Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: October 13, 2018

T. Krovetz
Sacramento State

April 11, 2018

RC6 and RC5 Test Vectors For Multiple Block Sizes
draft-krovetz-rc6-rc5-vectors-00

Abstract

The RC6 and RC5 block ciphers are parameterized, allowing a variety of block sizes, key sizes, and security levels. This flexibility, along with simple implementations, make RC6 and RC5 attractive for many applications. This document supplies test vectors to aid in the development of compatible ciphers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 13, 2018.

Table of Contents

1. Introduction

The RC5 block cipher was developed by Rivest [RC5], described in RFC 2040 [RFC2040], and has features that were patented in the United States by RSA Data Security [RC5pat]. The RC6 block cipher was developed by Rivest, Robshaw, Sidney, and Yin [RC6], was a finalist in NIST's AES competition [NIST], and has features that were patented in the United States by RSA Security [RC6pat]. These patents are now expired, making the RC6 and RC5 algorithms available to be used freely. The names "RC6" and "RC5" remain trade marks.

Both block ciphers are parameterized, allowing the specification of block size, key size, and number of internal iterations of the cipher's round function. RC6-w/r/b has a block size of 4w bits, a key size of 8b bits, and executes r rounds internally. RC5-w/r/b has a block size of 2w bits, a key size of 8b bits, and executes r rounds internally. The block ciphers are efficient when w-bit operations are well supported.

The RC6 and RC5 specifications both restrict r and b to the values 0 through 255. And although the RC5 specification explicitly restricts w to 16, 32, or 64, both RC6 and RC5 are well defined for any power of two w of at least eight. This means that both ciphers can support many security levels, key lengths, and block sizes. This document provides test vectors for w values 8, 16, 32, 64, and 128 -- the values for which simple efficient code can be written in C -- allowing block ciphers with block sizes as small as 16 bits (RC5, w=8) and as large as 512 bits (RC6, w=128).

RC6 and RC5 are most efficient and secure when w is a power of two. A non-standard modification to their definitions, however, allows for w to be any positive multiple of eight. When the base-two logarithm of w (ie, lg w) is not an integer, the definitions of RC6 and RC5 are

ambiguous. By interpreting all occurrences of lg w in their specifications as floor(lg w), both ciphers become well defined for all w values that are a positive multiple of eight (and when w is also a power of two, compatibility is maintained with the standard definitions of RC6 and RC5). Hence, if d = floor(lg w), then rotations specified to be of distance lg w become rotations of distance d, and rotations specified to be a distance equal to the least significant lg w bits of a quantity become rotations equal to the least significant d bits of that quantity. This modification introduces biases to data-dependent rotation distances when w is not a power of two that an attacker may be able to exploit. This possibility can be mitigated by increasing the number of rounds used. This document provides a small number test vectors for w not equal to a power of two, but marks them as "non-standard".

2. Security Considerations

The original RC5 publication suggested using 12 rounds when w=32 and 16 rounds when w=64. In response to cryptanalysis, the authors changed the recommendation when w=32 to 16 rounds [RC5sec]. NIST concluded in 2001 that RC6 with w=32 and r=20 "appears to have an adequate security margin" [NIST] which agrees with the RC6 inventor's analysis [RC6sec]. No other recommendations have been published regarding the number of rounds needed for good security with other w values.

From these recommendations, a reasonable rule of thumb for the selection of rounds can be interpolated: RC6 needs four more rounds than RC5 for the same w. Each doubling of w should increase the number of rounds by four. This rule yields the following minimum number of rounds for assorted w.

w	RC5	RC6
8	8	12
16	12	16
32	16	20
64	20	24
128	24	28
256	28	32

When w is not a power of two, the data-dependent rotations of RC6 and RC5 are not of approximately random distance. Attacks based on these biases have not been studied but likely allow for more effective attacks. It is best to avoid using w that is not a power of two, but if such a w is chosen the number of rounds used should be increased

significantly to mitigate the effectiveness of attacks. One possible choice for the number of rounds to use is to round the desired w up to the nearest power of two and add four to the specified number of rounds. For example, if RC5 with w=96 is desired, then 28 rounds would be suggested because 28 is four more than what is specified for RC5 when w=128. Further cryptanalysis is necessary to know whether these recommendations are appropriate.

Although RC6 and RC5 are specified in such a way that small block sizes are possible, it is generally a bad idea to use them. Most block cipher usages have attacks with an effectiveness following the birthday bound. Meaning that after n block encryptions with the same key, the probability of a successful attack may be proportional to (n^2)/(2^2w) for RC5 and (n^2)/(2^4w) for RC6. The easiest way to avoid such attacks is to use a large enough w to make the success probability negligible (eg, w at least 64 for RC5 and w at least 32 for RC6).

The RC6 and RC5 key schedule algorithms are considered to be strong. The key lengths used therefore need only be long enough to thwart brute force key attacks. Key lengths of at least 16 bytes are recommended. Unlike some other ciphers, longer keys have no negative effect on cipher performance. The test vectors provided in this document use a variety of key lengths to assist in validating implementations, they are not necessarily recommended key lengths.

