        The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms

Status of this Memo

   This memo provides information for the Internet community. This memo
   does not specify an Internet standard of any kind. Distribution of
   this memo is unlimited.

Table of Contents

1. Executive Summary

   This document defines four ciphers with enough detail to ensure
   interoperability between different implementations. The first cipher
   is the raw RC5 block cipher. The RC5 cipher takes a fixed size input
   block and produces a fixed sized output block using a transformation
   that depends on a key. The second cipher, RC5-CBC, is the Cipher
   Block Chaining (CBC) mode for RC5. It can process messages whose
   length is a multiple of the RC5 block size. The third cipher, RC5-
   CBC-Pad, handles plaintext of any length, though the ciphertext will
   be longer than the plaintext by at most the size of a single RC5

block.   The RC5-CTS cipher is the Cipher Text Stealing mode of RC5,
which handles plaintext of any length and the ciphertext length
matches the plaintext length.

The RC5 cipher was invented by Professor Ronald L. Rivest of the
Massachusetts Institute of Technology in 1994. It is a very fast and
simple algorithm that is parameterized by the block size, the number
of rounds, and key length. These parameters can be adjusted to meet
different goals for security, performance, and exportability.

RSA Data Security Incorporated has filed a patent application on the
RC5 cipher and for trademark protection for RC5, RC5-CBC, RC5-CBC-
Pad, RC5-CTS and assorted variations.

2. Overview

This memo is a restatement of existing published material. The
description of RC5 follows the notation and order of explanation
found in the original RC5 paper by Professor Rivest [2]. The CBC
mode appears in reference works such as the one by Bruce Schneier
[6]. The CBC-Pad mode is the same as in the Public Key Cryptography
Standard (PKCS) number five [5]. Sample C code [8] is included for
clarity only and is equivalent to the English language descriptions.

The ciphers will be explained in a bottom up object-oriented fashion.
First, RC5 keys will be presented along with the key expansion
algorithm. Second, the RC5 block cipher is explained, and finally,
the RC5-CBC and RC5-CBC-Pad ciphers are specified. For brevity, only
the encryption process is described. Decryption is achieved by
inverting the steps of encryption.

The object-oriented description found here should make it easier to
implement interoperable systems, though it is not as terse as the
functional descriptions found in the references. There are two
classes of objects, keys and cipher algorithms. Both classes share
operations that create and destroy these objects in a manner that
ensures that secret information is not returned to the memory
manager.

Keys also have a "set" operation that copies a secret key into the
object. The "set" operation for the cipher objects defines the
number of rounds, and the initialization vector.

There are four operations for the cipher objects described in this
memo. There is binding a key to a cipher object, setting a new
initialization vector for a cipher object without changing the key,
encrypting part of a message (this would be performed multiple times
for long messages), and processing the last part of a message which

may add padding or check the length of the message.

In summary, the cipher will be explained in terms of these
operations:

RC5_Key_Create   - Create a key object.

RC5_Key_Destroy   - Destroy a key object.

RC5_Key_Set   - Bind a user key to a key object.

RC5_CBC_Create   - Create a cipher object.

RC5_CBC_Destroy   - Destroy a cipher object.

RC5_CBC_Encrypt_Init   - Bind a key object to a cipher object.

RC5_CBC_SetIV   - Set a new IV without changing the key.

RC5_CBC_Encrypt_Update - Process part of a message.

RC5_CBC_Encrypt_Final   - Process the end of a message.

3. Terminology and Notation

The term "word" refers to a string of bits of a particular length
that can be operated on as either an unsigned integer or as a bit
vector. For example a "word" might be 32 or 64 bits long depending
on the desired block size for the RC5 cipher. A 32 bit word will
produce a 64 bit block size. For best performance the RC5 word size
should match the register size of the CPU. The term "byte" refers to
eight bits.

The following variables will be used throughout this memo with these
meanings:

W This is the word size for RC5 measured in bits. It is half the
   block size. The word sizes covered by this memo are 32 and 64.

WW This is the word size for RC5 measured in bytes.

B This is the block size for RC5 measured in bits. It is twice
   the word size. When RC5 is used as a 64 bit block cipher, B is
   64 and W is 32. 0 < B < 257. In the sample code, B, is used as
   a variable instead of a cipher system parameter, but this usage
   should be obvious from context.

BB This is the block size for RC5 measured in bytes. BB = B / 8.

   b This is the byte length of the secret key. 0 <= b < 256.

   K This is the secret key which is treated as a sequence of b
      bytes indexed by: K[0], ..., K[b-1].

   R This is the number of rounds of the inner RC5 transform.
      0 <= R < 256.

   T This is the number of words in the expanded key table. It is
      always 2*(R + 1). 1 < T < 513.

   S This is the expanded key table which is treated as a sequence
      of words indexed by: S[0], ..., S[T-1].

   N This is the byte length of the plaintext message.

   P This is the plaintext message which is treated as a sequence of
      N bytes indexed by: P[0], ..., P[N-1].

   C This is the ciphertext output which is treated as a sequence of
      bytes indexed by: C[0], C[1], ...

   I This is the initialization vector for the CBC mode which is
      treated as a sequence of bytes indexed by: I[0], ..., I[BB-1].

4. Description of RC5 Keys

   Like most block ciphers, RC5 expands a small user key into a table of
   internal keys. The byte length of the user key is one of the
   parameters of the cipher, so the RC5 user key object must be able to
   hold variable length keys. A possible structure for this in C is:

   /* Definition of RC5 user key object. */
   typedef struct rc5UserKey
   {
     int   keyLength; /* In Bytes. */
     unsigned char *keyBytes;
   } rc5UserKey;

   The basic operations on a key are to create, destroy and set. To
   avoid exposing key material to other parts of an application, the
   destroy operation zeros the memory allocated for the key before
   releasing it to the memory manager. A general key object may support
   other operations such as generating a new random key and deriving a
   key from key-agreement information.

