NAME

EVP_CIPHER_fetch, EVP_CIPHER_up_ref, EVP_CIPHER_free, EVP_CIPHER_CTX_new, EVP_CIPHER_CTX_reset, EVP_CIPHER_CTX_free, EVP_EncryptInit_ex, EVP_EncryptInit_ex2, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptInit_ex2, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherInit_ex2, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_Cipher, EVP_get_cipherbyname, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_is_a, EVP_CIPHER_get0_name, EVP_CIPHER_get0_description, EVP_CIPHER_names_do_all, EVP_CIPHER_get0_provider, EVP_CIPHER_get_nid, EVP_CIPHER_get_params, EVP_CIPHER_gettable_params, EVP_CIPHER_get_block_size, EVP_CIPHER_get_key_length, EVP_CIPHER_get_iv_length, EVP_CIPHER_get_flags, EVP_CIPHER_get_mode, EVP_CIPHER_get_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_get0_cipher, EVP_CIPHER_CTX_get1_cipher, EVP_CIPHER_CTX_get0_name, EVP_CIPHER_CTX_get_nid, EVP_CIPHER_CTX_get_params, EVP_CIPHER_gettable_ctx_params, EVP_CIPHER_CTX_gettable_params, EVP_CIPHER_CTX_set_params, EVP_CIPHER_settable_ctx_params, EVP_CIPHER_CTX_settable_params, EVP_CIPHER_CTX_get_block_size, EVP_CIPHER_CTX_get_key_length, EVP_CIPHER_CTX_get_iv_length, EVP_CIPHER_CTX_get_tag_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_set_flags, EVP_CIPHER_CTX_clear_flags, EVP_CIPHER_CTX_test_flags, EVP_CIPHER_CTX_get_type, EVP_CIPHER_CTX_get_mode, EVP_CIPHER_CTX_get_num, EVP_CIPHER_CTX_set_num, EVP_CIPHER_CTX_is_encrypting, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param, EVP_CIPHER_CTX_set_padding, EVP_enc_null, EVP_CIPHER_do_all_provided, EVP_CIPHER_nid, EVP_CIPHER_name, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_encrypting, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_tag_length, EVP_CIPHER_CTX_num, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_mode - EVP cipher routines

SYNOPSIS

```
#include <openssl/evp.h>

EVP_CIPHER *EVP_CIPHER_fetch(OSSL_LIB_CTX *ctx, const char *algorithm,
                             const char *properties);
int EVP_CIPHER_up_ref(EVP_CIPHER *cipher);
void EVP_CIPHER_free(EVP_CIPHER *cipher);
EVP_CIPHER_CTX *EVP_CIPHER_CTX_new(void);
```

```
int EVP_CIPHER_CTX_reset(EVP_CIPHER_CTX *ctx);
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *ctx);

int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, const unsigned char *key, const unsigned char *iv);
int EVP_EncryptInit_ex2(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                        const unsigned char *key, const unsigned char *iv,
                        const OSSL_PARAM params[]);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                      int *outl, const unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);

int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, const unsigned char *key, const unsigned char *iv);
int EVP_DecryptInit_ex2(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                        const unsigned char *key, const unsigned char *iv,
                        const OSSL_PARAM params[]);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                      int *outl, const unsigned char *in, int inl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);

int EVP_CipherInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                      ENGINE *impl, const unsigned char *key, const unsigned char *iv, int enc);
int EVP_CipherInit_ex2(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       const unsigned char *key, const unsigned char *iv,
                       int enc, const OSSL_PARAM params[]);
int EVP_CipherUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                     int *outl, const unsigned char *in, int inl);
int EVP_CipherFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);

int EVP_EncryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                    const unsigned char *key, const unsigned char *iv);
int EVP_EncryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);

int EVP_DecryptInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                    const unsigned char *key, const unsigned char *iv);
int EVP_DecryptFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);

int EVP_CipherInit(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                   const unsigned char *key, const unsigned char *iv, int enc);
```

```
int EVP_CipherFinal(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);

int EVP_Cipher(EVP_CIPHER_CTX *ctx, unsigned char *out,
               const unsigned char *in, unsigned int inl);

int EVP_CIPHER_CTX_set_padding(EVP_CIPHER_CTX *x, int padding);
int EVP_CIPHER_CTX_set_key_length(EVP_CIPHER_CTX *x, int keylen);
int EVP_CIPHER_CTX_ctrl(EVP_CIPHER_CTX *ctx, int cmd, int p1, void *p2);
int EVP_CIPHER_CTX_rand_key(EVP_CIPHER_CTX *ctx, unsigned char *key);
void EVP_CIPHER_CTX_set_flags(EVP_CIPHER_CTX *ctx, int flags);
void EVP_CIPHER_CTX_clear_flags(EVP_CIPHER_CTX *ctx, int flags);
int EVP_CIPHER_CTX_test_flags(const EVP_CIPHER_CTX *ctx, int flags);

const EVP_CIPHER *EVP_get_cipherbyname(const char *name);
const EVP_CIPHER *EVP_get_cipherbynid(int nid);
const EVP_CIPHER *EVP_get_cipherbyobj(const ASN1_OBJECT *a);

int EVP_CIPHER_get_nid(const EVP_CIPHER *e);
int EVP_CIPHER_is_a(const EVP_CIPHER *cipher, const char *name);
int EVP_CIPHER_names_do_all(const EVP_CIPHER *cipher,
                            void (*fn)(const char *name, void *data),
                            void *data);
const char *EVP_CIPHER_get0_name(const EVP_CIPHER *cipher);
const char *EVP_CIPHER_get0_description(const EVP_CIPHER *cipher);
const OSSL_PROVIDER *EVP_CIPHER_get0_provider(const EVP_CIPHER *cipher);
int EVP_CIPHER_get_block_size(const EVP_CIPHER *e);
int EVP_CIPHER_get_key_length(const EVP_CIPHER *e);
int EVP_CIPHER_get_iv_length(const EVP_CIPHER *e);
unsigned long EVP_CIPHER_get_flags(const EVP_CIPHER *e);
unsigned long EVP_CIPHER_get_mode(const EVP_CIPHER *e);
int EVP_CIPHER_get_type(const EVP_CIPHER *cipher);

const EVP_CIPHER *EVP_CIPHER_CTX_get0_cipher(const EVP_CIPHER_CTX *ctx);
EVP_CIPHER *EVP_CIPHER_CTX_get1_cipher(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_nid(const EVP_CIPHER_CTX *ctx);
const char *EVP_CIPHER_CTX_get0_name(const EVP_CIPHER_CTX *ctx);

int EVP_CIPHER_get_params(EVP_CIPHER *cipher, OSSL_PARAM params[]);
int EVP_CIPHER_CTX_set_params(EVP_CIPHER_CTX *ctx, const OSSL_PARAM params[]);
int EVP_CIPHER_CTX_get_params(EVP_CIPHER_CTX *ctx, OSSL_PARAM params[]);
```

```
const OSSL_PARAM *EVP_CIPHER_gettable_params(const EVP_CIPHER *cipher);
const OSSL_PARAM *EVP_CIPHER_settable_ctx_params(const EVP_CIPHER *cipher);
const OSSL_PARAM *EVP_CIPHER_gettable_ctx_params(const EVP_CIPHER *cipher);
const OSSL_PARAM *EVP_CIPHER_CTX_settable_params(EVP_CIPHER_CTX *ctx);
const OSSL_PARAM *EVP_CIPHER_CTX_gettable_params(EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_block_size(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_key_length(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_iv_length(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_tag_length(const EVP_CIPHER_CTX *ctx);
void *EVP_CIPHER_CTX_get_app_data(const EVP_CIPHER_CTX *ctx);
void EVP_CIPHER_CTX_set_app_data(const EVP_CIPHER_CTX *ctx, void *data);
int EVP_CIPHER_CTX_get_type(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_mode(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_get_num(const EVP_CIPHER_CTX *ctx);
int EVP_CIPHER_CTX_set_num(EVP_CIPHER_CTX *ctx, int num);
int EVP_CIPHER_CTX_is_encrypting(const EVP_CIPHER_CTX *ctx);

int EVP_CIPHER_param_to_asn1(EVP_CIPHER_CTX *c, ASN1_TYPE *type);
int EVP_CIPHER_asn1_to_param(EVP_CIPHER_CTX *c, ASN1_TYPE *type);

void EVP_CIPHER_do_all_provided(OSSL_LIB_CTX *libctx,
                                void (*fn)(EVP_CIPHER *cipher, void *arg),
                                void *arg);

#define EVP_CIPHER_nid EVP_CIPHER_get_nid
#define EVP_CIPHER_name EVP_CIPHER_get0_name
#define EVP_CIPHER_block_size EVP_CIPHER_get_block_size
#define EVP_CIPHER_key_length EVP_CIPHER_get_key_length
#define EVP_CIPHER_iv_length EVP_CIPHER_get_iv_length
#define EVP_CIPHER_flags EVP_CIPHER_get_flags
#define EVP_CIPHER_mode EVP_CIPHER_get_mode
#define EVP_CIPHER_type EVP_CIPHER_get_type
#define EVP_CIPHER_CTX_encrypting EVP_CIPHER_CTX_is_encrypting
#define EVP_CIPHER_CTX_nid EVP_CIPHER_CTX_get_nid
#define EVP_CIPHER_CTX_block_size EVP_CIPHER_CTX_get_block_size
#define EVP_CIPHER_CTX_key_length EVP_CIPHER_CTX_get_key_length
#define EVP_CIPHER_CTX_iv_length EVP_CIPHER_CTX_get_iv_length
#define EVP_CIPHER_CTX_tag_length EVP_CIPHER_CTX_get_tag_length
#define EVP_CIPHER_CTX_num EVP_CIPHER_CTX_get_num
#define EVP_CIPHER_CTX_type EVP_CIPHER_CTX_get_type
```

```
#define EVP_CIPHER_CTX_mode EVP_CIPHER_CTX_get_mode
```

