

# Cryptographic API Profile for AES Candidate Algorithm Submissions

(for ANSI C and Java™ implementations)

Original: January 16, 1998  
Revision 1: February 6, 1998  
Revision 2: March 19, 1998

## 1. Overview

This document specifies interface profiles for implementations of AES candidate algorithms. Since AES submissions will be written in the Java™ and C languages, two profiles are required.

The Java profile is a direct adaptation from Sun Microsystems' Java Cryptography Extension (JCE) v1.2 and the Java Development Kit (JDK) v1.2 specifications. Both are available in beta release form on Sun's Java Developer's Connection:

<<http://developer.javasoft.com/developer/jdchome.html>>

as of January 12, 1998. In cases where there is a discrepancy between this specification and the documentation published by Sun, Sun's specifications shall take precedence.

The Java interface profile assumes that AES submitters have coded their Java implementations according to the methods specified in the Java Cryptography Architecture and Java Cryptography Extension v1.2 documentation published by Sun. These documents and associated software development tools are available at the URL listed above. The C interface specification is an adaptation of the Java specification written in a non-object-oriented fashion.

(3/19/98) To accommodate international submitters who may not have access to the U.S. domestic version of JCE1.2, NIST is willing to accept AES packages constructed under international versions of the JCE (sometimes called IJCE) that are compatible with JCE1.1 or greater. One such implementation has been created by the Cryptix Development Team (<http://www.t-and-g.fl.net.au/java/cryptix/aes/#IJCE>), and links to other international JCE implementations are included in a Cryptix FAQ. Any JCE implementation used to create an AES submission package must be freely available to NIST.

(3/19/98) AES provider packages will be installed in the test environment as described in the Java Cryptography Architecture Specification, and must follow the naming conventions set by NIST. Packages shall be named as recommended by Sun in the JDK documentation. A sample name is "COM.acme.provider.ALGORITHM-NAME", where the candidate algorithm *ALGORITHM-NAME*, from the company at the domain *acme.com*, submits a provider package. C language implementations will not be required

to run in the Java environment. C implementations will only need to support the syntax and parameterization of the interface profile messages as described in the API.

(2/6/98) Also note that, since JCE provides the connection between the cryptographic API and the lower level Service Provider Interface (SPI), Java AES provider packages will implement the SPI.

(2/6/98) *Note that there are some modifications to the following sections, in light of the previous paragraph.*

(3/19/98) *A separate and distinct ANSI C language profile was added, in order to clarify the requirements for implementors. The Java API specification was also reformatted for improved readability.*

## 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 generation of 128, 192, and 256-bit keys, and are responsible for controlling the key generation process to avoid the possibility of creating the equivalent of weak keys for a given algorithm.

### 2.1. Java Specification

The following methods belong to the KeyGenerator and SecretKeyFactory classes. The AES subclass of KeyGenerator shall be AESKeyGenerator. Key objects with predetermined values will be created by instantiating AESKeySpec objects containing the desired key material, and then calling the generateSecret method of the SecretKeyFactory class to convert these key specifications into opaque key objects.

- Package javax.crypto.spec
- public class **AESKeySpec**  
extends Object  
implements KeySpec

#### ❖ **AESKeySpec**

```
public AESKeySpec (byte[] key) throws InvalidKeyException
```

Uses the first 128, 192, or 256 bytes in key as the AES key

**Parameters:**

key – the buffer with the AES key

**Throws:**

InvalidKeyException - if the given key material is not of the correct length

❖ **getKey**

```
public byte[] getKey()
```

Returns the AES key.

**Returns:**

the AES key

- Class javax.crypto.KeyGenerator

Public class **KeyGenerator**  
extends Object

❖ **getInstance**

```
public static final KeyGenerator getInstance(String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException
```

Generates a KeyGenerator object for the specified key algorithm from the specified provider.

**Parameters:**

algorithm - the standard name of the requested key algorithm, "AES"

Provider - the name of the provider as assigned by NIST

**Returns:**

the new KeyGenerator object

**Throws:**

NoSuchAlgorithmException - if the key generator for the requested algorithm is not available

NoSuchProviderException - if the requested provider is not available

#### ❖ **Init**

public final void Init(AlgorithmParameterSpec params, SecureRandom random) throws InvalidAlgorithmParameterException

Initializes the key generator with the specified parameter set and a user-provided source of randomness.