4.1 Creating an RC5 Key

   To create a key, the memory for the key object must be allocated and
   initialized. The C code below assumes that a function called
   "malloc" will return a block of uninitialized memory from the heap,
   or zero indicating an error.

```
/* Allocate and initialize an RC5 user key.
 * Return 0 if problems.
 */
rc5UserKey *RC5_Key_Create ()
{
  rc5UserKey *pKey;

  pKey = (rc5UserKey *) malloc (sizeof(*pKey));
  if (pKey != ((rc5UserKey *) 0))
  {
      pKey->keyLength = 0;
      pKey->keyBytes = (unsigned char *) 0;
  }
  return (pKey);
}
```

4.2 Destroying an RC5 Key

   To destroy a key, the memory must be zeroed and released to the
   memory manager. The C code below assumes that a function called
   "free" will return a block of memory to the heap.

```
/* Zero and free an RC5 user key.
 */
void RC5_Key_Destroy (pKey)
  rc5UserKey      *pKey;
{
  unsigned char   *to;
  int             count;

  if (pKey == ((rc5UserKey *) 0))
      return;
  if (pKey->keyBytes == ((unsigned char *) 0))
      return;
  to = pKey->keyBytes;
  for (count = 0 ; count < pKey->keyLength ; count++)
      *to++ = (unsigned char) 0;
  free (pKey->keyBytes);
  pKey->keyBytes = (unsigned char *) 0;
  pKey->keyLength = 0;
  free (pKey);
```

```
     }

4.3 Setting an RC5 Key

   Setting the key object makes a copy of the secret key into a block of
   memory allocated from the heap.

  /* Set the value of an RC5 user key.
   * Copy the key bytes so the caller can zero and
   * free the original.
   * Return zero if problems
   */
  int RC5_Key_Set (pKey, keyLength, keyBytes)
    rc5UserKey  *pKey;
    int         keyLength;
    unsigned char   *keyBytes;
  {
    unsigned char   *keyBytesCopy;
    unsigned char   *from, *to;
    int         count;

    keyBytesCopy = (unsigned char *) malloc (keyLength);
    if (keyBytesCopy == ((unsigned char *) 0))
        return (0);
    from = keyBytes;
    to = keyBytesCopy;
    for (count = 0 ; count < keyLength ; count++)
        *to++ = *from++;
    pKey->keyLength = count;
    pKey->keyBytes = keyBytesCopy;
    return (1);
  }
```

5. Description of RC5 Key Expansion

   This section describes the key expansion algorithm. To be specific,
   the sample code assumes that the block size is 64 bits. Several
   programming parameters depend on the block size.

```
  /* Definitions for RC5 as a 64 bit block cipher. */
  /* The "unsigned int" will be 32 bits on all but */
  /* the oldest compilers, which will make it 16 bits. */
  /* On a DEC Alpha "unsigned long" is 64 bits, not 32. */
  #define RC5_WORD     unsigned int
  #define W           (32)
  #define WW          (W / 8)
  #define ROT_MASK    (W - 1)
  #define BB          ((2 * W) / 8) /* Bytes per block */
```

```
   /* Define macros used in multiple procedures. */
   /* These macros assumes ">>" is an unsigned operation, */
   /* and that x and s are of type RC5_WORD. */
   #define SHL(x,s)    ((RC5_WORD)((x)<<((s)&ROT_MASK)))
   #define SHR(x,s,w)  ((RC5_WORD)((x)>>((w)-((s)&ROT_MASK))))
   #define ROTL(x,s,w) ((RC5_WORD)(SHL((x),(s))|SHR((x),(s),(w))))
```

5.1 Definition of initialization constants

   Two constants, Pw and Qw, are defined for any word size W by the
   expressions:

$$Pw = Odd((e-2)*2**W)$$

$$Qw = Odd((phi-1)*2**W)$$

   where e is the base of the natural logarithm (2.71828 ...), and phi
   is the golden ratio (1.61803 ...), and 2**W is 2 raised to the power
   of W, and Odd(x) is equal to x if x is odd, or equal to x plus one if
   x is even. For W equal to 16, 32, and 64, the Pw and Qw constants
   are the following hexadecimal values:

```
   #define P16  0xb7e1
   #define Q16  0x9e37
   #define P32  0xb7e15163
   #define Q32  0x9e3779b9
   #define P64  0xb7e151628aed2a6b
   #define Q64  0x9e3779b97f4a7c15
   #if W == 16
   #define Pw   P16 /* Select 16 bit word size */
   #define Qw   Q16
   #endif
   #if W == 32
   #define Pw   P32 /* Select 32 bit word size */
   #define Qw   Q32
   #endif
   #if W == 64
   #define Pw   P64 /* Select 64 bit word size */
   #define Qw   Q64
   #endif
```

5.2 Interface definition

   The key expansion routine converts the b-byte secret key, K, into an
   expanded key, S, which is a sequence of T = 2*(R+1) words. The
   expansion algorithm uses two constants that are derived from the
   constants, e, and phi. These are used to initialize S, which is then
   modified using K. A C code procedure header for this routine could
   be:

```
 /* Expand an RC5 user key.
  */
 void RC5_Key_Expand (b, K, R, S)
   int      b; /* Byte length of secret key */
   char        *K; /* Secret key */
   int      R; /* Number of rounds */
   RC5_WORD *S;     /* Expanded key buffer, 2*(R+1) words */
 {
```

5.3 Convert secret key from bytes to words

   This step converts the b-byte key into a sequence of words stored in
   the array L. On a little-endian processor this is accomplished by
   zeroing the L array and copying in the b bytes of K. The following C
   code will achieve this effect on all processors:

```
   int i, j, k, LL, t, T;
   RC5_WORD   L[256/WW];  /* Based on max key size */
   RC5_WORD   A, B;

   /* LL is number of elements used in L. */
   LL = (b + WW - 1) / WW;
   for (i = 0 ; i < LL ; i++)  {
       L[i] = 0;
   }
   for (i = 0 ; i < b ; i++)  {
       t = (K[i] & 0xFF) << (8*(i%4)); /* 0, 8, 16, 24*/
       L[i/WW] = L[i/WW] + t;
   }
```

5.4 Initialize the expanded key table

   This step fills in the S table with a fixed (key independent)
   pseudo-random pattern using an arithmetic progression based on Pw and
   Qw modulo 2**W. The element S[i] equals i*Qw + Pw modulo 2**W. This
   table could be precomputed and copied as needed or computed on the
   fly. In C code it can be computed by:

```
    T = 2*(R+1);
    S[0] = Pw;
    for (i = 1 ; i < T ; i++)  {
        S[i] = S[i-1] + Qw;
    }
```

5.5 Mix in the secret key

   This step mixes the secret key, K, into the expanded key, S. First
   the number of iterations of the mixing function, k, is set to three
   times the maximum of the number of initialized elements of L, called
   LL, and the number of elements in S, called T. Each iteration is
   similar to an interation of the encryption inner loop in that two
   variables A and B are updated by the first and second halves of the
   iteration.