The following function has been deprecated since OpenSSL 3.0, and can be hidden entirely by defining **OPENSSL_API_COMPAT** with a suitable version value, see **openssl_user_macros**(7):

```
const EVP_CIPHER *EVP_CIPHER_CTX_cipher(const EVP_CIPHER_CTX *ctx);
```

The following function has been deprecated since OpenSSL 1.1.0, and can be hidden entirely by defining **OPENSSL_API_COMPAT** with a suitable version value, see **openssl_user_macros**(7):

```
int EVP_CIPHER_CTX_flags(const EVP_CIPHER_CTX *ctx);
```

DESCRIPTION

The EVP cipher routines are a high-level interface to certain symmetric ciphers.

The **EVP_CIPHER** type is a structure for cipher method implementation.

EVP_CIPHER_fetch()

Fetches the cipher implementation for the given *algorithm* from any provider offering it, within the criteria given by the *properties*. See "ALGORITHM FETCHING" in **crypto**(7) for further information.

The returned value must eventually be freed with **EVP_CIPHER_free()**.

Fetched **EVP_CIPHER** structures are reference counted.

EVP_CIPHER_up_ref()

Increments the reference count for an **EVP_CIPHER** structure.

EVP_CIPHER_free()

Decrements the reference count for the fetched **EVP_CIPHER** structure. If the reference count drops to 0 then the structure is freed.

EVP_CIPHER_CTX_new()

Allocates and returns a cipher context.

EVP_CIPHER_CTX_free()

Clears all information from a cipher context and frees any allocated memory associated with it, including *ctx* itself. This function should be called after all operations using a cipher are complete so sensitive information does not remain in memory.

EVP_CIPHER_CTX_ctrl()

This is a legacy method. **EVP_CIPHER_CTX_set_params()** and **EVP_CIPHER_CTX_get_params()** is the mechanism that should be used to set and get parameters that are used by providers.

Performs cipher-specific control actions on context *ctx*. The control command is indicated in *cmd* and any additional arguments in *p1* and *p2*. **EVP_CIPHER_CTX_ctrl()** must be called after **EVP_CipherInit_ex2()**. Other restrictions may apply depending on the control type and cipher implementation.

If this function happens to be used with a fetched **EVP_CIPHER**, it will translate the controls that are known to OpenSSL into **OSSL_PARAM**(3) parameters with keys defined by OpenSSL and call **EVP_CIPHER_CTX_get_params()** or **EVP_CIPHER_CTX_set_params()** as is appropriate for each control command.

See "CONTROLS" below for more information, including what translations are being done.

EVP_CIPHER_get_params()

Retrieves the requested list of algorithm *params* from a CIPHER *cipher*. See "PARAMETERS" below for more information.

EVP_CIPHER_CTX_get_params()

Retrieves the requested list of *params* from CIPHER context *ctx*. See "PARAMETERS" below for more information.

EVP_CIPHER_CTX_set_params()

Sets the list of *params* into a CIPHER context *ctx*. See "PARAMETERS" below for more information.

EVP_CIPHER_gettable_params()

Get a constant **OSSL_PARAM**(3) array that describes the retrievable parameters that can be used with **EVP_CIPHER_get_params()**.

EVP_CIPHER_gettable_ctx_params() and **EVP_CIPHER_CTX_gettable_params()**

Get a constant **OSSL_PARAM**(3) array that describes the retrievable parameters that can be used with **EVP_CIPHER_CTX_get_params()**. **EVP_CIPHER_gettable_ctx_params()** returns the parameters that can be retrieved from the algorithm, whereas **EVP_CIPHER_CTX_gettable_params()** returns the parameters that can be retrieved in the context's current state.

EVP_CIPHER_settable_ctx_params() and **EVP_CIPHER_CTX_settable_params()**

Get a constant **OSSL_PARAM**(3) array that describes the settable parameters that can be used with **EVP_CIPHER_CTX_set_params()**. **EVP_CIPHER_settable_ctx_params()** returns the parameters that can be set from the algorithm, whereas **EVP_CIPHER_CTX_settable_params()** returns the parameters that can be set in the context's current state.

EVP_EncryptInit_ex2()

Sets up cipher context *ctx* for encryption with cipher *type*. *type* is typically supplied by calling **EVP_CIPHER_fetch()**. *type* may also be set using legacy functions such as **EVP_aes_256_cbc()**, but this is not recommended for new applications. *key* is the symmetric key to use and *iv* is the IV to use (if necessary), the actual number of bytes used for the key and IV depends on the cipher. The parameters *params* will be set on the context after initialisation. It is possible to set all parameters to NULL except *type* in an initial call and supply the remaining parameters in subsequent calls, all of which have *type* set to NULL. This is done when the default cipher parameters are not appropriate. For **EVP_CIPH_GCM_MODE** the IV will be generated internally if it is not specified.

EVP_EncryptInit_ex()

This legacy function is similar to **EVP_EncryptInit_ex2()** when *impl* is NULL. The implementation of the *type* from the *impl* engine will be used if it exists.

EVP_EncryptUpdate()

Encrypts *inl* bytes from the buffer *in* and writes the encrypted version to *out*. This function can be called multiple times to encrypt successive blocks of data. The amount of data written depends on the block alignment of the encrypted data. For most ciphers and modes, the amount of data written can be anything from zero bytes to (inl + cipher_block_size - 1) bytes. For wrap cipher modes, the amount of data written can be anything from zero bytes to (inl + cipher_block_size) bytes. For stream ciphers, the amount of data written can be anything from zero bytes to inl bytes. Thus, *out* should contain sufficient room for the operation being performed. The actual number of bytes written is placed in *outl*. It also checks if *in* and *out* are partially overlapping, and if they are 0 is returned to indicate failure.

If padding is enabled (the default) then **EVP_EncryptFinal_ex()** encrypts the "final" data, that is any data that remains in a partial block. It uses standard block padding (aka PKCS padding) as described in the NOTES section, below. The encrypted final data is written to *out* which should have sufficient space for one cipher block. The number of bytes written is placed in *outl*. After this function is called the encryption operation is finished and no further calls to **EVP_EncryptUpdate()** should be made.

If padding is disabled then **EVP_EncryptFinal_ex()** will not encrypt any more data and it will

return an error if any data remains in a partial block: that is if the total data length is not a multiple of the block size.

EVP_DecryptInit_ex2(), **EVP_DecryptInit_ex()**, **EVP_DecryptUpdate()** and **EVP_DecryptFinal_ex()**
These functions are the corresponding decryption operations. **EVP_DecryptFinal()** will return an error code if padding is enabled and the final block is not correctly formatted. The parameters and restrictions are identical to the encryption operations except that if padding is enabled the decrypted data buffer *out* passed to **EVP_DecryptUpdate()** should have sufficient room for (*inl* + cipher_block_size) bytes unless the cipher block size is 1 in which case *inl* bytes is sufficient.

EVP_CipherInit_ex2(), **EVP_CipherInit_ex()**, **EVP_CipherUpdate()** and **EVP_CipherFinal_ex()**
These functions can be used for decryption or encryption. The operation performed depends on the value of the *enc* parameter. It should be set to 1 for encryption, 0 for decryption and -1 to leave the value unchanged (the actual value of 'enc' being supplied in a previous call).

EVP_CIPHER_CTX_reset()
Clears all information from a cipher context and free up any allocated memory associated with it, except the *ctx* itself. This function should be called anytime *ctx* is reused by another **EVP_CipherInit()** / **EVP_CipherUpdate()** / **EVP_CipherFinal()** series of calls.

EVP_EncryptInit(), **EVP_DecryptInit()** and **EVP_CipherInit()**
Behave in a similar way to **EVP_EncryptInit_ex()**, **EVP_DecryptInit_ex()** and **EVP_CipherInit_ex()** except if the *type* is not a fetched cipher they use the default implementation of the *type*.