3. RC6 Test Vectors

This section contains hexadecimal representations of key and block inputs and the corresponding block outputs for RC6 with various word sizes, numbers of rounds and key bytes. After a number of these, lengthier examples are given, showing every assignment to arrays L and S (following the algorithm in section "Key schedule for RC6" of [RC6]) and every assignment to variables A, B, C and D during block encipherment (following the algorithm in section "2.2 Encryption and decryption" of [RC6]). All of the examples in this section were generated by an ANSI C program given in the appendix. The program can easily be adapted for other w/r/b of interest.

```
RC6-8/12/4
Key:          00010203
Block input:  00010203
Block output: AEFC4612

RC6-16/16/8
Key:          0001020304050607
Block input:  0001020304050607
Block output: 2FF0B68EAEFFAD5B
```

```
   RC6-32/20/16
   Key:          000102030405060708090A0B0C0D0E0F
   Block input:  000102030405060708090A0B0C0D0E0F
   Block output: 3A96F9C7F6755CFE46F00E3DCD5D2A3C

   RC6-64/24/24
   Key:          000102030405060708090A0B0C0D0E0F1011121314151617
   Block input:  000102030405060708090A0B0C0D0E0F
                 101112131415161718191A1B1C1D1E1F
   Block output: C002DE050BD55E5D36864AB9853338E6
                 DC4A1326C6BDAAEB1BC9E4FD67886617

   RC6-128/28/32
   Key:          000102030405060708090A0B0C0D0E0F
                 101112131415161718191A1B1C1D1E1F
   Block input:  000102030405060708090A0B0C0D0E0F
                 101112131415161718191A1B1C1D1E1F
                 202122232425262728292A2B2C2D2E2F
                 303132333435363738393A3B3C3D3E3F
   Block output: 4ED87C64BAFFECD4303EE6A79AAFAEF5
                 75B351C024272BE70A70B4A392CFC157
                 DBA52D529A79E83845BF43D67545383A
                 ED3DBF4F0D23640E44CBF6CDAA034DCB

   RC6-24/4/0 (non-standard, w not power of two)
   Key:
   Block input:  000102030405060708090A0B
   L[  0] = 000000
   S[  0] = B7E151
   S[  1] = 5618CA
   S[  2] = F45043
   S[  3] = 9287BC
   S[  4] = 30BF35
   S[  5] = CEF6AE
   S[  6] = 6D2E27
   S[  7] = 0B65A0
   S[  8] = A99D19
   S[  9] = 47D492
   S[ 10] = E60C0B
   S[ 11] = 844384
   S[  0] = BF0A8D
   L[  0] = 51B7E1
   S[  1] = 36D9C3
   L[  0] = A4985D
   S[  2] = 7E131E
   L[  0] = 1EC63A
   S[  3] = 7B08A1
   L[  0] = A8ADC4
```

```
   S[  4] = A3ACD2
   L[  0] = 4216BD
   S[  5] = A5D1ED
   L[  0] = FD9CA7
   S[  6] = 84E5D8
   L[  0] = 93400F
   S[  7] = 1C5C39
   L[  0] = DC5742
   S[  8] = 1284A5
   L[  0] = 9994E5
   S[  9] = 9F70E7
   L[  0] = AB1D29
   S[ 10] = 84D0D9
   L[  0] = 6C2CAF
   S[ 11] = AA0863
   L[  0] = 098706
   S[  0] = 94CFB3
   L[  0] = BB7F4F
   S[  1] = 39462C
   L[  0] = 265582
   S[  2] = ED7666
   L[  0] = 216A3A
   S[  3] = 4F4A0C
   L[  0] = 87A024
   S[  4] = D4B813
   L[  0] = FC2DF1
   S[  5] = B5BF8B
   L[  0] = B6DAE1
   S[  6] = 8C0227
   L[  0] = B7E9F9
   S[  7] = 0242CB
   L[  0] = 216BD7
   S[  8] = B19A39
   L[  0] = F471E7
   S[  9] = 2BE83A
   L[  0] = 299810
   S[ 10] = D2891E
   L[  0] = 4F896E
   S[ 11] = 60D77E
   L[  0] = A5AFFE
   S[  0] = DAB97C
   L[  0] = 65E098
   S[  1] = CF0203
   L[  0] = 199CD6
   S[  2] = B0A9FE
   L[  0] = 3E3AAE
   S[  3] = F175C1
   L[  0] = 8EB6F5
```

```
   S[  4] = A7264A
   L[  0] = 1A624A
   S[  5] = BA40FB
   L[  0] = E0B1FD
   S[  6] = 37A8F9
   L[  0] = 433CFE
   S[  7] = E94613
   L[  0] = DF801E
   S[  8] = D30353
   L[  0] = 24071F
   S[  9] = 179561
   L[  0] = 5FA39F
   S[ 10] = 4E10F2
   L[  0] = 1AB060
   S[ 11] = 4CC686
   L[  0] = 89D1A0
   B = DFBD7F
   D = DA0C0C
   A = D484CD
   C = 6468F6
   A = 417148
   C = 790376
   A = E41CA8
   C = 4A660F
   A = B96932
   C = BE7925
   A = 987701
   C = 30E32E
   Block output: 0177982579BE2EE3303269B9

   RC6-80/4/12 (non-standard, w not power of two)
   Key:          000102030405060708090A0B
   Block input:  000102030405060708090A0B0C0D0E0F
                 101112131415161718191A1B1C1D1E1F
                 2021222324252627
   L[  0] = 090807060504030201 00
   L[  1] = 0000000000000000B0A
   S[  0] = B7E151628AED2A6ABF71
   S[  1] = 5618CB1C0A37A680B30E
   S[  2] = F45044D589822296A6AB
   S[  3] = 9287BE8F08CC9EAC9A48
   S[  4] = 30BF384888171AC28DE5
   S[  5] = CEF6B202076196D88182
   S[  6] = 6D2E2BBB86AC12EE751F
   S[  7] = 0B65A57505F68F0468BC
   S[  8] = A99D1F2E85410B1A5C59
   S[  9] = 47D498E8048B87304FF6
   S[ 10] = E60C12A183D603464393
```

```
   S[ 11] = 84438C5B03207F5C3730
   S[  0] = BF0A8B1457695355FB8D
   L[  0] = 52434B8DAACAFF91B902
   S[  1] = 3B350DF0635FCB433CEB
   L[  1] = 595AA01EF1AF0B2FC1C5
   S[  2] = 46FF9726F487C84D2ADC
   L[  0] = 2203A61D4B47E53B05A7
   S[  3] = DC57DE9A44E261A6565F
   L[  1] = ED8935A07654844772D5
   S[  4] = D502641A1A700582B8CF
   L[  0] = C0C6F05314BE48F3FD7D
   S[  5] = 26003379B47F2A79BE73
   L[  1] = F7B52EC5D450596D3F91
   S[  6] = 571C6FD87BDCB6AB991C
   L[  0] = 6B219AC541F311DE2C9D
   S[  7] = 6D1D80961E32BC7173AE
   L[  1] = A25109A3B13DE6FEE67F
   S[  8] = C85D4B42A58D7455B435
   L[  0] = D32C751D5CFEFAB98BE6
   S[  9] = 1AF2CA4038BFB1FC808F
   L[  1] = 769E5E920E092028DF92
   S[ 10] = BCE9DB9E54F6AB5D1DA3
   L[  0] = C7F12360D695E9B7FFD8
   S[ 11] = 48F45AD17568A38AA558
   L[  1] = AD6B84C28783DCC45A07
   S[  0] = AB535542A2AE9D27D765
   L[  0] = 863A4314420AFFD6600C
   S[  1] = 6615323A40CB420BA2E3
   L[  1] = 0F532E7B4CDD7D08852D
   S[  2] = E33FBEE411843B0A9765
   L[  0] = C1CE81B2DFA5F279E334
   S[  3] = 0B30F989B064795687C4
   L[  1] = D8F025DC52A9B7DCE7E8
   S[  4] = C91C1C00EBF1B5B143DD
   L[  0] = 7B587203C82C0101DF2C
   S[  5] = 53A60BF344E709670BE3
   L[  1] = 51E9AFDE6122E97BD3F7
   S[  6] = E5615D510F354C73C7B7
   L[  0] = 0DBC5EB6ACA8DFCCCE21
   S[  7] = 01D9E4EED08745904C33
   L[  1] = 383DE530ED8EE4B617FF
   S[  8] = 13A8AB131D1CF4E0C338
   L[  0] = B1D4AC2CD1777D5BAA5C
   S[  9] = 03810C013AA121C7711F
   L[  1] = C99BD76C9CEAF7CD3C1E
   S[ 10] = 5035F8616416278E5704
   L[  0] = 49E272DCF5FB2E99EFEB
   S[ 11] = 1866307E7BCFCD97623F
```

```
L[  1] = D7CFFA3920AF91EB203A
S[  0] = DC4BFFD1F96FE552CEF4
L[  0] = A975F7C67F7F9B3A0406
S[  1] = 5EB94E95CDD614C3AEEF
L[  1] = 3D1A65FBFFE812ADC0A8
S[  2] = F89B9BAEFA1313E037E3
L[  0] = 5FCB8BCBD60E3FE48EF9
S[  3] = 1CC10824042E68DA7503
L[  1] = CC4A4B9A6F9EBDA24BB6
S[  4] = 913B7DFAFDF6E17024B5
L[  0] = 1EF7B7FB25EA8AAB0A1D
S[  5] = 1ECA0F4B4643AC11D5A8
L[  1] = 41825C1B799E8BE56F61
S[  6] = 2D6E45BE78BC23586602
L[  0] = C229CF46FC046F42CEA8
S[  7] = 8B8FCFA22A3EC15C06EF
L[  1] = 824FF0DE42227C479DFD
S[  8] = 0C435C9C4BF194234121
L[  0] = 62861FEB6B71942F4730
S[  9] = 92544447902250CFCB83
L[  1] = 0A358583B952A889EDB3
S[ 10] = 65FE11656C59074081D7
L[  0] = E6DB5244750FE6DAEB4A
S[ 11] = 29FCA142E9C5DD967B03
L[  1] = AF2163050D9F6A800361
B = EF5E10E2087DF25ED9FE
D = 85DF73B9F0F835E3CE0D
A = A7CD39B85814151A5603
C = D52CE793FC872209CD0C
A = 90A4275C5F537160193F
C = 63803F113CB741C148CF
A = 52EC2FD337B66443FA35
C = D2EE5C6A952D0B96574F
A = 6279E1B4001B31499A91
C = 54735471FFC0F101D417
A = 38EC6DD0018612D6D926
C = 7CE8D116217C41DA7538
Block output: 26D9D6128601D06DEC3817D401F1C0FF
              715473543875DA417C2116D1E87C919A
              49311B00B4E17962
```