   Initially A and B are zero as are the indexes into the S array, i,
   and the L array, j. In the first half of the iteration, a partial
   result is computed by summing S[i], A and B. The new value for A is
   this partial result rotated left three bits. The A value is then
   placed into S[i]. The second half of the iteration computes a second
   partial result that is the sum of L[j], A and B. The second partial
   result is then rotated left by A+B bit positions and set to be the
   new value for B. The new B value is then placed into L[j]. At the
   end of the iteration, i and j are incremented modulo the size of
   their respective arrays. In C code:

```
   i = j = 0;
   A = B = 0;
   if (LL > T)
       k = 3 * LL; /* Secret key len > expanded key. */
   else
       k = 3 * T;  /* Secret key len < expanded key. */
   for ( ; k > 0 ; k--)  {
       A = ROTL(S[i] + A + B, 3, W);
       S[i] = A;
       B = ROTL(L[j] + A + B, A + B, W);
       L[j] = B;
       i = (i + 1) % T;
       j = (j + 1) % LL;
   }
   return;
 } /* End of RC5_Key_Expand */
```

6. Description of RC5 Block Cipher

   This section describes the RC5 block cipher by explaining the steps
   required to perform an encryption of a single input block. The
   decryption process is the reverse of these steps so it will not be
   explained. The RC5 cipher is parameterized by a version number, V, a
   round count, R, and a word size in bits, W. This description
   corresponds to original version of RC5 (V = 16 decimal) and covers
   any positive value for R and the values 16, 32, and 64 for W.

   The inputs to this process are the expanded key table, S, the number
   of rounds, R, the input buffer pointer, in, and the output buffer
   pointer, out. A possible C code procedure header for this would be:

```
  void RC5_Block_Encrypt (S, R, in, out)
    RC5_WORD     *S;
    int  R;
    char     *in;
    char     *out;
  {
```

6.1 Loading A and B values

   This step converts input bytes into two unsigned integers called A
   and B. When RC5 is used as a 64 bit block cipher A and B are 32 bit
   values. The first input byte becomes the least significant byte of
   A, the fourth input byte becomes the most significant byte of A, the
   fifth input byte becomes the least significant byte of B and the last
   input byte becomes the most significant byte of B. This conversion
   can be very efficient for little-endian processors such as the Intel
   family. In C code this could be expressed as:

```
    int  i;
    RC5_WORD     A, B;

    A  =  in[0] & 0xFF;
    A += (in[1] & 0xFF) << 8;
    A += (in[2] & 0xFF) << 16;
    A += (in[3] & 0xFF) << 24;
    B  =  in[4] & 0xFF;
    B += (in[5] & 0xFF) << 8;
    B += (in[6] & 0xFF) << 16;
    B += (in[7] & 0xFF) << 24;
```

6.2 Iterating the round function

   This step mixes the expanded key with the input to perform the
   fundamental encryption operation. The first two words of the
   expanded key are added to A and B respectively, and then the round
   function is repeated R times.

   The first half of the round function computes a new value for A based
   on the values of A, B, and the next unused word in the expanded key
   table. Specifically, A is XOR'ed with B and then this first partial
   result is rotated to the left by an amount specified by B to form the
   second partial result. The rotation is performed on a W bit boundary
   (i.e., 32 bit rotation for the version of RC5 that has a 64 bit block
   size). The actual rotation amount only depends on the least
   significant log base-2 of W bits of B. The next unused word of the
   expanded key table is then added to the second partial result and
   this becomes the new value for A.

   The second half of the round function is identical except the roles
   of A and B are switched. Specifically, B is exclusive or'ed with A
   and then this first partial result is rotated to the left by an
   amount specified by A to form the second partial result. The next
   unused word of the expanded key table is then added to the second
   partial result and this becomes the new value for B.

   One way to express this in C code is:

```
 A = A + S[0];
 B = B + S[1];
 for (i = 1 ; i <= R ; i++) {
     A = A ^ B;
     A = ROTL(A, B, W) + S[2*i];
     B = B ^ A;
     B = ROTL(B, A, W) + S[(2*i)+1];
 }
```

6.3 Storing the A and B values

   The final step is to convert A and B back into a sequence of bytes.
   This is the inverse of the load operation. An expression of this in
   C code could be:

```
 out[0] = (A >>  0) & 0xFF;
 out[1] = (A >>  8) & 0xFF;
 out[2] = (A >> 16) & 0xFF;
 out[3] = (A >> 24) & 0xFF;
 out[4] = (B >>  0) & 0xFF;
 out[5] = (B >>  8) & 0xFF;
```

```
      out[6] = (B >> 16) & 0xFF;
      out[7] = (B >> 24) & 0xFF;
      return;
    } /* End of RC5_Block_Encrypt */
```

7. Description of RC5-CBC and RC5-CBC-Pad

   This section describes the CBC and CBC-Pad modes of the RC5 cipher.
   This description is based on the RC5 key objects and RC5 block cipher
   described earlier.

7.1 Creating cipher objects

   The cipher object needs to keep track of the padding mode, the number
   of rounds, the expanded key, the initialization vector, the CBC
   chaining block, and an input buffer. A possible structure definition
   for this in C code would be:

```
  /* Definition of the RC5 CBC algorithm object.
   */
  typedef struct rc5CBCAlg
  {
    int           Pad;   /* 1 = RC5-CBC-Pad, 0 = RC5-CBC. */
    int           R;     /* Number of rounds. */
    RC5_WORD      *S;     /* Expanded key. */
    unsigned char  I[BB]; /* Initialization vector. */
    unsigned char  chainBlock[BB];
    unsigned char  inputBlock[BB];
    int           inputBlockIndex; /* Next inputBlock byte. */
  } rc5CBCAlg;
```