EVP_EncryptFinal(), **EVP_DecryptFinal()** and **EVP_CipherFinal()**
Identical to **EVP_EncryptFinal_ex()**, **EVP_DecryptFinal_ex()** and **EVP_CipherFinal_ex()**. In previous releases they also cleaned up the *ctx*, but this is no longer done and **EVP_CIPHER_CTX_cleanup()** must be called to free any context resources.

EVP_Cipher()
Encrypts or decrypts a maximum *inl* amount of bytes from *in* and leaves the result in *out*.

For legacy ciphers - If the cipher doesn't have the flag **EVP_CIPH_FLAG_CUSTOM_CIPHER** set, then *inl* must be a multiple of **EVP_CIPHER_get_block_size()**. If it isn't, the result is undefined. If the cipher has that flag set, then *inl* can be any size.

Due to the constraints of the API contract of this function it shouldn't be used in applications, please consider using **EVP_CipherUpdate()** and **EVP_CipherFinal_ex()** instead.

EVP_get_cipherbyname(), **EVP_get_cipherbynid()** and **EVP_get_cipherbyobj()**
Returns an **EVP_CIPHER** structure when passed a cipher name, a cipher **NID** or an **ASN1_OBJECT** structure respectively.

EVP_get_cipherbyname() will return NULL for algorithms such as "AES-128-SIV", "AES-128-CBC-CTS" and "CAMELLIA-128-CBC-CTS" which were previously only accessible via low level interfaces.

The **EVP_get_cipherbyname()** function is present for backwards compatibility with OpenSSL prior to version 3 and is different to the **EVP_CIPHER_fetch()** function since it does not attempt to "fetch" an implementation of the cipher. Additionally, it only knows about ciphers that are built-in to OpenSSL and have an associated NID. Similarly **EVP_get_cipherbynid()** and **EVP_get_cipherbyobj()** also return objects without an associated implementation.

When the cipher objects returned by these functions are used (such as in a call to **EVP_EncryptInit_ex()**) an implementation of the cipher will be implicitly fetched from the loaded providers. This fetch could fail if no suitable implementation is available. Use **EVP_CIPHER_fetch()** instead to explicitly fetch the algorithm and an associated implementation from a provider.

See "ALGORITHM FETCHING" in **crypto**(7) for more information about fetching.

The cipher objects returned from these functions do not need to be freed with **EVP_CIPHER_free()**.

EVP_CIPHER_get_nid() and **EVP_CIPHER_CTX_get_nid()**
Return the NID of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The actual NID value is an internal value which may not have a corresponding OBJECT IDENTIFIER.

EVP_CIPHER_CTX_set_flags(), **EVP_CIPHER_CTX_clear_flags()** and **EVP_CIPHER_CTX_test_flags()**
Sets, clears and tests *ctx* flags. See "FLAGS" below for more information.

For provided ciphers **EVP_CIPHER_CTX_set_flags()** should be called only after the fetched cipher has been assigned to the *ctx*. It is recommended to use "PARAMETERS" instead.

EVP_CIPHER_CTX_set_padding()
Enables or disables padding. This function should be called after the context is set up for encryption or decryption with **EVP_EncryptInit_ex2()**, **EVP_DecryptInit_ex2()** or

EVP_CipherInit_ex2(). By default encryption operations are padded using standard block padding and the padding is checked and removed when decrypting. If the *pad* parameter is zero then no padding is performed, the total amount of data encrypted or decrypted must then be a multiple of the block size or an error will occur.

EVP_CIPHER_get_key_length() and **EVP_CIPHER_CTX_get_key_length()**
Return the key length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_KEY_LENGTH** is the maximum key length for all ciphers. Note: although **EVP_CIPHER_get_key_length()** is fixed for a given cipher, the value of **EVP_CIPHER_CTX_get_key_length()** may be different for variable key length ciphers.

EVP_CIPHER_CTX_set_key_length()
Sets the key length of the cipher context. If the cipher is a fixed length cipher then attempting to set the key length to any value other than the fixed value is an error.

EVP_CIPHER_get_iv_length() and **EVP_CIPHER_CTX_get_iv_length()**
Return the IV length of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX**. It will return zero if the cipher does not use an IV. The constant **EVP_MAX_IV_LENGTH** is the maximum IV length for all ciphers.

EVP_CIPHER_CTX_get_tag_length()
Returns the tag length of an AEAD cipher when passed a **EVP_CIPHER_CTX**. It will return zero if the cipher does not support a tag. It returns a default value if the tag length has not been set.

EVP_CIPHER_get_block_size() and **EVP_CIPHER_CTX_get_block_size()**
Return the block size of a cipher when passed an **EVP_CIPHER** or **EVP_CIPHER_CTX** structure. The constant **EVP_MAX_BLOCK_LENGTH** is also the maximum block length for all ciphers.

EVP_CIPHER_get_type() and **EVP_CIPHER_CTX_get_type()**
Return the type of the passed cipher or context. This "type" is the actual NID of the cipher OBJECT IDENTIFIER and as such it ignores the cipher parameters (40 bit RC2 and 128 bit RC2 have the same NID). If the cipher does not have an object identifier or does not have ASN1 support this function will return **NID_undef**.

EVP_CIPHER_is_a()
Returns 1 if *cipher* is an implementation of an algorithm that's identifiable with *name*, otherwise 0. If *cipher* is a legacy cipher (it's the return value from the likes of **EVP_aes128()** rather than the result of an **EVP_CIPHER_fetch()**), only cipher names registered with the default library context (see **OSSL_LIB_CTX**(3)) will be considered.

EVP_CIPHER_get0_name() and **EVP_CIPHER_CTX_get0_name()**
Return the name of the passed cipher or context. For fetched ciphers with multiple names, only one of them is returned. See also **EVP_CIPHER_names_do_all()**.

EVP_CIPHER_names_do_all()
Traverses all names for the *cipher*, and calls *fn* with each name and *data*. This is only useful with fetched **EVP_CIPHER**s.

EVP_CIPHER_get0_description()
Returns a description of the cipher, meant for display and human consumption. The description is at the discretion of the cipher implementation.

EVP_CIPHER_get0_provider()
Returns an **OSSL_PROVIDER** pointer to the provider that implements the given **EVP_CIPHER**.

EVP_CIPHER_CTX_get0_cipher()
Returns the **EVP_CIPHER** structure when passed an **EVP_CIPHER_CTX** structure. **EVP_CIPHER_CTX_get1_cipher()** is the same except the ownership is passed to the caller.

EVP_CIPHER_get_mode() and **EVP_CIPHER_CTX_get_mode()**
Return the block cipher mode: EVP_CIPH_ECB_MODE, EVP_CIPH_CBC_MODE, EVP_CIPH_CFB_MODE, EVP_CIPH_OFB_MODE, EVP_CIPH_CTR_MODE, EVP_CIPH_GCM_MODE, EVP_CIPH_CCM_MODE, EVP_CIPH_XTS_MODE, EVP_CIPH_WRAP_MODE, EVP_CIPH_OCB_MODE or EVP_CIPH_SIV_MODE. If the cipher is a stream cipher then EVP_CIPH_STREAM_CIPHER is returned.

EVP_CIPHER_get_flags()
Returns any flags associated with the cipher. See "FLAGS" for a list of currently defined flags.

EVP_CIPHER_CTX_get_num() and **EVP_CIPHER_CTX_set_num()**
Gets or sets the cipher specific "num" parameter for the associated *ctx*. Built-in ciphers typically use this to track how much of the current underlying block has been "used" already.

EVP_CIPHER_CTX_is_encrypting()
Reports whether the *ctx* is being used for encryption or decryption.

EVP_CIPHER_CTX_flags()
A deprecated macro calling "EVP_CIPHER_get_flags(EVP_CIPHER_CTX_get0_cipher(ctx))". Do not use.

EVP_CIPHER_param_to_asn1()

Sets the AlgorithmIdentifier "parameter" based on the passed cipher. This will typically include any parameters and an IV. The cipher IV (if any) must be set when this call is made. This call should be made before the cipher is actually "used" (before any **EVP_EncryptUpdate()**, **EVP_DecryptUpdate()** calls for example). This function may fail if the cipher does not have any ASN1 support.

EVP_CIPHER_asn1_to_param()

Sets the cipher parameters based on an ASN1 AlgorithmIdentifier "parameter". The precise effect depends on the cipher. In the case of **RC2**, for example, it will set the IV and effective key length. This function should be called after the base cipher type is set but before the key is set. For example **EVP_CipherInit()** will be called with the IV and key set to NULL, **EVP_CIPHER_asn1_to_param()** will be called and finally **EVP_CipherInit()** again with all parameters except the key set to NULL. It is possible for this function to fail if the cipher does not have any ASN1 support or the parameters cannot be set (for example the RC2 effective key length is not supported.