#### **Parameters:**

params - the key generation parameters. The first element in this parameter list will be the required keylength in bits, e.g. the ASCII text representation of the decimal number 128, 192, or 256.

random - the source of randomness for this key generator. This parameter is not relevant to the AES test process and can be ignored by a given AES implementation. It will be set to an arbitrary value by the AES test suite unless a submitter specifically requires it's use as stated in the provider documentation.

#### **Throws:**

InvalidAlgorithmParameterException - if params is inappropriate for this key generator

#### ❖ **GenerateKey**

public final SecretKey GenerateKey()

Generates a secret key.

#### **Returns:**

the new key

- public class **SecretKeyFactory**  
extends Object

#### ❖ **getInstance**

public static final SecretKeyFactory getInstance (String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException

**Parameters:**

algorithm – the standard name of the requested secret key algorithm, e.g. “AES”  
provider – the name of the provider

**Returns:**

A SecretKeyFactory object for the specified secret key algorithm.

**Throws:**

NoSuchAlgorithmException - if the algorithm is not available from the specified provider  
NoSuchProviderException -if the provider has not been configured.

❖ **generateSecret**

public final SecretKey generateSecret (KeySpec keySpec) throws InvalidKeySpecException

Generates a SecretKey object from the provided key specification (key material)

**Parameters:**

keySpec – the specification (key material) of the secret key

**Returns:**

the secret key

**Throws:**

InvalidKeySpecException - if the given key specification is inappropriate for this key factory to produce a secret key.

(3/19/98)

2.2. *ANSI C Specification*

The ANSI C key generation programming interface uses two structures and a set of functions to generate and manipulate key. The first structure, *keyInstance*, is an algorithm independent structure that shall remain unchanged for all submissions. The second structure, *keyParams*, is algorithm dependent and is used to contain any algorithm-specific key material or information that is necessary. **All**

**implementations must be sure to document any algorithm-specific parameters and their use.**

```
➤ typedef struct {
    int          opmode;
    int          keyLen;
    BYTE        *keyMaterial; // BYTE is defined as an unsigned char
    KeyParams   params;
} keyInstance;
```

```
➤ typedef struct {
    BYTE        *randomSeed;
} keyParams;
```

### ❖ **keyInit**

```
int keyInit(BYTE *params, BYTE *random, keyInstance *key)
```

Initializes a keyInstance with the following information:

- keyLen: The key length (128, 192, 256, and possibly more) of the key,
- opmode: ENCRYPT\_MODE or DECRYPT\_MODE, and
- Other parameters, if present, e.g., keyMaterial, random, etc.

#### **Parameters:**

params: a string that has a byte that indicates if the key is being set for encryption or decryption (opmode: Encrypt=1, Decrypt=2), three ASCII bytes for the key size() (keyLen), followed by blocksize/4 ASCII characters representing the hex values for the key (key value may not be present) (keyMaterial). For example, 21281234567890abcdef1234567890abcdef" is the params string for a key for decrypting of length 128 bits with the binary value:  
000100100011010001010110011110001001101010111100  
11011110...

The format of the "params" string IS IMPLEMENTATION SPECIFIC, some algorithms will require different parameters. However, leave the first 4 bytes alone. They should remain one byte for opmode and 3 bytes for the key length. If you need a longer key size than 3 digits, place zeros in the length field and put the actual length in the implementation specific area. SPECIFY THE USE OF THIS STRING FOR YOUR IMPLEMENTATION.

random: an optional seed value for the generation of a random key.

**Returns:**

TRUE - on success

BAD\_KEY\_PARAMS - params structure not valid for this cipher

BAD\_KEY\_MAT - keyMaterial is invalid (e.g., wrong length)

❖ **keyGenerateKey**

```
int keyGenerateKey(keyInstance *key)
```

Generates a random key (as opposed to keyGenerateSecret, which is used to load a known value).

This will most likely use the contents of key->params->randomSeed as a seed to an algorithm for generating a pseudo-random secret key. Then any algorithm specific setup must be performed, e.g., initialization of key schedule in DES. If you can save the raw key in key->keyMaterial you might as well do so.

**Returns:**

TRUE - on success

FALSE - on failure

❖ **keyGenerateSecret**

```
int keyGenerateSecret(keyInstance *key)
```

Generates a key from the information contained in key->keyMaterial. This is used to load a known key value, e.g., during the Known Answer Test.

keyInstance should already have the keyMaterial loaded. Use this value to perform any algorithm specific setup that is necessary, e.g., initialization of key schedule in DES.

