Danger

This is a "Hazardous Materials" module. You should **ONLY** use it if you're 100% absolutely sure that you know what you're doing because this module is full of land mines, dragons, and dinosaurs with laser guns. You may instead be interested in Fernet (symmetric encryption).

Symmetric encryption

Symmetric encryption is a way to encrypt or hide the contents of material where the sender and receiver both use the same secret key. Note that symmetric encryption is **not** sufficient for most applications because it only provides secrecy but not authenticity. That means an attacker can't see the message but an attacker can create bogus messages and force the application to decrypt them. In many contexts, a lack of authentication on encrypted messages can result in a loss of secrecy as well.

For this reason it is **strongly** recommended to combine encryption with a message authentication code, such as HMAC, in an "encrypt-then-MAC" formulation as described by Colin Percival. `cryptography` includes a recipe named Fernet (symmetric encryption) that does this for you. **To minimize the risk of security issues you should evaluate Fernet to see if it fits your needs before implementing anything using this module.**

class **`cryptography.hazmat.primitives.ciphers.Cipher`**(*algorithm*, *mode*, *backend=None*)

Cipher objects combine an algorithm such as **`AES`** with a mode like **`CBC`** or **`CTR`**. A simple example of encrypting and then decrypting content with AES is:

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> key = os.urandom(32)
>>> iv = os.urandom(16)
>>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message") + encryptor.finalize()
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct) + decryptor.finalize()
b'a secret message'
```

Parameters:
- **algorithm** – A `CipherAlgorithm` instance such as those described below.
- **mode** – A `Mode` instance such as those described below.
- **backend** – An optional `CipherBackend` instance.

Raises: cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `backend` does not implement `CipherBackend`

`encryptor()`

Returns: An encrypting `CipherContext` instance.

If the backend doesn't support the requested combination of `cipher` and `mode` an `UnsupportedAlgorithm` exception will be raised.

`decryptor()`

Returns: A decrypting `CipherContext` instance.

If the backend doesn't support the requested combination of `cipher` and `mode` an `UnsupportedAlgorithm` exception will be raised.

Algorithms

class **`cryptography.hazmat.primitives.ciphers.algorithms.AES`**(*key*)

AES (Advanced Encryption Standard) is a block cipher standardized by NIST. AES is both fast, and cryptographically strong. It is a good default choice for encryption.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. Either `128`, `192`, or `256` bits long.

class **`cryptography.hazmat.primitives.ciphers.algorithms.Camellia`**(*key*)

Camellia is a block cipher approved for use by CRYPTREC and ISO/IEC. It is considered to have comparable security and performance to AES but is not as widely studied or deployed.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. Either `128`, `192`, or `256` bits long.

class **`cryptography.hazmat.primitives.ciphers.algorithms.ChaCha20`**(*key*)

New in version 2.1.

Note

In most cases users should use `ChaCha20Poly1305` instead of this class. *ChaCha20* alone does not provide integrity so it must be combined with a MAC to be secure. `ChaCha20Poly1305` does this for you.

ChaCha20 is a stream cipher used in several IETF protocols. It is standardized in RFC 7539.

Parameters:

- **key** (bytes-like) – The secret key. This must be kept secret. `256` bits (32 bytes) in length.
- **nonce** – Should be unique, a nonce. It is critical to never reuse a `nonce` with a given key. Any reuse of a nonce with the same key compromises the security of every message encrypted with that key. The nonce does not need to be kept secret and may be included with the ciphertext. This must be `128` bits in length.