EVP_CIPHER_CTX_rand_key()

Generates a random key of the appropriate length based on the cipher context. The **EVP_CIPHER** can provide its own random key generation routine to support keys of a specific form. *key* must point to a buffer at least as big as the value returned by **EVP_CIPHER_CTX_get_key_length()**.

EVP_CIPHER_do_all_provided()

Traverses all ciphers implemented by all activated providers in the given library context *libctx*, and for each of the implementations, calls the given function *fn* with the implementation method and the given *arg* as argument.

PARAMETERS

See **OSSL_PARAM**(3) for information about passing parameters.

Gettable EVP_CIPHER parameters

When **EVP_CIPHER_fetch()** is called it internally calls **EVP_CIPHER_get_params()** and caches the results.

EVP_CIPHER_get_params() can be used with the following **OSSL_PARAM**(3) keys:

"mode" (**OSSL_CIPHER_PARAM_MODE**) <unsigned integer>

Gets the mode for the associated cipher algorithm *cipher*. See "**EVP_CIPHER_get_mode()** and **EVP_CIPHER_CTX_get_mode()**" for a list of valid modes. Use **EVP_CIPHER_get_mode()** to retrieve the cached value.

"keylen" (**OSSL_CIPHER_PARAM_KEYLEN**) <unsigned integer>
: Gets the key length for the associated cipher algorithm *cipher*. Use **EVP_CIPHER_get_key_length()** to retrieve the cached value.

"ivlen" (**OSSL_CIPHER_PARAM_IVLEN**) <unsigned integer>
: Gets the IV length for the associated cipher algorithm *cipher*. Use **EVP_CIPHER_get_iv_length()** to retrieve the cached value.

"blocksize" (**OSSL_CIPHER_PARAM_BLOCK_SIZE**) <unsigned integer>
: Gets the block size for the associated cipher algorithm *cipher*. The block size should be 1 for stream ciphers. Note that the block size for a cipher may be different to the block size for the underlying encryption/decryption primitive. For example AES in CTR mode has a block size of 1 (because it operates like a stream cipher), even though AES has a block size of 16. Use **EVP_CIPHER_get_block_size()** to retrieve the cached value.

"aead" (**OSSL_CIPHER_PARAM_AEAD**) <integer>
: Gets 1 if this is an AEAD cipher algorithm, otherwise it gets 0. Use (EVP_CIPHER_get_flags(cipher) & EVP_CIPH_FLAG_AEAD_CIPHER) to retrieve the cached value.

"custom-iv" (**OSSL_CIPHER_PARAM_CUSTOM_IV**) <integer>
: Gets 1 if the cipher algorithm *cipher* has a custom IV, otherwise it gets 0. Storing and initializing the IV is left entirely to the implementation, if a custom IV is used. Use (EVP_CIPHER_get_flags(cipher) & EVP_CIPH_CUSTOM_IV) to retrieve the cached value.

"cts" (**OSSL_CIPHER_PARAM_CTS**) <integer>
: Gets 1 if the cipher algorithm *cipher* uses ciphertext stealing, otherwise it gets 0. This is currently used to indicate that the cipher is a one shot that only allows a single call to **EVP_CipherUpdate()**. Use (EVP_CIPHER_get_flags(cipher) & EVP_CIPH_FLAG_CTS) to retrieve the cached value.

"tls-multi" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK**) <integer>
: Gets 1 if the cipher algorithm *cipher* supports interleaving of crypto blocks, otherwise it gets 0. The interleaving is an optimization only applicable to certain TLS ciphers. Use (EVP_CIPHER_get_flags(cipher) & EVP_CIPH_FLAG_TLS1_1_MULTIBLOCK) to retrieve the cached value.

"has-randkey" (**OSSL_CIPHER_PARAM_HAS_RANDKEY**) <integer>
: Gets 1 if the cipher algorithm *cipher* supports the gettable EVP_CIPHER_CTX parameter **OSSL_CIPHER_PARAM_RANDOM_KEY**. Only DES and 3DES set this to 1, all other OpenSSL ciphers return 0.

Gettable and Settable EVP_CIPHER_CTX parameters

The following **OSSL_PARAM**(3) keys can be used with both **EVP_CIPHER_CTX_get_params()** and **EVP_CIPHER_CTX_set_params()**.

"padding" (**OSSL_CIPHER_PARAM_PADDING**) <unsigned integer>

Gets or sets the padding mode for the cipher context *ctx*. Padding is enabled if the value is 1, and disabled if the value is 0. See also **EVP_CIPHER_CTX_set_padding()**.

"num" (**OSSL_CIPHER_PARAM_NUM**) <unsigned integer>

Gets or sets the cipher specific "num" parameter for the cipher context *ctx*. Built-in ciphers typically use this to track how much of the current underlying block has been "used" already. See also **EVP_CIPHER_CTX_get_num()** and **EVP_CIPHER_CTX_set_num()**.

"keylen" (**OSSL_CIPHER_PARAM_KEYLEN**) <unsigned integer>

Gets or sets the key length for the cipher context *ctx*. The length of the "keylen" parameter should not exceed that of a **size_t**. See also **EVP_CIPHER_CTX_get_key_length()** and **EVP_CIPHER_CTX_set_key_length()**.

"tag" (**OSSL_CIPHER_PARAM_AEAD_TAG**) <octet string>

Gets or sets the AEAD tag for the associated cipher context *ctx*. See "AEAD Interface" in **EVP_EncryptInit**(3).

"keybits" (**OSSL_CIPHER_PARAM_RC2_KEYBITS**) <unsigned integer>

Gets or sets the effective keybits used for a RC2 cipher. The length of the "keybits" parameter should not exceed that of a **size_t**.

"rounds" (**OSSL_CIPHER_PARAM_ROUNDS**) <unsigned integer>

Gets or sets the number of rounds to be used for a cipher. This is used by the RC5 cipher.

"alg_id_param" (**OSSL_CIPHER_PARAM_ALGORITHM_ID_PARAMS**) <octet string>

Used to pass the DER encoded AlgorithmIdentifier parameter to or from the cipher implementation. Functions like **EVP_CIPHER_param_to_asn1**(3) and **EVP_CIPHER_asn1_to_param**(3) use this parameter for any implementation that has the flag **EVP_CIPH_FLAG_CUSTOM_ASN1** set.

"cts_mode" (**OSSL_CIPHER_PARAM_CTS_MODE**) <UTF8 string>

Gets or sets the cipher text stealing mode. For all modes the output size is the same as the input size. The input length must be greater than or equal to the block size. (The block size for AES and CAMELLIA is 16 bytes).

Valid values for the mode are:

"CS1"
The NIST variant of cipher text stealing. For input lengths that are multiples of the block size it is equivalent to using a "AES-XXX-CBC" or "CAMELLIA-XXX-CBC" cipher otherwise the second last cipher text block is a partial block.

"CS2"
For input lengths that are multiples of the block size it is equivalent to using a "AES-XXX-CBC" or "CAMELLIA-XXX-CBC" cipher, otherwise it is the same as "CS3" mode.

"CS3"
The Kerberos5 variant of cipher text stealing which always swaps the last cipher text block with the previous block (which may be a partial or full block depending on the input length). If the input length is exactly one full block then this is equivalent to using a "AES-XXX-CBC" or "CAMELLIA-XXX-CBC" cipher.

The default is "CS1". This is only supported for "AES-128-CBC-CTS", "AES-192-CBC-CTS", "AES-256-CBC-CTS", "CAMELLIA-128-CBC-CTS", "CAMELLIA-192-CBC-CTS" and "CAMELLIA-256-CBC-CTS".

"tls1multi_interleave" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_INTERLEAVE**) <unsigned integer>
Sets or gets the number of records being sent in one go for a tls1 multiblock cipher operation (either 4 or 8 records).

Gettable EVP_CIPHER_CTX parameters

The following **OSSL_PARAM**(3) keys can be used with **EVP_CIPHER_CTX_get_params()**:

"ivlen" (**OSSL_CIPHER_PARAM_IVLEN** and <**OSSL_CIPHER_PARAM_AEAD_IVLEN**) <unsigned integer>
Gets the IV length for the cipher context *ctx*. The length of the "ivlen" parameter should not exceed that of a **size_t**. See also **EVP_CIPHER_CTX_get_iv_length()**.

"iv" (**OSSL_CIPHER_PARAM_IV**) <octet string OR octet ptr>
Gets the IV used to initialize the associated cipher context *ctx*. See also **EVP_CIPHER_CTX_get_original_iv()**.

"updated-iv" (**OSSL_CIPHER_PARAM_UPDATED_IV**) <octet string OR octet ptr>
Gets the updated pseudo-IV state for the associated cipher context, e.g., the previous ciphertext

block for CBC mode or the iteratively encrypted IV value for OFB mode. Note that octet pointer access is deprecated and is provided only for backwards compatibility with historical libcrypto APIs. See also **EVP_CIPHER_CTX_get_updated_iv()**.