4. RC5 Test Vectors

This section contains hexadecimal representations of key and block inputs and the corresponding block outputs for RC5 with various word sizes, numbers of rounds and key bytes. After a number of these, lengthier examples are given, showing every assignment to arrays L and S (following the algorithm in section "4.3 Key Expansion" of

[RC5]) and every assignment to variables A and B during block encipherment (following the algorithm in section "4.1 Encryption" of [RC5]). All of the examples in this section were generated by an ANSI C program given in the appendix. The program can easily be adapted for other w/r/b of interest.

```
RC5-8/12/4
Key:          00010203
Block input:  0001
Block output: 212A

RC5-16/16/8
Key:          0001020304050607
Block input:  00010203
Block output: 23A8D72E

RC5-32/20/16
Key:          000102030405060708090A0B0C0D0E0F
Block input:  0001020304050607
Block output: 2A0EDC0E9431FF73

RC5-64/24/24
Key:          000102030405060708090A0B0C0D0E0F1011121314151617
Block input:  000102030405060708090A0B0C0D0E0F
Block output: A46772820EDBCE0235ABEA32AE7178DA

RC5-128/28/32
Key:          000102030405060708090A0B0C0D0E0F
              101112131415161718191A1B1C1D1E1F
Block input:  000102030405060708090A0B0C0D0E0F
              101112131415161718191A1B1C1D1E1F
Block output: ECA5910921A4F4CFDD7AD7AD20A1FCBA
              068EC7A7CD752D68FE914B7FE180B440

RC5-24/4/0 (non-standard, w not power of two)
Key:
Block input:  000102030405
L[  0] = 000000
S[  0] = B7E151
S[  1] = 5618CA
S[  2] = F45043
S[  3] = 9287BC
S[  4] = 30BF35
S[  5] = CEF6AE
S[  6] = 6D2E27
S[  7] = 0B65A0
S[  8] = A99D19
S[  9] = 47D492
```

```
   S[  0] = BF0A8D
   L[  0] = 51B7E1
   S[  1] = 36D9C3
   L[  0] = A4985D
   S[  2] = 7E131E
   L[  0] = 1EC63A
   S[  3] = 7B08A1
   L[  0] = A8ADC4
   S[  4] = A3ACD2
   L[  0] = 4216BD
   S[  5] = A5D1ED
   L[  0] = FD9CA7
   S[  6] = 84E5D8
   L[  0] = 93400F
   S[  7] = 1C5C39
   L[  0] = DC5742
   S[  8] = 1284A5
   L[  0] = 9994E5
   S[  9] = 9F70E7
   L[  0] = AB1D29
   S[  0] = 4CC4E8
   L[  0] = 45FE75
   S[  1] = 4CE906
   L[  0] = 2F86C7
   S[  2] = D4175F
   L[  0] = C93B4C
   S[  3] = C2DA60
   L[  0] = 0F8555
   S[  4] = B0643B
   L[  0] = CF6EE5
   S[  5] = 2D2869
   L[  0] = 8CF301
   S[  6] = F80A11
   L[  0] = 47C04C
   S[  7] = E134B2
   L[  0] = 529C2D
   S[  8] = 32AC22
   L[  0] = 3E6BF2
   S[  9] = 8447D8
   L[  0] = 7EF004
   S[  0] = 7FE622
   L[  0] = 718A9F
   S[  1] = F2CE39
   L[  0] = E377D5
   S[  2] = 52EB6D
   L[  0] = 676C5C
   S[  3] = E9914B
   L[  0] = 3501DC
```

```
S[  4] = 77BB16
L[  0] = 86FB3B
S[  5] = 5EF5D1
L[  0] = C476CE
S[  6] = DBB580
L[  0] = C71928
S[  7] = 201AD4
L[  0] = D24AE4
S[  8] = 288ED1
L[  0] = A49339
S[  9] = 8B4F12
L[  0] = AC26A3
A = 81E722
B = F7D23C
A = A4D2D0
B = 3C9237
A = 982EE2
B = F1E927
A = BF9834
B = 072C08
A = DCCB89
B = 5A52C9
Block output: 89CBDCC9525A

RC5-80/4/12 (non-standard, w not power of two)
Key:          000102030405060708090A0B
Block input:  000102030405060708090A0B0C0D0E0F10111213
L[  0] = 09080706050403020100
L[  1] = 0000000000000000B0A
S[  0] = B7E151628AED2A6ABF71
S[  1] = 5618CB1C0A37A680B30E
S[  2] = F45044D589822296A6AB
S[  3] = 9287BE8F08CC9EAC9A48
S[  4] = 30BF384888171AC28DE5
S[  5] = CEF6B202076196D88182
S[  6] = 6D2E2BBB86AC12EE751F
S[  7] = 0B65A57505F68F0468BC
S[  8] = A99D1F2E85410B1A5C59
S[  9] = 47D498E8048B87304FF6
S[  0] = BF0A8B1457695355FB8D
L[  0] = 52434B8DAACAFF91B902
S[  1] = 3B350DF0635FCB433CEB
L[  1] = 595AA01EF1AF0B2FC1C5
S[  2] = 46FF9726F487C84D2ADC
L[  0] = 2203A61D4B47E53B05A7
S[  3] = DC57DE9A44E261A6565F
L[  1] = ED8935A07654844772D5
S[  4] = D502641A1A700582B8CF
```

```
   L[  0] = C0C6F05314BE48F3FD7D
   S[  5] = 26003379B47F2A79BE73
   L[  1] = F7B52EC5D450596D3F91
   S[  6] = 571C6FD87BDCB6AB991C
   L[  0] = 6B219AC541F311DE2C9D
   S[  7] = 6D1D80961E32BC7173AE
   L[  1] = A25109A3B13DE6FEE67F
   S[  8] = C85D4B42A58D7455B435
   L[  0] = D32C751D5CFEFAB98BE6
   S[  9] = 1AF2CA4038BFB1FC808F
   L[  1] = 769E5E920E092028DF92
   S[  0] = 84DD9F34F1912BDADD72
   L[  0] = EA872E45C9946BD48EAC
   S[  1] = 54CEDB58F42B1795484D
   L[  1] = 256D176BE8D061979147
   S[  2] = 09DC4F5E8C1A0BD02386
   L[  0] = 12A207CFDB27886F233A
   S[  3] = C6B1AE45611FAF2CE8FF
   L[  1] = 673B01FD819B024A2F32
   S[  4] = 1778A2E7E955B7CE8800
   L[  0] = 0A1F69B24556B2D51861
   S[  5] = 3CC2009F195CA8EAF6A2
   L[  1] = 70E362770272F051F1AD
   S[  6] = 260E9774BD627F440B58
   L[  0] = 222C73C0A5844D62ACD4
   S[  7] = AAC45E5C08CC48C15ED5
   L[  1] = 870CEBFAAC7BA8692761
   S[  8] = D174ACCAD6AB2C01D35F
   L[  0] = 7AAE0C8628AB21CDA794
   S[  9] = 38AC1C89C0AFFE5FDC13
   L[  1] = EB644B55841D338A854A
   S[  0] = 477038A1B2F2EE29F67D
   L[  0] = C1483EAFDDA1C111ADD6
   S[  1] = EC3A955425FE36876502
   L[  1] = 5987BD2B23982298E71F
   S[  2] = 7CF50EEEAD8327837D3A
   L[  0] = 935D7A165C245F2F8A15
   S[  3] = B821BA535639AEFF8276
   L[  1] = 378CA6AFB1863F9D5528
   S[  4] = 39381F5788AD335AFCF0
   L[  0] = 1D9657D227DC2D042240
   S[  5] = 9C83BE464F304A50AE94
   L[  1] = CC82892B6F225FCF1A6B
   S[  6] = 78A6F733DDA94B1EA2BC
   L[  0] = 53EBF8EFB3B15FEC18BA
   S[  7] = BABA73FCD1379E60D25B
   L[  1] = C2FE816BC380B01B651E
   S[  8] = 796D119B5B1BD3F056C2
```

```
L[  0] = D24DE3F7D49A90578BF6
S[  9] = 233890E78333153DF65C
L[  1] = D92C6D3956C39DC2E213
A = 50783FA7B7F6F12BF77D
B = FF4CA664350C4393700C
A = C691471E58AEAFFA982D
B = 15AEF7577D756B2ED02A
A = A84B7077A7FA321C2187
B = 8F474EB396DCE3C98572
A = BFFD8B0879DA462F6757
B = 98B1F74FC4A8B0F92FBD
A = 428A5684EAA4CB9EB59C
B = D619589DFCD532E1B078
Block output: 9CB59ECBA4EA84568A4278B0E132D5FC9D5819D6
```

