */
      printf("\n%d. key = ",i);
      for (j=0; j<b; j++) printf("%.2X ",key[j]);
      printf("\n   plaintext %.8lX %.8lX  --->  ciphertext %.8lX %.8lX  \n",
             pt1[0], pt1[1], ct[0], ct[1]);
      if (pt1[0]!=pt2[0] || pt1[1]!=pt2[1])
        printf("Decryption Error!");
    }
}
```

```
-------------------------------------------------------------------

RC5-32/12/16 examples:

1. key = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   plaintext 00000000 00000000  --->  ciphertext EEDBA521 6D8F4B15

2. key = 91 5F 46 19 BE 41 B2 51 63 55 A5 01 10 A9 CE 91
   plaintext EEDBA521 6D8F4B15  --->  ciphertext AC13C0F7 52892B5B

3. key = 78 33 48 E7 5A EB 0F 2F D7 B1 69 BB 8D C1 67 87
   plaintext AC13C0F7 52892B5B  --->  ciphertext B7B3422F 92FC6903

4. key = DC 49 DB 13 75 A5 58 4F 64 85 B4 13 B5 F1 2B AF
   plaintext B7B3422F 92FC6903  --->  ciphertext B278C165 CC97D184

5. key = 52 69 F1 49 D4 1B A0 15 24 97 57 4D 7F 15 31 25
   plaintext B278C165 CC97D184  --->  ciphertext 15E444EB 249831DA
```

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