"randkey" (**OSSL_CIPHER_PARAM_RANDOM_KEY**) <octet string>
Gets an implementation specific randomly generated key for the associated cipher context *ctx*. This is currently only supported by DES and 3DES (which set the key to odd parity).

"taglen" (**OSSL_CIPHER_PARAM_AEAD_TAGLEN**) <unsigned integer>
Gets the tag length to be used for an AEAD cipher for the associated cipher context *ctx*. It gets a default value if it has not been set. The length of the "taglen" parameter should not exceed that of a **size_t**. See also **EVP_CIPHER_CTX_get_tag_length()**.

"tlsaadpad" (**OSSL_CIPHER_PARAM_AEAD_TLS1_AAD_PAD**) <unsigned integer>
Gets the length of the tag that will be added to a TLS record for the AEAD tag for the associated cipher context *ctx*. The length of the "tlsaadpad" parameter should not exceed that of a **size_t**.

"tlsivgen" (**OSSL_CIPHER_PARAM_AEAD_TLS1_GET_IV_GEN**) <octet string>
Gets the invocation field generated for encryption. Can only be called after "tlsivfixed" is set. This is only used for GCM mode.

"tls1multi_enclen" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC_LEN**) <unsigned integer>
Get the total length of the record returned from the "tls1multi_enc" operation.

"tls1multi_maxbufsz" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_MAX_BUFSIZE**) <unsigned integer>
Gets the maximum record length for a TLS1 multiblock cipher operation. The length of the "tls1multi_maxbufsz" parameter should not exceed that of a **size_t**.

"tls1multi_aadpacklen" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_AAD_PACKLEN**) <unsigned integer>
Gets the result of running the "tls1multi_aad" operation.

"tls-mac" (**OSSL_CIPHER_PARAM_TLS_MAC**) <octet ptr>
Used to pass the TLS MAC data.

Settable EVP_CIPHER_CTX parameters

The following **OSSL_PARAM**(3) keys can be used with **EVP_CIPHER_CTX_set_params()**:

"mackey" (**OSSL_CIPHER_PARAM_AEAD_MAC_KEY**) <octet string>
Sets the MAC key used by composite AEAD ciphers such as AES-CBC-HMAC-SHA256.

"speed" (**OSSL_CIPHER_PARAM_SPEED**) <unsigned integer>
Sets the speed option for the associated cipher context. This is only supported by AES SIV ciphers which disallow multiple operations by default. Setting "speed" to 1 allows another encrypt or decrypt operation to be performed. This is used for performance testing.

"use-bits" (**OSSL_CIPHER_PARAM_USE_BITS**) <unsigned integer>
Determines if the input length *inl* passed to **EVP_EncryptUpdate()**, **EVP_DecryptUpdate()** and **EVP_CipherUpdate()** is the number of bits or number of bytes. Setting "use-bits" to 1 uses bits. The default is in bytes. This is only used for **CFB1** ciphers.

This can be set using EVP_CIPHER_CTX_set_flags(ctx, EVP_CIPH_FLAG_LENGTH_BITS).

"tls-version" (**OSSL_CIPHER_PARAM_TLS_VERSION**) <integer>
Sets the TLS version.

"tls-mac-size" (**OSSL_CIPHER_PARAM_TLS_MAC_SIZE**) <unsigned integer>
Set the TLS MAC size.

"tlsaad" (**OSSL_CIPHER_PARAM_AEAD_TLS1_AAD**) <octet string>
Sets TLSv1.2 AAD information for the associated cipher context *ctx*. TLSv1.2 AAD information is always 13 bytes in length and is as defined for the "additional_data" field described in section 6.2.3.3 of RFC5246.

"tlsivfixed" (**OSSL_CIPHER_PARAM_AEAD_TLS1_IV_FIXED**) <octet string>
Sets the fixed portion of an IV for an AEAD cipher used in a TLS record encryption/ decryption for the associated cipher context. TLS record encryption/decryption always occurs "in place" so that the input and output buffers are always the same memory location. AEAD IVs in TLSv1.2 consist of an implicit "fixed" part and an explicit part that varies with every record. Setting a TLS fixed IV changes a cipher to encrypt/decrypt TLS records. TLS records are encrypted/decrypted using a single OSSL_FUNC_cipher_cipher call per record. For a record decryption the first bytes of the input buffer will be the explicit part of the IV and the final bytes of the input buffer will be the AEAD tag. The length of the explicit part of the IV and the tag length will depend on the cipher in use and will be defined in the RFC for the relevant ciphersuite. In order to allow for "in place" decryption the plaintext output should be written to the same location in the output buffer that the ciphertext payload was read from, i.e. immediately after the explicit IV.

When encrypting a record the first bytes of the input buffer should be empty to allow space for the

explicit IV, as will the final bytes where the tag will be written. The length of the input buffer will include the length of the explicit IV, the payload, and the tag bytes. The cipher implementation should generate the explicit IV and write it to the beginning of the output buffer, do "in place" encryption of the payload and write that to the output buffer, and finally add the tag onto the end of the output buffer.

Whether encrypting or decrypting the value written to **outl* in the OSSL_FUNC_cipher_cipher call should be the length of the payload excluding the explicit IV length and the tag length.

"tlsivinv" (**OSSL_CIPHER_PARAM_AEAD_TLS1_SET_IV_INV**) <octet string>
Sets the invocation field used for decryption. Can only be called after "tlsivfixed" is set. This is only used for GCM mode.

"tls1multi_enc" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC**) <octet string>
Triggers a multiblock TLS1 encrypt operation for a TLS1 aware cipher that supports sending 4 or 8 records in one go. The cipher performs both the MAC and encrypt stages and constructs the record headers itself. "tls1multi_enc" supplies the output buffer for the encrypt operation, "tls1multi_encin" & "tls1multi_interleave" must also be set in order to supply values to the encrypt operation.

"tls1multi_encin" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC_IN**) <octet string>
Supplies the data to encrypt for a TLS1 multiblock cipher operation.

"tls1multi_maxsndfrag"
(**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_MAX_SEND_FRAGMENT**) <unsigned integer>
Sets the maximum send fragment size for a TLS1 multiblock cipher operation. It must be set before using "tls1multi_maxbufsz". The length of the "tls1multi_maxsndfrag" parameter should not exceed that of a **size_t**.

"tls1multi_aad" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_AAD**) <octet string>
Sets the authenticated additional data used by a TLS1 multiblock cipher operation. The supplied data consists of 13 bytes of record data containing: Bytes 0-7: The sequence number of the first record Byte 8: The record type Byte 9-10: The protocol version Byte 11-12: Input length (Always 0)

"tls1multi_interleave" must also be set for this operation.

CONTROLS

The Mappings from **EVP_CIPHER_CTX_ctrl()** identifiers to PARAMETERS are listed in the following section. See the "PARAMETERS" section for more details.

EVP_CIPHER_CTX_ctrl() can be used to send the following standard controls:

EVP_CTRL_AEAD_SET_IVLEN and EVP_CTRL_GET_IVLEN
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** and **EVP_CIPHER_CTX_get_params()** get called with an **OSSL_PARAM**(3) item with the key "ivlen" (**OSSL_CIPHER_PARAM_IVLEN**).

EVP_CTRL_AEAD_SET_IV_FIXED
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "tlsivfixed" (**OSSL_CIPHER_PARAM_AEAD_TLS1_IV_FIXED**).

EVP_CTRL_AEAD_SET_MAC_KEY
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "mackey" (**OSSL_CIPHER_PARAM_AEAD_MAC_KEY**).

EVP_CTRL_AEAD_SET_TAG and EVP_CTRL_AEAD_GET_TAG
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** and **EVP_CIPHER_CTX_get_params()** get called with an **OSSL_PARAM**(3) item with the key "tag" (**OSSL_CIPHER_PARAM_AEAD_TAG**).

EVP_CTRL_CCM_SET_L
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "ivlen" (**OSSL_CIPHER_PARAM_IVLEN**) with a value of (15 - L)

EVP_CTRL_COPY
: There is no OSSL_PARAM mapping for this. Use **EVP_CIPHER_CTX_copy()** instead.

EVP_CTRL_GCM_SET_IV_INV
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "tlsivinv" (**OSSL_CIPHER_PARAM_AEAD_TLS1_SET_IV_INV**).

EVP_CTRL_RAND_KEY
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "randkey" (**OSSL_CIPHER_PARAM_RANDOM_KEY**).

EVP_CTRL_SET_KEY_LENGTH
: When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an

OSSL_PARAM(3) item with the key "keylen" (**OSSL_CIPHER_PARAM_KEYLEN**).