5. IANA Considerations

This document has no IANA actions.

[RFC Editor: please remove this section prior to publication.]

6. References

6.1. Normative References

[RC5] Rivest, R., "The RC5 encryption algorithm (revised March 20, 1997)", in Fast Software Encryption - FSE 1994, Springer, 1995.

[RC6] Rivest, R., Robshaw, M., Sidney, R., and Y. Yin, "The RC6 block cipher", Posted on RSA Data Security website August 20, 1998.

6.2. Informative References

[NIST] Nechvatal, J., Barker, E., Bassham, L., Burr, W., Dworkin, M., Foti, J., and E. Roback, "Report on the development of the Advanced Encryption Standard (AES)", J. Res. Natl. Inst. Stand. Technol. 106, March 2001.

[RC5pat] Rivest, R., "Block encryption algorithm with data-dependent rotations", U.S. Patent 5,835,600, filed April 21, 1997.

[RC5sec] Kaliski, B. and Y. Yin, "On the security of the RC5 encryption algorithm", RSA Laboratories Technical Report TR-602, September 1998.

[RC6pat] Rivest, R., Robshaw, M., Sidney, R., and Y. Yin, "Enhanced block ciphers with data-dependent rotations", U.S. Patent 6,269,163, filed June 15, 1998.

[RC6sec] Contini, S., Rivest, R., Robshaw, M., and Y. Yin, "The security of the RC6 block cipher", Posted on RSA Data Security website, August 1998.

[RFC2040] Baldwin, R. and R. Rivest, "The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS algorithms", RFC 2040, October 1996.

Appendix A. Test Vector Generator

This section contains the C program used to generate the RC6 test vectors in Section 3. Calls to "print_vector(w,r,b)" can be invoked for any RC6-w/r/b where w is a positive multiple of 8 up to 1024, and both r and b are in the range 0..255 (inclusive). To aid in debugging, the global variable "vectors" may be set to a non-zero value, after which every assignment to the S and L arrays during setup and A/B/C/D variables during encryption are displayed. The C code compiles without emitting any warnings under gcc and g++ 7.3 with flags "-ansi -Wall -Wextra -Wpedantic". Its output in Section 3 was slightly reformatted for presentation.