```
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> nonce = os.urandom(16)
>>> algorithm = algorithms.ChaCha20(key, nonce)
>>> cipher = Cipher(algorithm, mode=None)
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message")
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct)
b'a secret message'
```

class **cryptography.hazmat.primitives.ciphers.algorithms.TripleDES**(*key*)

Triple DES (Data Encryption Standard), sometimes referred to as 3DES, is a block cipher standardized by NIST. Triple DES has known crypto-analytic flaws, however none of them currently enable a practical attack. Nonetheless, Triple DES is not recommended for new applications because it is incredibly slow; old applications should consider moving away from it.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. Either `64`, `128`, or `192` bits long. DES only uses `56`, `112`, or `168` bits of the key as there is a parity byte in each component of the key. Some writing refers to there being up to three separate keys that are each `56` bits long, they can simply be concatenated to produce the full key.

class **cryptography.hazmat.primitives.ciphers.algorithms.CAST5**(*key*)

New in version 0.2.

CAST5 (also known as CAST-128) is a block cipher approved for use in the Canadian government by the Communications Security Establishment. It is a variable key length cipher and supports keys from 40-128 bits in length.

Parameters: **key** (bytes-like) – The secret key, This must be kept secret. 40 to 128 bits in length in increments of 8 bits.

---

class **cryptography.hazmat.primitives.ciphers.algorithms.SEED**(*key*)

New in version 0.4.

SEED is a block cipher developed by the Korea Information Security Agency (KISA). It is defined in **RFC 4269** and is used broadly throughout South Korean industry, but rarely found elsewhere.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. `128` bits in length.

---

class **cryptography.hazmat.primitives.ciphers.algorithms.SM4**(*key*)

New in version 35.0.0.

SM4 is a block cipher developed by the Chinese Government and standardized in the GB/T 32907-2016. It is used in the Chinese WAPI (Wired Authentication and Privacy Infrastructure) standard. (An English description is available at draft-ribose-cfrg-sm4-10.) This block cipher should be used for compatibility purposes where required and is not otherwise recommended for use.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. `128` bits in length.

Weak ciphers

Warning

These ciphers are considered weak for a variety of reasons. New applications should avoid their use and existing applications should strongly consider migrating away.

---

class **cryptography.hazmat.primitives.ciphers.algorithms.Blowfish**(*key*)

Blowfish is a block cipher developed by Bruce Schneier. It is known to be susceptible to attacks when using weak keys. The author has recommended that users of Blowfish move to newer algorithms such as `AES`.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. 32 to 448 bits in length in increments of 8 bits.

class **cryptography.hazmat.primitives.ciphers.algorithms.ARC4**(*key*)

ARC4 (Alleged RC4) is a stream cipher with serious weaknesses in its initial stream output. Its use is strongly discouraged. ARC4 does not use mode constructions.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. Either `40`, `56`, `64`, `80`, `128`, `192`, or `256` bits in length.

```
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> algorithm = algorithms.ARC4(key)
>>> cipher = Cipher(algorithm, mode=None)
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message")
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct)
b'a secret message'
```

class **cryptography.hazmat.primitives.ciphers.algorithms.IDEA**(*key*)

IDEA (International Data Encryption Algorithm) is a block cipher created in 1991. It is an optional component of the OpenPGP standard. This cipher is susceptible to attacks when using weak keys. It is recommended that you do not use this cipher for new applications.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. `128` bits in length.

Modes

class **cryptography.hazmat.primitives.ciphers.modes.CBC**(*initialization_vector*)

CBC (Cipher Block Chaining) is a mode of operation for block ciphers. It is considered cryptographically strong.

Padding is required when using this mode.

:Parameters: **initialization_vector** (bytes-like) – Must be random bytes. They do not need to be kept secret and they can be included in a transmitted message. Must be the same number of bytes as the `block_size` of the cipher. Each time something is encrypted a new `initialization_vector` should be generated. Do not reuse an `initialization_vector` with a given `key`, and particularly do not use a constant `initialization_vector`.

A good construction looks like:

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.modes import CBC
>>> iv = os.urandom(16)
>>> mode = CBC(iv)
```

While the following is bad and will leak information:

```
>>> from cryptography.hazmat.primitives.ciphers.modes import CBC
>>> iv = b"a" * 16
>>> mode = CBC(iv)
```

class **cryptography.hazmat.primitives.ciphers.modes.CTR**(*nonce*)

Warning

Counter mode is not recommended for use with block ciphers that have a block size of less than 128-bits.