EVP_CTRL_SET_RC2_KEY_BITS and EVP_CTRL_GET_RC2_KEY_BITS
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** and **EVP_CIPHER_CTX_get_params()** get called with an **OSSL_PARAM**(3) item with the key "keybits" (**OSSL_CIPHER_PARAM_RC2_KEYBITS**).

EVP_CTRL_SET_RC5_ROUNDS and EVP_CTRL_GET_RC5_ROUNDS
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** and **EVP_CIPHER_CTX_get_params()** get called with an **OSSL_PARAM**(3) item with the key "rounds" (**OSSL_CIPHER_PARAM_ROUNDS**).

EVP_CTRL_SET_SPEED
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key "speed" (**OSSL_CIPHER_PARAM_SPEED**).

EVP_CTRL_GCM_IV_GEN
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_get_params()** gets called with an **OSSL_PARAM**(3) item with the key "tlsivgen" (**OSSL_CIPHER_PARAM_AEAD_TLS1_GET_IV_GEN**).

EVP_CTRL_AEAD_TLS1_AAD
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** get called with an **OSSL_PARAM**(3) item with the key "tlsaad" (**OSSL_CIPHER_PARAM_AEAD_TLS1_AAD**) followed by **EVP_CIPHER_CTX_get_params()** with a key of "tlsaadpad" (**OSSL_CIPHER_PARAM_AEAD_TLS1_AAD_PAD**).

EVP_CTRL_TLS1_1_MULTIBLOCK_MAX_BUFSIZE
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with an **OSSL_PARAM**(3) item with the key OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_MAX_SEND_FRAGMENT followed by **EVP_CIPHER_CTX_get_params()** with a key of "tls1multi_maxbufsz" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_MAX_BUFSIZE**).

EVP_CTRL_TLS1_1_MULTIBLOCK_AAD
When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with **OSSL_PARAM**(3) items with the keys "tls1multi_aad" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_AAD**) and "tls1multi_interleave" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_INTERLEAVE**) followed by **EVP_CIPHER_CTX_get_params()** with keys of "tls1multi_aadpacklen"

(**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_AAD_PACKLEN**) and "tls1multi_interleave" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_INTERLEAVE**).

EVP_CTRL_TLS1_1_MULTIBLOCK_ENCRYPT

When used with a fetched **EVP_CIPHER**, **EVP_CIPHER_CTX_set_params()** gets called with **OSSL_PARAM**(3) items with the keys "tls1multi_enc" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC**), "tls1multi_encin" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC_IN**) and "tls1multi_interleave" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_INTERLEAVE**), followed by **EVP_CIPHER_CTX_get_params()** with a key of "tls1multi_enclen" (**OSSL_CIPHER_PARAM_TLS1_MULTIBLOCK_ENC_LEN**).

FLAGS

EVP_CIPHER_CTX_set_flags(), **EVP_CIPHER_CTX_clear_flags()** and **EVP_CIPHER_CTX_test_flags()**. can be used to manipulate and test these **EVP_CIPHER_CTX** flags:

EVP_CIPH_NO_PADDING

Used by **EVP_CIPHER_CTX_set_padding()**.

See also "Gettable and Settable EVP_CIPHER_CTX parameters" "padding"

EVP_CIPH_FLAG_LENGTH_BITS

See "Settable EVP_CIPHER_CTX parameters" "use-bits".

EVP_CIPHER_CTX_FLAG_WRAP_ALLOW

Used for Legacy purposes only. This flag needed to be set to indicate the cipher handled wrapping.

EVP_CIPHER_flags() uses the following flags that have mappings to "Gettable EVP_CIPHER parameters":

EVP_CIPH_FLAG_AEAD_CIPHER

See "Gettable EVP_CIPHER parameters" "aead".

EVP_CIPH_CUSTOM_IV

See "Gettable EVP_CIPHER parameters" "custom-iv".

EVP_CIPH_FLAG_CTS

See "Gettable EVP_CIPHER parameters" "cts".

EVP_CIPH_FLAG_TLS1_1_MULTIBLOCK;
See "Gettable EVP_CIPHER parameters" "tls-multi".

EVP_CIPH_RAND_KEY
See "Gettable EVP_CIPHER parameters" "has-randkey".

EVP_CIPHER_flags() uses the following flags for legacy purposes only:

EVP_CIPH_VARIABLE_LENGTH
EVP_CIPH_FLAG_CUSTOM_CIPHER
EVP_CIPH_ALWAYS_CALL_INIT
EVP_CIPH_CTRL_INIT
EVP_CIPH_CUSTOM_KEY_LENGTH
EVP_CIPH_CUSTOM_COPY
EVP_CIPH_FLAG_DEFAULT_ASN1
See **EVP_CIPHER_meth_set_flags**(3) for further information related to the above flags.

RETURN VALUES

EVP_CIPHER_fetch() returns a pointer to a **EVP_CIPHER** for success and **NULL** for failure.

EVP_CIPHER_up_ref() returns 1 for success or 0 otherwise.

EVP_CIPHER_CTX_new() returns a pointer to a newly created **EVP_CIPHER_CTX** for success and **NULL** for failure.

EVP_EncryptInit_ex2(), **EVP_EncryptUpdate()** and **EVP_EncryptFinal_ex()** return 1 for success and 0 for failure.

EVP_DecryptInit_ex2() and **EVP_DecryptUpdate()** return 1 for success and 0 for failure.
EVP_DecryptFinal_ex() returns 0 if the decrypt failed or 1 for success.

EVP_CipherInit_ex2() and **EVP_CipherUpdate()** return 1 for success and 0 for failure.
EVP_CipherFinal_ex() returns 0 for a decryption failure or 1 for success.

EVP_Cipher() returns 1 on success or 0 on failure, if the flag **EVP_CIPH_FLAG_CUSTOM_CIPHER** is not set for the cipher. **EVP_Cipher()** returns the number of bytes written to *out* for encryption / decryption, or the number of bytes authenticated in a call specifying AAD for an AEAD cipher, if the flag **EVP_CIPH_FLAG_CUSTOM_CIPHER** is set for the cipher.

EVP_CIPHER_CTX_reset() returns 1 for success and 0 for failure.

EVP_get_cipherbyname(), **EVP_get_cipherbynid()** and **EVP_get_cipherbyobj()** return an **EVP_CIPHER** structure or NULL on error.

EVP_CIPHER_get_nid() and **EVP_CIPHER_CTX_get_nid()** return a NID.

EVP_CIPHER_get_block_size() and **EVP_CIPHER_CTX_get_block_size()** return the block size.

EVP_CIPHER_get_key_length() and **EVP_CIPHER_CTX_get_key_length()** return the key length.

EVP_CIPHER_CTX_set_padding() always returns 1.

EVP_CIPHER_get_iv_length() and **EVP_CIPHER_CTX_get_iv_length()** return the IV length or zero if the cipher does not use an IV.

EVP_CIPHER_CTX_get_tag_length() return the tag length or zero if the cipher does not use a tag.

EVP_CIPHER_get_type() and **EVP_CIPHER_CTX_get_type()** return the NID of the cipher's OBJECT IDENTIFIER or NID_undef if it has no defined OBJECT IDENTIFIER.

EVP_CIPHER_CTX_cipher() returns an **EVP_CIPHER** structure.

EVP_CIPHER_CTX_get_num() returns a nonnegative num value or **EVP_CTRL_RET_UNSUPPORTED** if the implementation does not support the call or on any other error.

EVP_CIPHER_CTX_set_num() returns 1 on success and 0 if the implementation does not support the call or on any other error.

EVP_CIPHER_CTX_is_encrypting() returns 1 if the *ctx* is set up for encryption 0 otherwise.

EVP_CIPHER_param_to_asn1() and **EVP_CIPHER_asn1_to_param()** return greater than zero for success and zero or a negative number on failure.

EVP_CIPHER_CTX_rand_key() returns 1 for success and zero or a negative number for failure.

EVP_CIPHER_names_do_all() returns 1 if the callback was called for all names. A return value of 0 means that the callback was not called for any names.

CIPHER LISTING

All algorithms have a fixed key length unless otherwise stated.

Refer to "SEE ALSO" for the full list of ciphers available through the EVP interface.

EVP_enc_null()
Null cipher: does nothing.

AEAD INTERFACE

The EVP interface for Authenticated Encryption with Associated Data (AEAD) modes are subtly altered and several additional *ctrl* operations are supported depending on the mode specified.

To specify additional authenticated data (AAD), a call to **EVP_CipherUpdate()**, **EVP_EncryptUpdate()** or **EVP_DecryptUpdate()** should be made with the output parameter *out* set to **NULL**. In this case, on success, the parameter *outl* is set to the number of bytes authenticated.

When decrypting, the return value of **EVP_DecryptFinal()** or **EVP_CipherFinal()** indicates whether the operation was successful. If it does not indicate success, the authentication operation has failed and any output data **MUST NOT** be used as it is corrupted.