A.1. C code for RC6 Vector Generation

```
<CODE BEGINS>
/*
// RC6 & RC5 block cipher supporting unusual block sizes. This
// implementation is designed only for testing interoperability.
//
// Written by Ted Krovetz (ted@krovetz.net). Modified April 10, 2018.
//
// RC6 and RC5 were both patented and trademarked around the time
// each was invented. The author of this code believes the patents
// have expired and that the trademarks may still be in force. Seek
// legal advice before using RC5 or RC6 in any project.
//
// This is free and unencumbered software released into the public
// domain.
//
// Anyone is free to copy, modify, publish, use, compile, sell, or
// distribute this software, either in source code form or as a
// compiled binary, for any purpose, commercial or non-commercial,
// and by any means.
//
// In jurisdictions that recognize copyright laws, the author or
// authors of this software dedicate any and all copyright interest
```

```
// in the software to the public domain. We make this dedication for
// the benefit of the public at large and to the detriment of our
// heirs and successors. We intend this dedication to be an overt act
// of relinquishment in perpetuity of all present and future rights
// to this software under copyright law.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
// EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
// OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
// NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY
// CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
// CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
// WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
//
// For more information, please refer to <http://unlicense.org/>
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* set vectors non-zero to print intermediate setup/encrypt values */
static int vectors = 0;

/* pbuf is used to print sequences of bytes from in memory         */
static void pbuf(const void *p, int len, const void *s)
{
    int i;
    if (s) printf("%s", (char *)s);
    for (i=0; i<len; i++) printf("%02X", ((unsigned char *)p)[i]);
    printf("\n");
}

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * C O N S T A N T   D A T A   &   U T I L I T Y   F U N C T I O N S
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* 1024 bits of P_w/Q_w. For any w, grab w bits & set last bit 1.  */
/* WolframAlpha: IntegerPart[(e - 2) * 2^1024] to hex              */
static const unsigned char PP[] = {
    0xb7,0xe1,0x51,0x62,0x8a,0xed,0x2a,0x6a,0xbf,0x71,0x58,0x80,0x9c,
    0xf4,0xf3,0xc7,0x62,0xe7,0x16,0x0f,0x38,0xb4,0xda,0x56,0xa7,0x84,
    0xd9,0x04,0x51,0x90,0xcf,0xef,0x32,0x4e,0x77,0x38,0x92,0x6c,0xfb,
    0xe5,0xf4,0xbf,0x8d,0x8d,0x8c,0x31,0xd7,0x63,0xda,0x06,0xc8,0x0a,
    0xbb,0x11,0x85,0xeb,0x4f,0x7c,0x7b,0x57,0x57,0xf5,0x95,0x84,0x90,
    0xcf,0xd4,0x7d,0x7c,0x19,0xbb,0x42,0x15,0x8d,0x95,0x54,0xf7,0xb4,
    0x6b,0xce,0xd5,0x5c,0x4d,0x79,0xfd,0x5f,0x24,0xd6,0x61,0x3c,0x31,
    0xc3,0x83,0x9a,0x2d,0xdf,0x8a,0x9a,0x27,0x6b,0xcf,0xbf,0xa1,0xc8,
    0x77,0xc5,0x62,0x84,0xda,0xb7,0x9c,0xd4,0xc2,0xb3,0x29,0x3d,0x20,
```

```
    0xe9,0xe5,0xea,0xf0,0x2a,0xc6,0x0a,0xcc,0x93,0xed,0x87};
/* WolframAlpha: IntegerPart[(GoldenRatio - 1) * 2^1024] to hex    */
static const unsigned char QQ[] = {
    0x9e,0x37,0x79,0xb9,0x7f,0x4a,0x7c,0x15,0xf3,0x9c,0xc0,0x60,0x5c,
    0xed,0xc8,0x34,0x10,0x82,0x27,0x6b,0xf3,0xa2,0x72,0x51,0xf8,0x6c,
    0x6a,0x11,0xd0,0xc1,0x8e,0x95,0x27,0x67,0xf0,0xb1,0x53,0xd2,0x7b,
    0x7f,0x03,0x47,0x04,0x5b,0x5b,0xf1,0x82,0x7f,0x01,0x88,0x6f,0x09,
    0x28,0x40,0x30,0x02,0xc1,0xd6,0x4b,0xa4,0x0f,0x33,0x5e,0x36,0xf0,
    0x6a,0xd7,0xae,0x97,0x17,0x87,0x7e,0x85,0x83,0x9d,0x6e,0xff,0xbd,
    0x7d,0xc6,0x64,0xd3,0x25,0xd1,0xc5,0x37,0x16,0x82,0xca,0xdd,0x0c,
    0xcc,0xfd,0xff,0xbb,0xe1,0x62,0x6e,0x33,0xb8,0xd0,0x4b,0x43,0x31,
    0xbb,0xf7,0x3c,0x79,0x0d,0x94,0xf7,0x9d,0x47,0x1c,0x4a,0xb3,0xed,
    0x3d,0x82,0xa5,0xfe,0xc5,0x07,0x70,0x5e,0x4a,0xe6,0xe5};

#define MAXSZ ((int)sizeof(PP)) /* Defines max bytes allowed for W */

/* d[0..n-1] = a[0..n-1] xor b[0..n-1]                             */
static void eor(unsigned char d[], unsigned char a[],
                unsigned char b[], int n) {
    for ( ; n>0; n--) d[n-1] = a[n-1] ^ b[n-1];
}

/* d[0..n-1] = a[0..n-1] + b[0..n-1] (mod 2^8n)                    */
static void add(unsigned char d[], unsigned char a[],
                unsigned char b[], int n) {
    int tmp, carry = 0;
    for ( ; n>0; n--) {
        d[n-1] = tmp = a[n-1] + b[n-1] + carry;
        carry = tmp >> 8;
    }
}

/* d[0..n-1] = a[0..n-1] - b[0..n-1] (mod 2^8n)                    */
static void sub(unsigned char d[], unsigned char a[],
                unsigned char b[], int n) {
    int tmp, borrow = 0;
    for ( ; n>0; n--) {
        d[n-1] = tmp = a[n-1] - b[n-1] - borrow;
        borrow = (tmp < 0 ? 1 : 0);
    }
}

/* d[0..n-1] = a[0..n-1] * b[0..n-1] (mod 2^8n)                    */