**Returns:**

TRUE - on success

BAD\_KEY\_SPEC - didn't include keyMaterial to generate a key

### ❖ **keyGetKey**

```
int keyGetKey(keyInstance *key, BYTE *retBuffer)
```

Returns the raw AES key.

#### **Parameters:**

key: the keyInstance in which the keyMaterial is contained.  
retBuffer: the buffer in which the raw key is returned.

#### **Returns:**

TRUE - on success  
BAD\_KEY\_INSTANCE - key is invalid (e.g., not initialized)

## **3. Cipher Object Interface**

### *3.1. Java Specification*

(3/19/98) The NIST test code will instantiate objects of type Cipher by calling Cipher's getInstance method with the appropriate transformation string and provider name. The transformation string shall be of the form "AES/<mode>/NoPadding", where <mode> is replaced with "ECB", "CBC", or "CFB" as appropriate. "CFB" shall indicate 1-bit cipher feedback specifically. The Monte Carlo and Known Answer tests specified by NIST do not require cryptographic service provider packages to implement padding schemes.

- Class javax.crypto.Cipher

```
public class Cipher  
extends Object
```

### ❖ **getInstance**

```
public static final Cipher getInstance(String transformation, String  
provider) throws NoSuchAlgorithmException, NoSuchProviderException,  
NoSuchPaddingException
```

Creates a Cipher object that implements the specified transformation, as supplied by the specified provider.

#### **Parameters:**

transformation - the string representation of the requested algorithm, as described above

provider - the name of the cipher provider

**Returns:**

a cipher that implements the requested algorithm

**Throws:**

NoSuchAlgorithmException - if the requested algorithm is not available

NoSuchProviderException - if the requested provider is not available

NoSuchPaddingException - if the requested padding is not available

❖ **Init**

```
public final void Init (int opmode, Key key, AlgorithmParameterSpec  
params, SecureRandom random) throws InvalidKeyException,  
InvalidAlgorithmParameterException
```

Initializes this cipher with a key, a set of algorithm parameters, and a source of randomness. The cipher is initialized for encryption or decryption, depending on the value of opmode.

If this cipher (including its underlying feedback or padding scheme) requires any random bytes, it will get them from random.

**Parameters:**

opmode - the operation mode of this cipher (this is either ENCRYPT\_MODE or DECRYPT\_MODE)

key - the encryption key

params - the algorithm parameters (implementation defined)

random - the source of randomness

**Throws:**

InvalidKeyException - if the given key is inappropriate for initializing this cipher

InvalidAlgorithmParameterException - if the given algorithm parameters are inappropriate for this cipher

❖ **Update**

public final byte[] Update (byte input[], int inputOffset, int inputLen)  
throws IllegalStateException

Continues a multiple-part encryption or decryption operation (depending on how this cipher was initialized), processing another data part.

The first inputLen bytes in the input buffer, starting at inputOffset, are processed, and the result is stored in a new buffer.

**Parameters:**

input - the input buffer

inputOffset - the offset in input where the input starts

inputLen - the input length

**Returns:**

the new buffer with the result, or null if the underlying cipher is a block cipher and the input data is too short to result in a new block.

**Throws:**

IllegalStateException - if this cipher is in a wrong state (e.g., has not been initialized)

❖ **DoFinal**

public final int DoFinal (byte input[], int inputOffset, int inputLen) throws  
IllegalBlockSizeException, BadPaddingException

Encrypts or decrypts data in a single-part operation, or finishes a multiple-part operation. The data is encrypted or decrypted, depending on how this cipher was initialized.

The first inputLen bytes in the input buffer, starting at inputOffset, and any input bytes that may have been buffered during a previous update operation, are processed, with padding (if requested) being applied. The result is stored in a new buffer.

**Parameters:**

input - the input buffer

inputOffset - the offset in input where the input starts

inputLen - the input length

**Returns:**

the new buffer with the result

**Throws:**

IllegalStateException - if this cipher is in a wrong state (e.g., has not been initialized)

IllegalBlockSizeException - if this cipher is a block cipher, no padding has been requested (only in encryption mode), and the total input length of the data processed by this cipher is not a multiple of block size

BadPaddingException - if this cipher is in decryption mode, and (un)padding has been requested, but the decrypted data is not bounded by the appropriate padding bytes

**❖ GetOutputSize**

public final int getOutputSize (int inputLen) throws IllegalStateException

Returns the length in bytes that an output buffer would need to be in order to hold the result of the next update or doFinal operation, given the input length inputLen (in bytes).