   To create a cipher algorithm object, the parameters must be checked
   and then space allocated for the expanded key table. The expanded
   key is initialized using the method described earlier. Finally, the
   state variables (padding mode, number of rounds, and the input
   buffer) are set to their initial values. In C this could be
   accomplished by:

```
  /* Allocate and initialize the RC5 CBC algorithm object.
   * Return 0 if problems.
   */
  rc5CBCAlg *RC5_CBC_Create (Pad, R, Version, bb, I)
    int     Pad;        /* 1 = RC5-CBC-Pad, 0 = RC5-CBC. */
    int     R;          /* Number of rounds. */
    int     Version;    /* RC5 version number. */
    int     bb;         /* Bytes per RC5 block == IV len. */
    char    *I;         /* CBC IV, bb bytes long. */
  {
```

```
    rc5CBCAlg    *pAlg;
    int          index;

    if ((Version != RC5_FIRST_VERSION) ||
        (bb != BB) ||   (R < 0) || (255 < R))
        return ((rc5CBCAlg *) 0);
    pAlg = (rc5CBCAlg *) malloc (sizeof(*pAlg));
    if (pAlg == ((rc5CBCAlg *) 0))
        return ((rc5CBCAlg *) 0);
    pAlg->S = (RC5_WORD *) malloc (BB * (R + 1));
    if (pAlg->S == ((RC5_WORD *) 0))     {
        free (pAlg);
        return ((rc5CBCAlg *) 0);
    }
    pAlg->Pad = Pad;
    pAlg->R = R;
    pAlg->inputBlockIndex = 0;
    for (index = 0 ; index < BB ; index++)
        pAlg->I[index] = I[index];
    return (pAlg);
}
```

7.2 Destroying cipher objects

   Destroying the cipher object is the inverse of creating it with care
   being take to zero memory before returning it to the memory manager.
   In C this could be accomplished by:

```
  /* Zero and free an RC5 algorithm object.
   */
  void RC5_CBC_Destroy (pAlg)
    rc5CBCAlg    *pAlg;
  {
    RC5_WORD     *to;
    int      count;

    if (pAlg == ((rc5CBCAlg *) 0))
        return;
    if (pAlg->S == ((RC5_WORD *) 0))
        return;
    to = pAlg->S;
    for (count = 0 ; count < (1 + pAlg->R) ; count++)
    {
        *to++ = 0;   /* Two expanded key words per round. */
        *to++ = 0;
    }
   free (pAlg->S);
    for (count = 0 ; count < BB ; count++)
```

```
      {
          pAlg->I[count] = (unsigned char) 0;
          pAlg->inputBlock[count] = (unsigned char) 0;
          pAlg->chainBlock[count] = (unsigned char) 0;
      }
      pAlg->Pad = 0;
      pAlg->R = 0;
      pAlg->inputBlockIndex = 0;
      free (pAlg);
  }
```

7.3 Setting the IV for cipher objects

   For CBC cipher objects, the state of the algorithm depends on the
   expanded key, the CBC chain block, and any internally buffered input.
   Often the same key is used with many messages that each have a unique
   initialization vector. To avoid the overhead of creating a new
   cipher object, it makes more sense to provide an operation that
   allows the caller to change the initialization vector for an existing
   cipher object. In C this could be accomplished by the following
   code:

```
  /* Setup a new initialization vector for a CBC operation
   * and reset the CBC object.
   * This can be called after Final without needing to
   * call Init or Create again.
   * Return zero if problems.
   */
  int RC5_CBC_SetIV (pAlg, I)
    rc5CBCAlg   *pAlg;
    char        *I;     /* CBC Initialization vector, BB bytes. */
  {
    int     index;

    pAlg->inputBlockIndex = 0;
    for (index = 0 ; index < BB ; index++)
    {
        pAlg->I[index] = pAlg->chainBlock[index] = I[index];
        pAlg->inputBlock[index] = (unsigned char) 0;
    }
    return (1);
  }
```

7.4 Binding a key to a cipher object

   The operation that binds a key to a cipher object performs the key
   expansion. Key expansion could be an operation on keys, but that
   would not work correctly for ciphers that modify the expanded key as

they operate. After expanding the key, this operation must
initialize the CBC chain block from the initialization vector and
prepare the input buffer to receive the first character. In C this
could be done by:

```
/* Initialize the encryption object with the given key.
 * After this routine, the caller frees the key object.
 * The IV for this CBC object can be changed by calling
 * the SetIV routine.  The only way to change the key is
 * to destroy the CBC object and create a new one.
 * Return zero if problems.
 */
int RC5_CBC_Encrypt_Init (pAlg, pKey)
  rc5CBCAlg       *pAlg;
  rc5UserKey  *pKey;
{
  if ((pAlg == ((rc5CBCAlg *) 0)) ||
      (pKey == ((rc5UserKey *) 0)))
      return (0);
  RC5_Key_Expand (Key->keyLength, pKey->keyBytes,
                  pAlg->R, pAlg->S);
  return (RC5_CBC_SetIV(pAlg, pAlg->I));
}
```

7.5 Processing part of a message

The encryption process described here uses the Init-Update-Final
paradigm. The update operation can be performed on a sequence of
message parts in order to incrementally produce the ciphertext.
After the last part is processed, the Final operation is called to
pick up any plaintext bytes or padding that are buffered inside the
cipher object. An appropriate procedure header for this operation
would be:

```
/* Encrypt a buffer of plaintext.
 * The plaintext and ciphertext buffers can be the same.
 * The byte len of the ciphertext is put in *pCipherLen.
 * Call this multiple times passing successive
 * parts of a large message.
 * After the last part has been passed to Update,
 * call Final.
 * Return zero if problems like output buffer too small.
 */
int RC5_CBC_Encrypt_Update (pAlg, N, P,
                              pCipherLen, maxCipherLen, C)
  rc5CBCAlg   *pAlg;       /* Cipher algorithm object. */
  int         N;          /* Byte length of P. */
  char        *P;         /* Plaintext buffer. */
```

```
   int           *pCipherLen;/* Gets byte len of C. */
   int            maxCipherLen;  /* Size of C. */
   char          *C;            /* Ciphertext buffer. */
 {
```

7.5.1   Output buffer size check.

   The first step of plaintext processing is to make sure that the
   output buffer is big enough hold the ciphertext. The ciphertext will
   be produced in multiples of the block size and depends on the number
   of plaintext characters passed to this operation plus any characters
   that are in the cipher object's internal buffer. In C code this
   would be:

```
   int       plainIndex, cipherIndex, j;

   /* Check size of the output buffer. */
   if (maxCipherLen < (((pAlg->inputBlockIndex+N)/BB)*BB))
   {
       *pCipherLen = 0;
       return (0);
   }
```