static void mul(unsigned char d[], unsigned char a[],
                unsigned char b[], int n) {
    int i,j;
    unsigned char t[MAXSZ] = {0};
    for (i=0; i<n; i++) {
```

```
            int tmp, carry = 0;
            for (j=0; i+j<n; j++) {
                tmp = a[n-i-1] * b[n-j-1] + t[n-i-j-1] + carry;
                t[n-i-j-1] = tmp;
                carry = tmp >> 8;
            }
        }
        memcpy(d,t,n);
    }

    /* d[0..n-1] = a[0..n-1] rotated left r bits                        */
    static void rotl(unsigned char d[], unsigned char a[], int r, int n){
        int i;
        unsigned char t[MAXSZ];
        for (i = 0; i < n; i++)
            t[i] = (a[(i+r/8)%n] << r%8) | (a[(i+r/8+1)%n] >> (8-r%8));
        memcpy(d,t,n);
    }

    /* Calculate floor(base-2 log of x) for any x>0.                    */
    static int lg2(int x) {
        int ans=0;
        for ( ; x!=1; x>>=1)
            ans++;
        return ans;
    }

    /* Return last nbits of a[0..n-1] as int. Pre: 0 <= nbits <= 16.    */
    static int bits(unsigned char a[], int n, int nbits) {
        int mask = ((1 << nbits) - 1);
        if (nbits <= 8) return a[n-1] & mask;
        else            return ((a[n-2] << 8) | a[n-1]) & mask;
    }

    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     * A R C 6   A N D   A R C 5   F U N C T I O N S
     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

    /* Preconditions: 0 < w <=1024, w%8==0, 0 <= r < 256, 0 <= b < 256 */
    static int setup(void *rkey, int rk_words,
                         int w, int r, int b, void *key) {
        if (w<=0 || w>MAXSZ*8 || w%8!=0 || r<0 || r>255 || b<0 || b>255)
            return -1;
        else {
            unsigned char L[256+MAXSZ], Q[MAXSZ];
            unsigned char A[MAXSZ] = {0}, B[MAXSZ] = {0};
            unsigned char *rk = (unsigned char *)rkey;
            int i, mix_steps, n = w/8, lgw = lg2(w);
```

```
        int l_words = (b==0 ? 1 : (b+n-1)/n);
        memcpy(Q, QQ, n); Q[n-1] |= 1;         /* Load Q, make odd */
        /* Initialize rkey with specified P & Q constant values    */
        memcpy(rk, PP, n); rk[n-1] |= 1;       /* Load P, make odd */
        for (i=1; i<rk_words; i++)
            add(rk+i*n, rk+(i-1)*n, Q, n);
        /* Fill L: Zero last word, little-endian copy each word    */
        memset(L+(l_words-1)*n, 0, n);
        for (i=0; i<b; i++)
            L[i/n*n + n-1 - i%n] = ((unsigned char *)key)[i];
        if (vectors) {          /* Print initial values of L and S */
            for (i=0; i<l_words; i++)
                {printf("L[%3d] = ", i); pbuf((char *)L+i*n,n,0);}
            for (i=0; i<rk_words; i++)
                {printf("S[%3d] = ", i); pbuf((char *)rkey+i*n,n,0);}
        }
        /* Mix L and rkey                                          */
        mix_steps = 3 * (rk_words>l_words ? rk_words : l_words);
        for (i=0; i < mix_steps; i++) {
            unsigned rot_amt, ko = i%rk_words*n, lo = i%l_words*n;
            add(A,A,B,n); add(A,A,rk+ko,n); rotl(A,A,3,n);
            memcpy(rk+ko,A,n);
            add(B,B,A,n); rot_amt = bits(B,n,lgw);
            add(B,B,L+lo,n); rotl(B,B,rot_amt,n);
            memcpy(L+lo,B,n);
            if (vectors) {          /* Print new values of L and S */
                printf("S[%3d] = ", ko/n); pbuf(A,n,0);
                printf("L[%3d] = ", lo/n); pbuf(B,n,0);
            }
        }
        return 0;
    }
}
int rc5_setup(void *rkey, int w, int r, int b, void *key) {
    return setup(rkey, 2*r+2, w, r, b, key);
}
int rc6_setup(void *rkey, int w, int r, int b, void *key) {
    return setup(rkey, 2*r+4, w, r, b, key);
}

void rc5_encrypt(void *rkey, int w, int r, void *pt, void *ct) {
    unsigned char A[MAXSZ], B[MAXSZ];
    unsigned char *rk = (unsigned char *)rkey,
                  *p = (unsigned char *)pt,
                  *c = (unsigned char *)ct;
    int rot_amt, i, n = w/8, lgw = lg2(w);
    /* Read A and B in byte-reverse order */
    for (i=0; i<n; i++) { A[i] = p[n-i-1]; B[i] = p[2*n-i-1]; }
```

```
        add(A,A,rk,n);
        add(B,B,rk+n,n);
        if (vectors) { pbuf(A,n,"A = "); pbuf(B,n,"B = "); }
        for (i=1; i<=r; i++) {
            rot_amt = bits(B,n,lgw);
            eor(A,A,B,n); rotl(A,A,rot_amt,n); add(A,A,rk+2*i*n,n);
            rot_amt = bits(A,n,lgw);
            eor(B,B,A,n); rotl(B,B,rot_amt,n); add(B,B,rk+2*i*n+n,n);
            if (vectors) { pbuf(A,n,"A = "); pbuf(B,n,"B = "); }
        }
        /* Write A and B in byte-reverse order */
        for (i=0; i<n; i++) { c[n-i-1] = A[i]; c[2*n-i-1] = B[i]; }
    }

    void rc5_decrypt(void *rkey, int w, int r, void *ct, void *pt) {
        unsigned char A[MAXSZ], B[MAXSZ];
        unsigned char *rk = (unsigned char *)rkey,
                      *p = (unsigned char *)pt,
                      *c = (unsigned char *)ct;
        int rot_amt, i, n = w/8, lgw = lg2(w);
        /* Read A and B in byte-reverse order */