GCM and OCB Modes

The following *ctrl*s are supported in GCM and OCB modes.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_IVLEN, ivlen, NULL)
Sets the IV length. This call can only be made before specifying an IV. If not called a default IV length is used.

For GCM AES and OCB AES the default is 12 (i.e. 96 bits). For OCB mode the maximum is 15.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_GET_TAG, taglen, tag)
Writes "taglen" bytes of the tag value to the buffer indicated by "tag". This call can only be made when encrypting data and **after** all data has been processed (e.g. after an **EVP_EncryptFinal()** call).

For OCB, "taglen" must either be 16 or the value previously set via **EVP_CTRL_AEAD_SET_TAG**.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_TAG, taglen, tag)
When decrypting, this call sets the expected tag to "taglen" bytes from "tag". "taglen" must be between 1 and 16 inclusive. The tag must be set prior to any call to **EVP_DecryptFinal()** or **EVP_DecryptFinal_ex()**.

For GCM, this call is only valid when decrypting data.

For OCB, this call is valid when decrypting data to set the expected tag, and when encrypting to set the desired tag length.

In OCB mode, calling this when encrypting with "tag" set to "NULL" sets the tag length. The tag length can only be set before specifying an IV. If this is not called prior to setting the IV during encryption, then a default tag length is used.

For OCB AES, the default tag length is 16 (i.e. 128 bits). It is also the maximum tag length for OCB.

CCM Mode

The EVP interface for CCM mode is similar to that of the GCM mode but with a few additional requirements and different *ctrl* values.

For CCM mode, the total plaintext or ciphertext length **MUST** be passed to **EVP_CipherUpdate()**, **EVP_EncryptUpdate()** or **EVP_DecryptUpdate()** with the output and input parameters (*in* and *out*) set to **NULL** and the length passed in the *inl* parameter.

The following *ctrl*s are supported in CCM mode.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_TAG, taglen, tag)

This call is made to set the expected **CCM** tag value when decrypting or the length of the tag (with the "tag" parameter set to NULL) when encrypting. The tag length is often referred to as **M**. If not set a default value is used (12 for AES). When decrypting, the tag needs to be set before passing in data to be decrypted, but as in GCM and OCB mode, it can be set after passing additional authenticated data (see "AEAD INTERFACE").

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_CCM_SET_L, ivlen, NULL)

Sets the CCM **L** value. If not set a default is used (8 for AES).

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_IVLEN, ivlen, NULL)

Sets the CCM nonce (IV) length. This call can only be made before specifying a nonce value. The nonce length is given by **15 - L** so it is 7 by default for AES.

SIV Mode

For SIV mode ciphers the behaviour of the EVP interface is subtly altered and several additional ctrl operations are supported.

To specify any additional authenticated data (AAD) and/or a Nonce, a call to **EVP_CipherUpdate()**, **EVP_EncryptUpdate()** or **EVP_DecryptUpdate()** should be made with the output parameter *out* set to

NULL.

RFC5297 states that the Nonce is the last piece of AAD before the actual encrypt/decrypt takes place. The API does not differentiate the Nonce from other AAD.

When decrypting the return value of **EVP_DecryptFinal()** or **EVP_CipherFinal()** indicates if the operation was successful. If it does not indicate success the authentication operation has failed and any output data **MUST NOT** be used as it is corrupted.

The API does not store the the SIV (Synthetic Initialization Vector) in the cipher text. Instead, it is stored as the tag within the EVP_CIPHER_CTX. The SIV must be retrieved from the context after encryption, and set into the context before decryption.

This differs from RFC5297 in that the cipher output from encryption, and the cipher input to decryption, does not contain the SIV. This also means that the plain text and cipher text lengths are identical.

The following ctrls are supported in SIV mode, and are used to get and set the Synthetic Initialization Vector:

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_GET_TAG, taglen, tag);
: Writes *taglen* bytes of the tag value (the Synthetic Initialization Vector) to the buffer indicated by *tag*. This call can only be made when encrypting data and **after** all data has been processed (e.g. after an **EVP_EncryptFinal()** call). For SIV mode the taglen must be 16.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_TAG, taglen, tag);
: Sets the expected tag (the Synthetic Initialization Vector) to *taglen* bytes from *tag*. This call is only legal when decrypting data and must be made **before** any data is processed (e.g. before any **EVP_DecryptUpdate()** calls). For SIV mode the taglen must be 16.

SIV mode makes two passes over the input data, thus, only one call to **EVP_CipherUpdate()**, **EVP_EncryptUpdate()** or **EVP_DecryptUpdate()** should be made with *out* set to a non-**NULL** value. A call to **EVP_DecryptFinal()** or **EVP_CipherFinal()** is not required, but will indicate if the update operation succeeded.

ChaCha20-Poly1305

The following *ctrl*s are supported for the ChaCha20-Poly1305 AEAD algorithm.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_IVLEN, ivlen, NULL)
: Sets the nonce length. This call is now redundant since the only valid value is the default length of

12 (i.e. 96 bits). Prior to OpenSSL 3.0 a nonce of less than 12 bytes could be used to automatically pad the iv with leading 0 bytes to make it 12 bytes in length.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_GET_TAG, taglen, tag)

Writes "taglen" bytes of the tag value to the buffer indicated by "tag". This call can only be made when encrypting data and **after** all data has been processed (e.g. after an **EVP_EncryptFinal()** call).

"taglen" specified here must be 16 (**POLY1305_BLOCK_SIZE**, i.e. 128-bits) or less.

EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_AEAD_SET_TAG, taglen, tag)

Sets the expected tag to "taglen" bytes from "tag". The tag length can only be set before specifying an IV. "taglen" must be between 1 and 16 (**POLY1305_BLOCK_SIZE**) inclusive. This call is only valid when decrypting data.

NOTES

Where possible the **EVP** interface to symmetric ciphers should be used in preference to the low-level interfaces. This is because the code then becomes transparent to the cipher used and much more flexible. Additionally, the **EVP** interface will ensure the use of platform specific cryptographic acceleration such as AES-NI (the low-level interfaces do not provide the guarantee).

PKCS padding works by adding **n** padding bytes of value **n** to make the total length of the encrypted data a multiple of the block size. Padding is always added so if the data is already a multiple of the block size **n** will equal the block size. For example if the block size is 8 and 11 bytes are to be encrypted then 5 padding bytes of value 5 will be added.

When decrypting the final block is checked to see if it has the correct form.

Although the decryption operation can produce an error if padding is enabled, it is not a strong test that the input data or key is correct. A random block has better than 1 in 256 chance of being of the correct format and problems with the input data earlier on will not produce a final decrypt error.

If padding is disabled then the decryption operation will always succeed if the total amount of data decrypted is a multiple of the block size.

The functions **EVP_EncryptInit()**, **EVP_EncryptInit_ex()**, **EVP_EncryptFinal()**, **EVP_DecryptInit()**, **EVP_DecryptInit_ex()**, **EVP_CipherInit()**, **EVP_CipherInit_ex()** and **EVP_CipherFinal()** are obsolete but are retained for compatibility with existing code. New code should use **EVP_EncryptInit_ex2()**, **EVP_EncryptFinal_ex()**, **EVP_DecryptInit_ex2()**, **EVP_DecryptFinal_ex()**, **EVP_CipherInit_ex2()** and **EVP_CipherFinal_ex()** because they can reuse an existing context without allocating and freeing it up

on each call.

There are some differences between functions **EVP_CipherInit()** and **EVP_CipherInit_ex()**, significant in some circumstances. **EVP_CipherInit()** fills the passed context object with zeros. As a consequence, **EVP_CipherInit()** does not allow step-by-step initialization of the ctx when the *key* and *iv* are passed in separate calls. It also means that the flags set for the CTX are removed, and it is especially important for the **EVP_CIPHER_CTX_FLAG_WRAP_ALLOW** flag treated specially in **EVP_CipherInit_ex()**.

Ignoring failure returns of the **EVP_CIPHER_CTX** initialization functions can lead to subsequent undefined behavior when calling the functions that update or finalize the context. The only valid calls on the **EVP_CIPHER_CTX** when initialization fails are calls that attempt another initialization of the context or release the context.

EVP_get_cipherbynid(), and **EVP_get_cipherbyobj()** are implemented as macros.

BUGS

EVP_MAX_KEY_LENGTH and **EVP_MAX_IV_LENGTH** only refer to the internal ciphers with default key lengths. If custom ciphers exceed these values the results are unpredictable. This is because it has become standard practice to define a generic key as a fixed unsigned char array containing **EVP_MAX_KEY_LENGTH** bytes.

The ASN1 code is incomplete (and sometimes inaccurate) it has only been tested for certain common S/MIME ciphers (RC2, DES, triple DES) in CBC mode.