7.5.2   Divide plaintext into blocks

   The next step is to add characters to the internal buffer until a
   full block has been constructed. When that happens, the buffer
   pointers are reset and the input buffer is exclusive-or'ed (XORed)
   with the CBC chaining block. The byte order of the chaining block is
   the same as the input block. For example, the ninth input byte is
   XOR'ed with the first ciphertext byte. The result is then passed to
   the RC5 block cipher which was described earlier. To reduce data
   movement and byte alignment problems, the output of RC5 can be
   directly written into the CBC chaining block. Finally, this output
   is copied to the ciphertext buffer provided by the user. Before
   returning, the actual size of the ciphertext is passed back to the
   caller. In C, this step can be performed by:

```
   plainIndex = cipherIndex = 0;
   while (plainIndex < N)
   {
       if (pAlg->inputBlockIndex < BB)
       {
           pAlg->inputBlock[pAlg->inputBlockIndex]
                   = P[plainIndex];
           pAlg->inputBlockIndex++;
           plainIndex++;
       }
```

```
        if (pAlg->inputBlockIndex == BB)
        {    /* Have a complete input block, process it. */
            pAlg->inputBlockIndex = 0;
            for (j = 0 ; j < BB ; j++)
            {    /* XOR in the chain block. */
                pAlg->inputBlock[j] = pAlg->inputBlock[j]
                                    ^ pAlg->chainBlock[j];
            }
            RC5_Block_Encrypt(pAlg->S, pAlg->R
                              pAlg->inputBlock,
                              pAlg->chainBlock);
            for (j = 0 ; j < BB ; j++)
            {    /* Output the ciphertext. */
                C[cipherIndex] = pAlg->chainBlock[j];
                cipherIndex++;
            }
        }
    }
    *pCipherLen = cipherIndex;
    return (1);
  } /* End of RC5_CBC_Encrypt_Update */
```

7.6 Final block processing

   This step handles the last block of plaintext. For RC5-CBC, this
   step just performs error checking to ensure that the plaintext length
   was indeed a multiple of the block length. For RC5-CBC-Pad, padding
   bytes are added to the plaintext. The pad bytes are all the same and
   are set to a byte that represents the number of bytes of padding.
   For example if there are eight bytes of padding, the bytes will all
   have the hexadecimal value 0x08. There will be between one and BB
   padding bytes, inclusive. In C code this would be:

```
  /* Produce the final block of ciphertext including any
   * padding, and then reset the algorithm object.
   * Return zero if problems.
   */
  int RC5_CBC_Encrypt_Final (pAlg, pCipherLen, maxCipherLen, C)
    rc5CBCAlg    *pAlg;
    int          *pCipherLen;    /* Gets byte len of C. */
    int           maxCipherLen;  /* Len of C buffer. */
    char         *C;             /* Ciphertext buffer. */
  {
    int    cipherIndex, j;
    int    padLength;

    /* For non-pad mode error if input bytes buffered. */
    *pCipherLen = 0;
```

```
   if ((pAlg->Pad == 0) && (pAlg->inputBlockIndex != 0))
       return (0);

   if (pAlg->Pad == 0)
       return (1);
   if (maxCipherLen < BB)
       return (0);

   padLength = BB - pAlg->inputBlockIndex;
   for (j = 0 ; j < padLength ; j++)
   {
       pAlg->inputBlock[pAlg->inputBlockIndex]
              = (unsigned char) padLength;
       pAlg->inputBlockIndex++;
   }
   for (j = 0 ; j < BB ; j++)
   {   /* XOR the chain block into the plaintext block. */
       pAlg->inputBlock[j] = pAlg->inputBlock[j]
                              ^ pAlg->chainBlock[j];
   }
   RC5_Block_Encrypt(pAlg->S, pAlg->R,
                     pAlg->inputBlock, pAlg->chainBlock);
   cipherIndex = 0;
   for (j = 0 ; j < BB ; j++)
   {   /* Output the ciphertext. */
       C[cipherIndex] = pAlg->chainBlock[j];
       cipherIndex++;
   }
   *pCipherLen = cipherIndex;

   /* Reset the CBC algorithm object. */
   return (RC5_CBC_SetIV(pAlg, pAlg->I));
 } /* End of RC5_CBC_Encrypt_Final */
```

8. Description of RC5-CTS

   The Cipher Text Stealing (CTS) mode for block ciphers is described by
   Schneier on pages 195 and 196 of [6]. This mode handles any length
   of plaintext and produces ciphertext whose length matches the
   plaintext length. The CTS mode behaves like the CBC mode for all but
   the last two blocks of the plaintext. The following steps describe
   how to handle the last two portions of the plaintext, called Pn-1 and
   Pn, where the length of Pn-1 equals the block size, BB, and the
   length of the last block, Pn, is Ln bytes. Notice that Ln ranges
   from 1 to BB, inclusive, so Pn could in fact be a complete block.

1. Exclusive-or Pn-1 with the previous ciphertext
   block, Cn-2, to create Xn-1.

2. Encrypt Xn-1 to create En-1.

3. Select the first Ln bytes of En-1 to create Cn.

4. Pad Pn with zeros at the end to create P of length BB.

5. Exclusive-or En-1 with P to create to create Dn.

6. Encrypt Dn to create Cn-1

7. The last two parts of the ciphertext are Cn-1 and
   Cn respectively.

To implement CTS encryption, the RC5-CTS object must hold on to
(buffer) at most 2*BB bytes of plaintext and process them specially
when the RC5_CTS_Encrypt_Final routine is called.

The following steps describe how to decrypt Cn-1 and Cn.

1. Decrypt Cn-1 to create Dn.

2. Pad Cn with zeros at the end to create C of length BB.

3. Exclusive-or Dn with C to create Xn.

4. Select the first Ln bytes of Xn to create Pn.

5. Append the tail (BB minus Ln) bytes of Xn to Cn
   to create En.

6. Decrypt En to create Pn-1.

7. The last two parts of the plaintext are Pn-1 and
   Pn respectively.

9. Test Program and Vectors

   To help confirm the correctness of an implementation, this section
   gives a test program and results from a set of test vectors.

