ANSI C Cryptographic API Profile
for AES Candidate Algorithm Submissions

Revision 5: April 15, 1998

1. Overview

This document specifies the ANSI C interface profile for implementations of AES candidate algorithms. C implementations shall support the syntax and parameterization of the interface profile messages as described in this API. The functions specified in this API have return values listed that are largely used to supply error codes in the event of incomplete execution of the routines. The error values listed are not meant to be an exhaustive list. If additional error codes are useful for your implementation, please provide them.

2. Key Generation Interface

Each AES submitter will be required to implement this interface because NIST anticipates that some candidate algorithms will have unique requirements for and methods of key generation. Implementations shall support key of lengths of 128, 192, and 256-bit. Additionally, if an algorithm can support other key sizes, these should be supported in the code as well.

The ANSI C key generation programming interface uses one structure and one routine to manipulate keys. The structure, *keyInstance*, contains the length of the key, the raw key material, a direction flag that indicates if the key will be used for encryption or decryption, and any algorithm specific key information such as the key schedule used in DES. **All implementations must be sure to document any algorithm-specific parameters and their use.**

The key function, *makeKey()*, is called with the appropriate parameters which get loaded into the *keyInstance* structure. These parameters are then used to perform any key specific setup that is necessary, e.g., allocation and initialization of a key schedule table.

```
typedef struct {
        BYTE  direction;
        int       keyLen;
        char     keyMaterial[MAX_KEY_SIZE+1];
        /* The following parameters are algorithm dependent */
        } keyInstance;
```

*(4/15/98 – changed BYTE *keyMaterial to char keyMaterial[MAX_KEY_SIZE+1] to avoid malloc and free calls.)*

❖ **makeKey**

int makeKey(keyInstance *key, BYTE direction, int keyLen, char *keyMaterial)

Initializes a keyInstance with the following information:

- direction: the key is being setup for encryption or decryption
- keyLen: The key length (128, 192, 256, or others) of the key, and
- keyMaterial: The raw key data.

Parameters:

key: a structure that holds the keyInstance information
direction: the key is being setup for encryption or decryption
keyLen: an integer value that indicates the length of the key in bits.
keyMaterial: the raw key information (keyLen/4 ASCII characters representing the hex values for the key). For example, "0123456789abcdef0123456789abcdef" is the string for a key with the binary value:
00000001001000110100010101100111100010011010101111100…

Returns:

TRUE - on success
BAD_KEY_DIR - direction is invalid (e.g., unknown value)
BAD_KEY_MAT - keyMaterial is invalid (e.g., wrong length)

3. Cipher Object Interface

The ANSI C cipher programming interface uses one structures and a set of functions to manipulate cipher data. The structure, *cipherInstance*, contains fields for the mode being used (e.g., Electronic Codebook, Cipher Block Chaining, or 1-bit Cipher Feedback) and an initialization vector necessary for some modes. Additional algorithm-specific parameters may be added if necessary. **All implementations must be sure to document any algorithm-specific parameters and their use.**

The cipher routines get used in following way. First, *cipherInit()* is called with the appropriate parameters to be loaded into the *cipherInstance* structure. *cipherInit()* will perform any additional algorithm setup that is required, e.g., establishing an Initialization Vector. Then full blocks of data are supplied to either *blockEncrypt()* or *blockDecrypt()* for ciphering. The data passed to blockEncrypt() and blockDecrypt() must be integral block units, i.e., n*blocksize bits long (this will allow for more accurate testing of bulk encryption times). If any algorithm specific parameters are needed, they should be loaded before calling *cipherInit()*.

```
typedef struct {
        BYTE mode;
        BYTE IV[MAX_IV_SIZE];
        /* Add any algorithm specific parameters needed here  */
        } cipherInstance;
```

*(4/15/98 – changed BYTE *IV to BYTE IV[MAX_IV_SIZE] to avoid malloc and free calls.)*

- **cipherInit**

int cipherInit(cipherInstance *cipher, BYTE mode, char *IV)

Initializes the cipher with the mode and, if present, sets the Initialization Vector. If any algorithm specific setup is necessary, cipherInit() must take care of that as well. The IV parameter passed to cipherInit() is an ASCII hex string representation of the IV, i.e. the IV passed as a parameter will typically be 32 bytes long. The IV field of the cipherInstance structure is the binary value of the IV, i.e. it will typically be 16 bytes long.

Algorithm specific parameters must be loaded into the cipherInstance structure before calling cipherInit(). For example, if the algorithm can use other block sizes than 128-bits, a field should be added to the cipherInstance structure and the value being used should be loaded into the cipher parameter before calling cipherInit().

Parameters:

cipher – the cipherInstance being loaded
mode - the operation mode of this cipher (this is one of MODE_ECB, MODE_CBC, or MODE_CFB1)
IV - the cipher initialization vector, necessary for some modes

Returns:

TRUE - on success
BAD_CIPHER_MODE - the mode passed is unknown.

- **blockEncrypt**

int blockEncrypt(cipherInstance *cipher, keyInstance *key, BYTE *input, int inputLen, BYTE *outBuffer)

Uses the cipherInstance object and the keyInstance object to encrypt one block of data in the input buffer. The output (the encrypted data) is returned in outBuffer, which is the same size as inputLen. The routine

returns the number of bits enciphered. inputLen will typically be 128 bits, but some algorithms may handle additional block sizes. Additionally, it is acceptable to use this routine to encrypt multiple "blocks" of data with one call. For example, if your algorithm has a block size of 128 bits, it is acceptable to pass n*128 bits to blockEncrypt().