This call takes into account any unprocessed (buffered) data from a previous update call, and padding.

The actual output length of the next update or doFinal call may be smaller than the length returned by this method.

**Parameters:**

inputLen - the input length (in bytes)

**Returns:**

the required output buffer size (in bytes)

**Throws:**

IllegalStateException - if this cipher is in a wrong state (e.g., has not yet been initialized)

(3/19/98)

### 3.2. ANSI C Specification

The ANSI C cipher programming interface uses two structures and a set of functions to manipulate cipher data. The first structure, *cipherInstance*, is an

algorithm independent structure that shall remain unchanged for all submissions. The second structure, *cipherParams*, is algorithm dependent and is used to contain any algorithm-specific cipher data or information that is necessary. **All implementations must be sure to document any algorithm-specific parameters and their use.**

- typedef struct {  
    BYTE mode;          // ENCRYPT\_MODE or DECRYPT\_MODE  
    int  numBytes;      // Number of bytes processed by cipher  
    int  bufLen;        // Number of bytes in the buffer  
    BYTE buffer[BLOCKSIZE]; // Unprocessed data  
    cipherParams params;  // Algorithm specific information  
} cipherInstance;
  
- typedef struct {  
    } cipherParams;

#### ❖ cipherInit

```
int cipherInit(cipherInstance *cipher, keyInstance *key, BYTE opmode,  
cipherParams *params, BYTE *random)
```

Initializes the cipher with the opmode and sets the buffer information to empty. If any algorithm specific setup is necessary take care of that as well.

#### Parameters:

*cipher* – the cipherInstance being loaded  
*key* – the ciphering key  
*opmode* - the operation mode of this cipher (this is either ENCRYPT\_MODE or DECRYPT\_MODE)  
*params* - the algorithm parameters (implementation defined)  
*random* - the source of randomness

#### Returns:

TRUE - on success  
BAD\_CIPHER\_KEY - the key passed does not agree with this cipher (e.g., bad opmode)  
BAD\_CIPHER\_PARAMS - the params struct is invalid for this cipher

#### ❖ cipherUpdate

```
int cipherUpdate(cipherInstance *cipher, keyInstance *key, BYTE *input,
int inputOffset, int inputLen, BYTE *outBuffer)
```

Uses the cipherInstance object and the keyInstance object to encrypt or decrypt the data in the "input" buffer. This data need not be an integral block unit (128 bits). The output (either the encrypted or decrypted data) is return in outBuffer.

**Parameters:**

cipher – the cipherInstance to be used

key – the ciphering key

input - the input buffer

inputOffset - the offset in input where the input starts

inputLen - the input length

outBuffer – contains the ciphered data

**Returns:**

TRUE - on success

BAD\_CIPHER\_STATE - cipher in bad state (e.g., not initialized)

❖ **cipherDoFinal**

```
int cipherDoFinal(cipherInstance *cipher, keyInstance *key, BYTE *input,
int inputOffset, int inputLen, BYTE *outBuffer)
```

Uses the cipherInstance object and the keyInstance object to encrypt or decrypt the data in the "input" buffer. This data need not be an integral block unit (128 bits). Since this is the final call for an encryption or decryption operation, a non-integral block unit is padded to a full block and the ciphered. The output (either the encrypted or decrypted data) is return in outBuffer.

**Parameters:**

cipher – the cipherInstance to be used

key – the ciphering key

input - the input buffer

inputOffset - the offset in input where the input starts

inputLen - the input length

outBuffer – contains the ciphered data

**Returns:**

TRUE - on success

BAD\_CIPHER\_STATE - cipher in bad state (e.g., not initialized)

BAD\_CIPHER\_BLOCK - cipher is block, but didn't receive full block and no padding requested

❖ **cipherGetOutputSize**

```
int cipherGetOutputSize(cipherInstance *cipher, int inputLen)
```

Returns the number of bytes necessary to hold the output of a call to cipherUpdate or cipherFinal. Actual call may be less than this value.

**Parameters:**

inputLen – the length of input in bytes

**Returns:**

The max expected buffer size, or  
BAD\_CIPHER\_STATE - cipher in bad state (e.g., not initialized)