9.1 Test Program

   The following test program written in C reads test vectors from its
   input stream and writes results on its output stream. The following
   subsections give a set of test vectors for inputs and the resulting

```
 outputs.

 #include <stdio.h>

 #define BLOCK_LENGTH      (8 /* bytes */)
 #define MAX_KEY_LENGTH    (64 /* bytes */)
 #define MAX_PLAIN_LENGTH  (128 /* bytes */)
 #define MAX_CIPHER_LENGTH(MAX_PLAIN_LENGTH + BLOCK_LENGTH)
 #define MAX_ROUNDS        (20)
 #define MAX_S_LENGTH      (2 * (MAX_ROUNDS + 1))

 typedef struct test_vector
 {
   int padding_mode;
   int rounds;
   char    keytext[2*MAX_KEY_LENGTH+1];
   int key_length;
   char    key[MAX_KEY_LENGTH];
   char    ivtext[2*BLOCK_LENGTH+1];
   int iv_length;
   char    iv[BLOCK_LENGTH];
   char    plaintext[2*MAX_PLAIN_LENGTH+1];
   int plain_length;
   char    plain[MAX_PLAIN_LENGTH];
   char    ciphertext[2*MAX_CIPHER_LENGTH+1];
   int cipher_length;
   char    cipher[MAX_CIPHER_LENGTH];
   RC5_WORD    S[MAX_S_LENGTH];
 } test_vector;

 void show_banner()
 {
   (void) printf("RC5 CBC Tester.\n");
   (void) printf("Each input line should contain the following\n");
   (void) printf("test parameters separated by a single space:\n");
   (void) printf("- Padding mode flag.  Use 1 for RC5_CBC_Pad, else
 0.\n");
   (void) printf("- Number of rounds for RC5.\n");
   (void) printf("- Key bytes in hexadecimal.  Two characters per
 byte like '01'.\n");
   (void) printf("- IV bytes in hexadecimal.  Must be 16 hex
 characters.\n");
   (void) printf("- Plaintext bytes in hexadecimal.\n");
   (void) printf("An end of file or format error terminates the
 tester.\n");
   (void) printf("\n");
 }
```

```
  /* Convert a buffer from ascii hex to bytes.
   * Set pTo_length to the byte length of the result.
   * Return 1 if everything went OK.
   */
  int hex_to_bytes (from, to, pTo_length)
    char    *from, *to;
    int     *pTo_length;
  {
    char    *pHex;  /* Ptr to next hex character. */
    char    *pByte;    /* Ptr to next resulting byte. */
    int  byte_length = 0;
    int  value;

    pByte = to;
    for (pHex = from ; *pHex != 0 ; pHex += 2)  {
        if (1 != sscanf(pHex, "%02x", &value))
            return (0);
        *pByte++ = ((char)(value & 0xFF));
        byte_length++;
    }
    *pTo_length = byte_length;
    return (1);
  }

  /* Convert a buffer from bytes to ascii hex.
   * Return 1 if everything went OK.
   */
  int bytes_to_hex (from, from_length, to)
    char    *from, *to;
    int from_length;
  {
    char    *pHex;  /* Ptr to next hex character. */
    char    *pByte;    /* Ptr to next resulting byte. */
    int  value;

    pHex = to;
    for (pByte = from ; from_length > 0 ; from_length--)  {
        value = *pByte++ & 0xFF;
        (void) sprintf(pHex, "%02x", value);
        pHex += 2;
    }
    return (1);
  }

  /* Return 1 if get a valid test vector. */
  int get_test_vector(ptv)
    test_vector *ptv;
  {
```

```
   if (1 != scanf("%d", &ptv->padding_mode))
      return (0);
   if (1 != scanf("%d", &ptv->rounds))
      return (0);
   if ((ptv->rounds < 0) || (MAX_ROUNDS < ptv->rounds))
      return (0);
   if (1 != scanf("%s", &ptv->keytext))
      return (0);
   if (1 != hex_to_bytes(ptv->keytext, ptv->key,
                      &ptv->key_length))
      return (0);
   if (1 != scanf("%s", &ptv->ivtext))
      return (0);
   if (1 != hex_to_bytes(ptv->ivtext, ptv->iv,
                      &ptv->iv_length))
      return (0);
   if (BLOCK_LENGTH != ptv->iv_length)
      return (0);
   if (1 != scanf("%s", &ptv->plaintext))
      return (0);
   if (1 != hex_to_bytes(ptv->plaintext, ptv->plain,
                      &ptv->plain_length))
      return (0);
   return (1);
}

void run_test (ptv)
  test_vector *ptv;
{
  rc5UserKey  *pKey;
  rc5CBCAlg       *pAlg;
  int          numBytesOut;

  pKey = RC5_Key_Create ();
  RC5_Key_Set (pKey, ptv->key_length, ptv->key);

  pAlg = RC5_CBC_Create (ptv->padding_mode,
                 ptv->rounds,
                 RC5_FIRST_VERSION,
                 BB,
                 ptv->iv);
  (void) RC5_CBC_Encrypt_Init (pAlg, pKey);
  ptv->cipher_length = 0;
  (void) RC5_CBC_Encrypt_Update (pAlg,
                 ptv->plain_length, ptv->plain,
                 &(numBytesOut),
                 MAX_CIPHER_LENGTH - ptv->cipher_length,
                 &(ptv->cipher[ptv->cipher_length]));
```

```
      ptv->cipher_length += numBytesOut;
      (void) RC5_CBC_Encrypt_Final (pAlg,
                      &(numBytesOut),
                      MAX_CIPHER_LENGTH - ptv->cipher_length,
                      &(ptv->cipher[ptv->cipher_length]));
      ptv->cipher_length += numBytesOut;
      bytes_to_hex (ptv->cipher, ptv->cipher_length,
                  ptv->ciphertext);
      RC5_Key_Destroy (pKey);
      RC5_CBC_Destroy (pAlg);
  }

  void show_results (ptv)
    test_vector *ptv;
  {
    if (ptv->padding_mode)
        printf ("RC5_CBC_Pad ");
    else
        printf ("RC5_CBC     ");
    printf ("R = %2d ", ptv->rounds);
    printf ("Key = %s ", ptv->keytext);
    printf ("IV = %s ", ptv->ivtext);
    printf ("P = %s ", ptv->plaintext);
    printf ("C = %s", ptv->ciphertext);
    printf ("\n");
  }

  int main(argc, argv)
    int argc;
    char *argv[];
  {
    test_vector tv;
    test_vector *ptv = &tv;

    show_banner();
    while (get_test_vector(ptv))  {
        run_test(ptv);
        show_results(ptv);
    }
    return (0);
  }
```