EXAMPLES

Encrypt a string using IDEA:

```
int do_crypt(char *outfile)
{
    unsigned char outbuf[1024];
    int outlen, tmplen;
    /*
     * Bogus key and IV: we'd normally set these from
     * another source.
     */
    unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    unsigned char iv[] = {1,2,3,4,5,6,7,8};
    char intext[] = "Some Crypto Text";
    EVP_CIPHER_CTX *ctx;
    FILE *out;
```

```
    ctx = EVP_CIPHER_CTX_new();
    if (!EVP_EncryptInit_ex2(ctx, EVP_idea_cbc(), key, iv, NULL)) {
        /* Error */
        EVP_CIPHER_CTX_free(ctx);
        return 0;
    }

    if (!EVP_EncryptUpdate(ctx, outbuf, &outlen, intext, strlen(intext))) {
        /* Error */
        EVP_CIPHER_CTX_free(ctx);
        return 0;
    }
    /*
     * Buffer passed to EVP_EncryptFinal() must be after data just
     * encrypted to avoid overwriting it.
     */
    if (!EVP_EncryptFinal_ex(ctx, outbuf + outlen, &tmplen)) {
        /* Error */
        EVP_CIPHER_CTX_free(ctx);
        return 0;
    }
    outlen += tmplen;
    EVP_CIPHER_CTX_free(ctx);
    /*
     * Need binary mode for fopen because encrypted data is
     * binary data. Also cannot use strlen() on it because
     * it won't be NUL terminated and may contain embedded
     * NULs.
     */
    out = fopen(outfile, "wb");
    if (out == NULL) {
        /* Error */
        return 0;
    }
    fwrite(outbuf, 1, outlen, out);
    fclose(out);
    return 1;
}
```

The ciphertext from the above example can be decrypted using the **openssl** utility with the command

line (shown on two lines for clarity):

```
openssl idea -d \
   -K 000102030405060708090A0B0C0D0E0F -iv 0102030405060708 <filename
```

General encryption and decryption function example using FILE I/O and AES128 with a 128-bit key:

```
int do_crypt(FILE *in, FILE *out, int do_encrypt)
{
   /* Allow enough space in output buffer for additional block */
   unsigned char inbuf[1024], outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
   int inlen, outlen;
   EVP_CIPHER_CTX *ctx;
   /*
    * Bogus key and IV: we'd normally set these from
    * another source.
    */
   unsigned char key[] = "0123456789abcdeF";
   unsigned char iv[] = "1234567887654321";

   /* Don't set key or IV right away; we want to check lengths */
   ctx = EVP_CIPHER_CTX_new();
   if (!EVP_CipherInit_ex2(ctx, EVP_aes_128_cbc(), NULL, NULL,
                           do_encrypt, NULL)) {
      /* Error */
      EVP_CIPHER_CTX_free(ctx);
      return 0;
   }
   OPENSSL_assert(EVP_CIPHER_CTX_get_key_length(ctx) == 16);
   OPENSSL_assert(EVP_CIPHER_CTX_get_iv_length(ctx) == 16);

   /* Now we can set key and IV */
   if (!EVP_CipherInit_ex2(ctx, NULL, key, iv, do_encrypt, NULL)) {
      /* Error */
      EVP_CIPHER_CTX_free(ctx);
      return 0;
   }

   for (;;) {
      inlen = fread(inbuf, 1, 1024, in);
```

```
        if (inlen <= 0)
            break;
        if (!EVP_CipherUpdate(ctx, outbuf, &outlen, inbuf, inlen)) {
            /* Error */
            EVP_CIPHER_CTX_free(ctx);
            return 0;
        }
        fwrite(outbuf, 1, outlen, out);
    }
    if (!EVP_CipherFinal_ex(ctx, outbuf, &outlen)) {
        /* Error */
        EVP_CIPHER_CTX_free(ctx);
        return 0;
    }
    fwrite(outbuf, 1, outlen, out);

    EVP_CIPHER_CTX_free(ctx);
    return 1;
}
```

Encryption using AES-CBC with a 256-bit key with "CS1" ciphertext stealing.

```
int encrypt(const unsigned char *key, const unsigned char *iv,
            const unsigned char *msg, size_t msg_len, unsigned char *out)
{
    /*
     * This assumes that key size is 32 bytes and the iv is 16 bytes.
     * For ciphertext stealing mode the length of the ciphertext "out" will be
     * the same size as the plaintext size "msg_len".
     * The "msg_len" can be any size >= 16.
     */
    int ret = 0, encrypt = 1, outlen, len;
    EVP_CIPHER_CTX *ctx = NULL;
    EVP_CIPHER *cipher = NULL;
    OSSL_PARAM params[2];

    ctx = EVP_CIPHER_CTX_new();
    cipher = EVP_CIPHER_fetch(NULL, "AES-256-CBC-CTS", NULL);
    if (ctx == NULL || cipher == NULL)
        goto err;
```

```
    /*
     * The default is "CS1" so this is not really needed,
     * but would be needed to set either "CS2" or "CS3".
     */
    params[0] = OSSL_PARAM_construct_utf8_string(OSSL_CIPHER_PARAM_CTS_MODE,
                                                 "CS1", 0);
    params[1] = OSSL_PARAM_construct_end();

    if (!EVP_CipherInit_ex2(ctx, cipher, key, iv, encrypt, params))
        goto err;

    /* NOTE: CTS mode does not support multiple calls to EVP_CipherUpdate() */
    if (!EVP_CipherUpdate(ctx, out, &outlen, msg, msg_len))
        goto err;
     if (!EVP_CipherFinal_ex(ctx, out + outlen, &len))
        goto err;
    ret = 1;
 err:
    EVP_CIPHER_free(cipher);
    EVP_CIPHER_CTX_free(ctx);
    return ret;
 }
```

SEE ALSO

evp(7), **property**(7), "ALGORITHM FETCHING" in **crypto**(7), **provider-cipher**(7), **life_cycle-cipher**(7)

Supported ciphers are listed in:

EVP_aes_128_gcm(3), **EVP_aria_128_gcm**(3), **EVP_bf_cbc**(3), **EVP_camellia_128_ecb**(3), **EVP_cast5_cbc**(3), **EVP_chacha20**(3), **EVP_des_cbc**(3), **EVP_desx_cbc**(3), **EVP_idea_cbc**(3), **EVP_rc2_cbc**(3), **EVP_rc4**(3), **EVP_rc5_32_12_16_cbc**(3), **EVP_seed_cbc**(3), **EVP_sm4_cbc**(3),

HISTORY

Support for OCB mode was added in OpenSSL 1.1.0.

EVP_CIPHER_CTX was made opaque in OpenSSL 1.1.0. As a result, **EVP_CIPHER_CTX_reset()** appeared and **EVP_CIPHER_CTX_cleanup()** disappeared. **EVP_CIPHER_CTX_init()** remains as an alias for **EVP_CIPHER_CTX_reset()**.

The **EVP_CIPHER_CTX_cipher()** function was deprecated in OpenSSL 3.0; use

EVP_CIPHER_CTX_get0_cipher() instead.

The **EVP_EncryptInit_ex2()**, **EVP_DecryptInit_ex2()**, **EVP_CipherInit_ex2()**, **EVP_CIPHER_fetch()**, **EVP_CIPHER_free()**, **EVP_CIPHER_up_ref()**, **EVP_CIPHER_CTX_get0_cipher()**, **EVP_CIPHER_CTX_get1_cipher()**, **EVP_CIPHER_get_params()**, **EVP_CIPHER_CTX_set_params()**, **EVP_CIPHER_CTX_get_params()**, **EVP_CIPHER_gettable_params()**, **EVP_CIPHER_settable_ctx_params()**, **EVP_CIPHER_gettable_ctx_params()**, **EVP_CIPHER_CTX_settable_params()** and **EVP_CIPHER_CTX_gettable_params()** functions were added in 3.0.

The **EVP_CIPHER_nid()**, **EVP_CIPHER_name()**, **EVP_CIPHER_block_size()**, **EVP_CIPHER_key_length()**, **EVP_CIPHER_iv_length()**, **EVP_CIPHER_flags()**, **EVP_CIPHER_mode()**, **EVP_CIPHER_type()**, **EVP_CIPHER_CTX_nid()**, **EVP_CIPHER_CTX_block_size()**, **EVP_CIPHER_CTX_key_length()**, **EVP_CIPHER_CTX_iv_length()**, **EVP_CIPHER_CTX_tag_length()**, **EVP_CIPHER_CTX_num()**, **EVP_CIPHER_CTX_type()**, and **EVP_CIPHER_CTX_mode()** functions were renamed to include "get" or "get0" in their names in OpenSSL 3.0, respectively. The old names are kept as non-deprecated alias macros.

The **EVP_CIPHER_CTX_encrypting()** function was renamed to **EVP_CIPHER_CTX_is_encrypting()** in OpenSSL 3.0. The old name is kept as non-deprecated alias macro.

The **EVP_CIPHER_CTX_flags()** macro was deprecated in OpenSSL 1.1.0.

COPYRIGHT