(4/15/98 – Allow blockEncrypt() to handle an integral number of algorithm blocks in one call.)

Parameters:

cipher – the cipherInstance to be used
key – the ciphering key
input - the input buffer
inputLen - the input length, in bits
outBuffer – contains the encrypted data

Returns:

The number of bits ciphered, or
BAD_CIPHER_STATE - cipher in bad state (e.g., not initialized)
BAD_KEY_MATERIAL – direction not set for DIR_ENCRYPT

❖ **blockDecrypt**

int blockDecrypt(cipherInstance *cipher, keyInstance *key, BYTE *input, int inputLen, BYTE *outBuffer)

Uses the cipherInstance object and the keyInstance object to decrypt one block of data in the input buffer. The output (the decrypted data) is returned in outBuffer, which is the same size as inputLen. The routine returns the number of bits deciphered. inputLen will typically be 128 bits, but some algorithms may handle additional block sizes. Additionally, it is acceptable to use this routine to decrypt multiple "blocks" of data with one call. For example, if your algorithm has a block size of 128 bits, it is acceptable to pass n*128 bits to blockDecrypt().

(4/15/98 – Allow blockDecrypt() to handle an integral number of algorithm blocks in one call.)

Parameters:

cipher – the cipherInstance to be used
key – the ciphering key
input - the input buffer
inputLen - the input length, in bits
outBuffer – contains the decrypted data

Returns:

The number of bits ciphered, or
BAD_CIPHER_STATE - cipher in bad state (e.g., not initialized)
BAD_KEY_MATERIAL – direction not set for DIR_DECRYPT

```
/*  aes.h  */

/*  AES Cipher header file for ANSI C Submissions
      Lawrence E. Bassham III
      Computer Security Division
      National Institute of Standards and Technology

    This sample is to assist implementers developing to the
Cryptographic API Profile for AES Candidate Algorithm Submissions.
Please consult this document as a cross-reference.

    ANY CHANGES, WHERE APPROPRIATE, TO INFORMATION PROVIDED IN THIS FILE
MUST BE DOCUMENTED.  CHANGES ARE ONLY APPROPRIATE WHERE SPECIFIED WITH
THE STRING "CHANGE POSSIBLE".  FUNCTION CALLS AND THEIR PARAMETERS
CANNOT BE CHANGED.  STRUCTURES CAN BE ALTERED TO ALLOW IMPLEMENTERS TO
INCLUDE IMPLEMENTATION SPECIFIC INFORMATION.
*/

/*  Includes:
      Standard include files
*/

#include <stdio.h>

/*  Defines:
      Add any additional defines you need
*/

#define     DIR_ENCRYPT     0    /*  Are we encrpyting?  */
#define     DIR_DECRYPT     1    /*  Are we decrpyting?  */
#define     MODE_ECB        1    /*  Are we ciphering in ECB mode?   */
#define     MODE_CBC        2    /*  Are we ciphering in CBC mode?   */
#define     MODE_CFB1       3    /*  Are we ciphering in 1-bit CFB mode?
*/
#define     TRUE            1
#define     FALSE           0

/*  Error Codes - CHANGE POSSIBLE: inclusion of additional error codes
*/
#define     BAD_KEY_DIR        -1  /*  Key direction is invalid, e.g.,
                                       unknown value */
#define     BAD_KEY_MAT        -2  /*  Key material not of correct
                                       length */
#define     BAD_KEY_INSTANCE   -3  /*  Key passed is not valid  */
#define     BAD_CIPHER_MODE    -4  /*  Params struct passed to
                                       cipherInit invalid */
#define     BAD_CIPHER_STATE   -5  /*  Cipher in wrong state (e.g., not
                                       initialized) */

/*  CHANGE POSSIBLE:  inclusion of algorithm specific defines  */
#define     MAX_KEY_SIZE       64  /* # of ASCII char's needed to
                                       represent a key */
#define     MAX_IV_SIZE        16  /* # of bytes needed to
                                       represent an IV  */

/*  Typedefs:

      Typedef'ed data storage elements.  Add any algorithm specific
parameters at the bottom of the structs as appropriate.
*/
```

```
typedef    unsigned char    BYTE;

/*  The structure for key information */
typedef struct {
      BYTE  direction;     /*  Key used for encrypting or decrypting? */
      int   keyLen;        /*  Length of the key  */
      char  keyMaterial[MAX_KEY_SIZE+1];  /*  Raw key data in ASCII,
                                  e.g., user input or KAT values */
      /*  The following parameters are algorithm dependent, replace or
             add as necessary  */
      BYTE  *KS;           /*  (example)Pointer to a Key Schedule, a la
                                  DES */
      } keyInstance;

/*  The structure for cipher information */
typedef struct {
      BYTE  mode;            /* MODE_ECB, MODE_CBC, or MODE_CFB1 */
      BYTE  IV[MAX_IV_SIZE]; /* A possible Initialization Vector for
                                  ciphering */
      /*  Add any algorithm specific parameters needed here  */
      int   blockSize;    /* Sample: Handles non-128 bit block sizes
                                  (if available) */
      } cipherInstance;


/*  Function protoypes  */
int makeKey(keyInstance *key, BYTE direction, int keyLen,
                  char *keyMaterial);

int cipherInit(cipherInstance *cipher, BYTE mode, char *IV);

int blockEncrypt(cipherInstance *cipher, keyInstance *key, BYTE *input,
                  int inputLen, BYTE *outBuffer);

int blockDecrypt(cipherInstance *cipher, keyInstance *key, BYTE *input,
                  int inputLen, BYTE *outBuffer);
```