9.2 Test vectors

   The following text is an input file to the test program presented in
   the previous subsection. The output is given in the next subsection.

```
 0 00 00                   0000000000000000 0000000000000000
 0 00 00                   0000000000000000 ffffffffffffffff
 0 00 00                   0000000000000001 0000000000000000
 0 00 00                   0000000000000000 0000000000000001
 0 00 00                   0102030405060708 1020304050607080
 0 01 11                   0000000000000000 0000000000000000
 0 02 00                   0000000000000000 0000000000000000
 0 02 00000000             0000000000000000 0000000000000000
 0 08 00                   0000000000000000 0000000000000000
 0 08 00                   0102030405060708 1020304050607080
 0 12 00                   0102030405060708 1020304050607080
 0 16 00                   0102030405060708 1020304050607080
 0 08 01020304             0000000000000000 ffffffffffffffff
 0 12 01020304             0000000000000000 ffffffffffffffff
 0 16 01020304             0000000000000000 ffffffffffffffff
 0 12 0102030405060708     0000000000000000 ffffffffffffffff
 0 08 0102030405060708     0102030405060708 1020304050607080
 0 12 0102030405060708     0102030405060708 1020304050607080
 0 16 0102030405060708     0102030405060708 1020304050607080
 0 08 01020304050607081020304050607080
                           0102030405060708 1020304050607080
 0 12 01020304050607081020304050607080
                           0102030405060708 1020304050607080
 0 16 01020304050607081020304050607080
                           0102030405060708 1020304050607080

 0 12 0102030405           0000000000000000 ffffffffffffffff
 0 08 0102030405           0000000000000000 ffffffffffffffff
 0 08 0102030405           7875dbf6738c6478 0808080808080808
 1 08 0102030405           0000000000000000 ffffffffffffffff

 0 08 0102030405           0000000000000000 0000000000000000
 0 08 0102030405           7cb3f1df34f94811 1122334455667701

 1 08 0102030405           0000000000000000
ffffffffffffffff7875dbf6738c647811223344556677
```

9.3 Test results

   The following text is the output produced by the test program run on
   the inputs given in the previous subsection.

   RC5 CBC Tester.
   Each input line should contain the following
   test parameters separated by a single space:
   - Padding mode flag. Use 1 for RC5_CBC_Pad, else 0.
   - Number of rounds for RC5.
   - Key bytes in hexadecimal. Two characters per byte
     like '01'.
   - IV bytes in hexadecimal. Must be 16 hex characters.
   - Plaintext bytes in hexadecimal.
   An end of file or format error terminates the tester.

   RC5_CBC   R =   0 Key = 00 IV = 0000000000000000
    P = 0000000000000000 C = 7a7bba4d79111d1e
   RC5_CBC   R =   0 Key = 00 IV = 0000000000000000
    P = ffffffffffffffff C = 797bba4d78111d1e
   RC5_CBC   R =   0 Key = 00 IV = 0000000000000001
    P = 0000000000000000 C = 7a7bba4d79111d1f
   RC5_CBC   R =   0 Key = 00 IV = 0000000000000000
    P = 0000000000000001 C = 7a7bba4d79111d1f
   RC5_CBC   R =   0 Key = 00 IV = 0102030405060708
    P = 1020304050607080 C = 8b9ded91ce7794a6
   RC5_CBC   R =   1 Key = 11 IV = 0000000000000000
    P = 0000000000000000 C = 2f759fe7ad86a378
   RC5_CBC   R =   2 Key = 00 IV = 0000000000000000
    P = 0000000000000000 C = dca2694bf40e0788
   RC5_CBC   R =   2 Key = 00000000 IV = 0000000000000000
    P = 0000000000000000 C = dca2694bf40e0788
   RC5_CBC   R =   8 Key = 00 IV = 0000000000000000
    P = 0000000000000000 C = dcfe098577eca5ff
   RC5_CBC   R =   8 Key = 00 IV = 0102030405060708
    P = 1020304050607080 C = 9646fb77638f9ca8
   RC5_CBC   R = 12 Key = 00 IV = 0102030405060708
    P = 1020304050607080 C = b2b3209db6594da4
   RC5_CBC   R = 16 Key = 00 IV = 0102030405060708
    P = 1020304050607080 C = 545f7f32a5fc3836
   RC5_CBC   R =   8 Key = 01020304 IV = 0000000000000000
    P = ffffffffffffffff C = 8285e7c1b5bc7402
   RC5_CBC   R = 12 Key = 01020304 IV = 0000000000000000
    P = ffffffffffffffff C = fc586f92f7080934
   RC5_CBC   R = 16 Key = 01020304 IV = 0000000000000000
    P = ffffffffffffffff C = cf270ef9717ff7c4
   RC5_CBC   R = 12 Key = 0102030405060708 IV = 0000000000000000
    P = ffffffffffffffff C = e493f1c1bb4d6e8c

```
RC5_CBC     R =  8 Key = 0102030405060708 IV = 0102030405060708
 P = 1020304050607080 C = 5c4c041e0f217ac3
RC5_CBC     R = 12 Key = 0102030405060708 IV = 0102030405060708
 P = 1020304050607080 C = 921f12485373b4f7
RC5_CBC     R = 16 Key = 0102030405060708 IV = 0102030405060708
 P = 1020304050607080 C = 5ba0ca6bbe7f5fad
RC5_CBC     R =  8 Key = 01020304050607081020304050607080
 IV = 0102030405060708
 P = 1020304050607080 C = c533771cd0110e63
RC5_CBC     R = 12 Key = 01020304050607081020304050607080
 IV = 0102030405060708
 P = 1020304050607080 C = 294ddb46b3278d60
RC5_CBC     R = 16 Key = 01020304050607081020304050607080
 IV = 0102030405060708
 P = 1020304050607080 C = dad6bda9dfe8f7e8
RC5_CBC     R = 12 Key = 0102030405 IV = 0000000000000000
 P = ffffffffffffffff C = 97e0787837ed317f
RC5_CBC     R =  8 Key = 0102030405 IV = 0000000000000000
 P = ffffffffffffffff C = 7875dbf6738c6478
RC5_CBC     R =  8 Key = 0102030405 IV = 7875dbf6738c6478
 P = 0808080808080808 C = 8f34c3c681c99695
RC5_CBC_Pad R =  8 Key = 0102030405 IV = 0000000000000000
 P = ffffffffffffffff C = 7875dbf6738c64788f34c3c681c99695
RC5_CBC     R =  8 Key = 0102030405 IV = 0000000000000000
 P = 0000000000000000 C = 7cb3f1df34f94811
RC5_CBC     R =  8 Key = 0102030405 IV = 7cb3f1df34f94811
 P = 1122334455667701 C = 7fd1a023a5bba217
RC5_CBC_Pad R =  8 Key = 0102030405 IV = 0000000000000000
 P = ffffffffffffffff7875dbf6738c647811223344556677
 C = 7875dbf6738c64787cb3f1df34f948117fd1a023a5bba217
```

10. Security Considerations

   The RC5 cipher is relatively new so critical reviews are still being
   performed. However, the cipher's simple structure makes it easy to
   analyze and hopefully easier to assess its strength. Reviews so far
   are very promising.