        for (i=0; i<n; i++) { A[i] = c[n-i-1]; B[i] = c[2*n-i-1]; }
        for (i=r; i>0; i--) {
            rot_amt = bits(A,n,lgw);
            sub(B,B,rk+2*i*n+n,n); rotl(B,B,w-rot_amt,n); eor(B,B,A,n);
            rot_amt = bits(B,n,lgw);
            sub(A,A,rk+2*i*n,n); rotl(A,A,w-rot_amt,n); eor(A,A,B,n);
        }
        sub(B,B,rk+n,n);
        sub(A,A,rk,n);
        /* Write A and B in byte-reverse order */
        for (i=0; i<n; i++) { p[n-i-1] = A[i]; p[2*n-i-1] = B[i]; }
    }

    void rc6_encrypt(void *rkey, int w, int r, void *pt, void *ct) {
        unsigned char A[MAXSZ], B[MAXSZ], C[MAXSZ], D[MAXSZ];
        unsigned char t[MAXSZ], u[MAXSZ];
        unsigned char *rk = (unsigned char *)rkey,
                      *p = (unsigned char *)pt,
                      *c = (unsigned char *)ct;
        int rot_amt, i, n = w/8, lgw = lg2(w);
        /* Read A/B/C/D in byte-reverse order */
        for (i=0; i<n; i++) {
            A[i] = p[n-i-1];     B[i] = p[2*n-i-1];
            C[i] = p[3*n-i-1];   D[i] = p[4*n-i-1];
        }
        add(B,B,rk,n); add(D,D,rk+n,n);
        if (vectors) { pbuf(B,n,"B = "); pbuf(D,n,"D = "); }
```

```
        for (i=1; i<=r; i++) {
            rotl(t, B, 1, n); t[n-1] |= 1;       /* t = 2*B+1          */
            rotl(u, D, 1, n); u[n-1] |= 1;       /* u = 2*D+1          */
            mul(t, t, B, n); rotl(t, t, lgw, n); /* t = rotl(B*t, lgw) */
            mul(u, u, D, n); rotl(u, u, lgw, n); /* u = rotl(D*u, lgw) */
            rot_amt = bits(u,n,lgw);
            eor(A,A,t,n); rotl(A,A,rot_amt,n); add(A,A,rk+2*i*n,n);
            rot_amt = bits(t,n,lgw);
            eor(C,C,u,n); rotl(C,C,rot_amt,n); add(C,C,rk+2*i*n+n,n);
            if (vectors) { pbuf(A,n,"A = "); pbuf(C,n,"C = "); }
            memcpy(t,A,n);memcpy(A,B,n);memcpy(B,C,n);
            memcpy(C,D,n);memcpy(D,t,n);
        }
        add(A,A,rk+(2*r+2)*n,n); add(C,C,rk+(2*r+3)*n,n);
        if (vectors) { pbuf(A,n,"A = "); pbuf(C,n,"C = "); }
        /* Write A/B/C/D in byte-reverse order */
        for (i=0; i<n; i++) {
            c[n-i-1] = A[i];     c[2*n-i-1] = B[i];
            c[3*n-i-1] = C[i];   c[4*n-i-1] = D[i];
        }
    }

    void rc6_decrypt(void *rkey, int w, int r, void *ct, void *pt) {
        unsigned char A[MAXSZ], B[MAXSZ], C[MAXSZ], D[MAXSZ];
        unsigned char t[MAXSZ], u[MAXSZ];
        unsigned char *rk = (unsigned char *)rkey,
                      *p = (unsigned char *)pt,
                      *c = (unsigned char *)ct;
        int rot_amt, i, n = w/8, lgw = lg2(w);
        /* Read A/B/C/D in byte-reverse order */
        for (i=0; i<n; i++) {
            A[i] = c[n-i-1];     B[i] = c[2*n-i-1];
            C[i] = c[3*n-i-1];   D[i] = c[4*n-i-1];
        }
        sub(A,A,rk+(2*r+2)*n,n); sub(C,C,rk+(2*r+3)*n,n);
        for (i=r; i>=1; i--) {
            memcpy(t,D,n);memcpy(D,C,n);memcpy(C,B,n);
            memcpy(B,A,n);memcpy(A,t,n);
            rotl(t, B, 1, n); t[n-1] |= 1;       /* t = 2*B+1          */
            rotl(u, D, 1, n); u[n-1] |= 1;       /* u = 2*D+1          */
            mul(t, t, B, n); rotl(t, t, lgw, n); /* t = rotl(B*t, lgw) */
            mul(u, u, D, n); rotl(u, u, lgw, n); /* u = rotl(D*u, lgw) */
            rot_amt = bits(t,n,lgw);
            sub(C,C,rk+2*i*n+n,n); rotl(C,C,w-rot_amt,n); eor(C,C,u,n);
            rot_amt = bits(u,n,lgw);
            sub(A,A,rk+2*i*n,n); rotl(A,A,w-rot_amt,n); eor(A,A,t,n);
        }
        sub(B,B,rk,n); sub(D,D,rk+n,n);
```

```
        /* Write A/B/C/D in byte-reverse order */
        for (i=0; i<n; i++) {
            p[n-i-1] = A[i];     p[2*n-i-1] = B[i];
            p[3*n-i-1] = C[i];   p[4*n-i-1] = D[i];
        }
    }

    static void print_vector(int w, int r, int b) {
        if (w%8!=0 || w<8 || w/8>MAXSZ || r<0 || r>255 || b<0 || b>255) {
            printf("Unsupported w/r/b: %d/%d/%d\n", w, r, b);
        } else {
            int j, bpw=w/8, bpb=4*bpw;    /* bytes per: word and block */
            unsigned char *rkey = (unsigned char *)malloc((2*r+4)*bpw);
            unsigned char *key = (unsigned char *)malloc(b);
            unsigned char *buf = (unsigned char *)malloc(bpb);
            for (j=0; j<b; j++)   key[j]=j;
            for (j=0; j<bpb; j++) buf[j]=j;
            printf("RC6-%d/%d/%d\n",w,r,b);
            pbuf(key, b, "Key:          ");
            pbuf(buf, bpb, "Block input:  ");
            rc6_setup(rkey, w, r, b, key);
            rc6_encrypt(rkey, w, r, buf, buf);
            pbuf(buf, bpb, "Block output: ");
            free(rkey); free(key); free(buf);
        }
    }

    int main() {
        print_vector(8,12,4);    printf("\n");
        print_vector(16,16,8);   printf("\n");
        print_vector(32,20,16);  printf("\n");
        print_vector(64,24,24);  printf("\n");
        print_vector(128,28,32); printf("\n");
        vectors = 1;
        print_vector(24,4,0);    printf("\n");
        print_vector(80,4,12);
        return 0;
    }
    <CODE ENDS>
```