   Early results [1] suggest that for RC5 with a 64 bit block size (32
   bit word size), 12 rounds will suffice to resist linear and
   differential cyptanalysis. The 128 bit block version has not been
   studied as much as the 64 bit version, but it appears that 16 rounds
   would be an appropriate minimum. Block sizes less than 64 bits are
   academically interesting but should not be used for cryptographic
   security. Greater security can be achieved by increasing the number
   of rounds at the cost of decreasing the throughput of the cipher.

The length of the secret key helps determine the cipher's resistance
to brute force key searching attacks. A key length of 128 bits
should give adequate protection against brute force key searching by
a well funded opponent for a couple decades [7]. For RC5 with 12
rounds, the key setup time and data encryption time are the same for
all key lengths less than 832 bits, so there is no performance reason
for choosing short keys. For larger keys, the key expansion step
will run slower because the user key table, L, will be longer than
the expanded key table, S. However, the encryption time will be
unchanged since it is only a function of the number of rounds.

To comply with export regulations it may be necessary to choose keys
that only have 40 unknown bits. A poor way to do this would be to
choose a simple 5 byte key. This should be avoided because it would
be easy for an opponent to pre-compute key searching information.
Another common mechanism is to pick a 128 bit key and publish the
first 88 bits. This method reveals a large number of the entries in
the user key table, L, and the question of whether RC5 key expansion
provides adequate security in this situation has not been studied,
though it may be fine. A conservative way to conform to a 40 bit
limitation is to pick a seed value of 128 bits, publish 88 bits of
this seed, run the entire seed through a hash function like MD5 [4],
and use the 128 bit output of the hash function as the RC5 key.

In the case of 40 unknown key bits with 88 known key bits (i.e., 88
salt bits) there should still be 12 or more rounds for the 64 bit
block version of RC5, otherwise the value of adding salt bits to the
key is likely to be lost.

The lifetime of the key also influences security. For high security
applications, the key to any 64 bit block cipher should be changed
after encrypting 2^{32} blocks (2^{64} blocks for a 128 bit block
cipher). This helps to guard against linear and differential
cryptanalysis. For the case of 64 bit blocks, this rule would
recommend changing the key after 2^{40} (i.e. 10^{12}) bytes are
encrypted. See Schneier [6] page 183 for further discussion.

11. ASN.1 Identifiers

   For applications that use ASN.1 descriptions, it is necessary to
   define the algorithm identifier for these ciphers along with their
   parameter block formats. The ASN.1 definition of an algorithm
   identifier already exists and is listed below for reference.

   AlgorithmIdentifier ::= SEQUENCE {
     algorithm   OBJECT IDENTIFIER,
     parameters   ANY DEFINED BY algorithm OPTIONAL
   }

   The values for the algorithm field are:

   RC5_CBC   OBJECT IDENTIFIER ::=
     { iso (1) member-body (2) US (840) rsadsi (113549)
       encryptionAlgorithm (3) RC5CBC (8) }

   RC5_CBC_Pad OBJECT IDENTIFIER ::=
   { iso (1) member-body (2) US (840) rsadsi (113549)
     encryptionAlgorithm (3) RC5CBCPAD (9) }

   The structure of the parameters field for these algorithms is given
   below. NOTE: if the iv field is not included, then the
   initialization vector defaults to a block of zeros whose size depends
   on the blockSizeInBits field.

   RC5_CBC_Parameters ::= SEQUENCE {
     version   INTEGER (v1_0(16)),
     rounds   INTEGER (8..127),
     blockSizeInBits   INTEGER (64, 128),
     iv   OCTET STRING OPTIONAL
   }

References

   [1] Kaliski, Burton S., and Yinqun Lisa Yin, "On Differential and
   Linear Cryptanalysis of the RC5 Encryption Algorithm", In Advances
   in Cryptology - Crypto '95, pages 171-184, Springer-Verlag, New
   York, 1995.

   [2] Rivest, Ronald L., "The RC5 Encryption Algorithm", In
   Proceedings of the Second International Workshop on Fast Software
   Encryption, pages 86-96, Leuven Belgium, December 1994.

   [3] Rivest, Ronald L., "RC5 Encryption Algorithm", In Dr. Dobbs
   Journal, number 226, pages 146-148, January 1995.

    [4] Rivest, Ronald L., "The MD5 Message-Digest Algorithm", RFC
    1321.

    [5] RSA Laboratories, "Public Key Cryptography Standards (PKCS)",
    RSA Data Security Inc. See ftp.rsa.com.

    [6] Schneier, Bruce, "Applied Cryptography", Second Edition, John
    Wiley and Sons, New York, 1996. Errata: on page 195, line 13, the
    reference number should be [402].

    [7] Business Software Alliance, Matt Blaze et al., "Minimum Key
    Length for Symmetric Ciphers to Provide Adequate Commercial
    Security", http://www.bsa.org/bsa/cryptologists.html.

    [8] RSA Data Security Inc., "RC5 Reference Code in C", See the web
    site: www.rsa.com, for availability. Not available with the first
    draft of this document.

Authors' Addresses

    Robert W. Baldwin
    RSA Data Security, Inc.
    100 Marine Parkway
    Redwood City, CA 94065

    Phone: (415) 595-8782
    Fax: (415) 595-1873
    EMail: baldwin@rsa.com, or baldwin@lcs.mit.edu

    Ronald L. Rivest
    Massachusetts Institute of Technology
    Laboratory for Computer Science
    NE43-324
    545 Technology Square
    Cambridge, MA 02139-1986

    Phone: (617) 253-5880
    EMail: rivest@theory.lcs.mit.edu