A.2. C code for RC5 Vector Generation

Substituting the following for the print_vector function of the C program will generate test vectors for RC5 instead of RC6.

```
<CODE BEGINS>
static void print_vector(int w, int r, int b) {
    if (w%8!=0 || w<8 || w/8>MAXSZ || r<0 || r>255 || b<0 || b>255) {
        printf("Unsupported w/r/b: %d/%d/%d\n", w, r, b);
    } else {
        int j, bpw=w/8, bpb=2*bpw;    /* bytes per: word and block */
        unsigned char *rkey = (unsigned char *)malloc((2*r+2)*bpw);
        unsigned char *key = (unsigned char *)malloc(b);
        unsigned char *buf = (unsigned char *)malloc(bpb);
        for (j=0; j<b; j++)   key[j]=j;
        for (j=0; j<bpb; j++) buf[j]=j;
        printf("RC5-%d/%d/%d\n",w,r,b);
        pbuf(key, b, "Key:          ");
        pbuf(buf, bpb, "Block input:  ");
        rc5_setup(rkey, w, r, b, key);
        rc5_encrypt(rkey, w, r, buf, buf);
        pbuf(buf, bpb, "Block output: ");
        free(rkey); free(key); free(buf);
    }
}
<CODE ENDS>
```

Author's Address

Ted Krovetz
Computer Science Department
California State University
6000 J Street
Sacramento, CA 95819-6021
USA

Email: ted@krovetz.net