CTR (Counter) is a mode of operation for block ciphers. It is considered cryptographically strong. It transforms a block cipher into a stream cipher.

This mode does not require padding.

:Parameters: **nonce** (bytes-like) – Should be unique, a nonce. It is critical to never reuse a `nonce` with a given key. Any reuse of a nonce with the same key compromises the security of every message encrypted with that key. Must be the same number of bytes as the `block_size` of the cipher with a given key. The nonce does not need to be kept secret and may be included with the ciphertext.

class **cryptography.hazmat.primitives.ciphers.modes.OFB**(*initialization_vector*)

OFB (Output Feedback) is a mode of operation for block ciphers. It transforms a block

cipher into a stream cipher.

This mode does not require padding.

Parameters: **initialization_vector** (bytes-like) – Must be random bytes. They do not need to be kept secret and they can be included in a transmitted message. Must be the same number of bytes as the `block_size` of the cipher. Do not reuse an `initialization_vector` with a given `key`.

---

class **`cryptography.hazmat.primitives.ciphers.modes.CFB`**(*initialization_vector*)

CFB (Cipher Feedback) is a mode of operation for block ciphers. It transforms a block cipher into a stream cipher.

This mode does not require padding.

Parameters: **initialization_vector** (bytes-like) – Must be random bytes. They do not need to be kept secret and they can be included in a transmitted message. Must be the same number of bytes as the `block_size` of the cipher. Do not reuse an `initialization_vector` with a given `key`.

---

class **`cryptography.hazmat.primitives.ciphers.modes.CFB8`**(*initialization_vector*)

CFB (Cipher Feedback) is a mode of operation for block ciphers. It transforms a block cipher into a stream cipher. The CFB8 variant uses an 8-bit shift register.

This mode does not require padding.

Parameters: **initialization_vector** (bytes-like) – Must be random bytes. They do not need to be kept secret and they can be included in a transmitted message. Must be the same number of bytes as the `block_size` of the cipher. Do not reuse an `initialization_vector` with a given `key`.

---

class **`cryptography.hazmat.primitives.ciphers.modes.GCM`**(*initialization_vector, tag=None, min_tag_length=16*)

Danger

If you are encrypting data that can fit into memory you should strongly consider using `AESGCM` instead of this.

When using this mode you **must** not use the decrypted data until the appropriate finalization method (`finalize()` or `finalize_with_tag()`) has been called. GCM provides **no** guarantees of ciphertext integrity until decryption is complete.

GCM (Galois Counter Mode) is a mode of operation for block ciphers. An AEAD (authenticated encryption with additional data) mode is a type of block cipher mode that simultaneously encrypts the message as well as authenticating it. Additional unencrypted data may also be authenticated. Additional means of verifying integrity such as HMAC are not necessary.

This mode does not require padding.

Parameters: **initialization_vector** (bytes-like) – Must be unique, a nonce. They do not need to be kept secret and they can be included in a transmitted message. NIST recommends a 96-bit IV length for performance critical situations but it can be up to 2^{64} - 1 bits. Do not reuse an `initialization_vector` with a given `key`.

Note

Cryptography will generate a 128-bit tag when finalizing encryption. You can shorten a tag by truncating it to the desired length but this is **not recommended** as it makes it easier to forge messages, and also potentially leaks the key (NIST SP-800-38D recommends 96-bits or greater). Applications wishing to allow truncation can pass the `min_tag_length` parameter.

Changed in version 0.5: The `min_tag_length` parameter was added in `0.5`, previously truncation down to `4` bytes was always allowed.

Parameters:

- **tag** (*bytes*) – The tag bytes to verify during decryption. When encrypting this must be `None`. When decrypting, it may be `None` if the tag is supplied on finalization using `finalize_with_tag()`. Otherwise, the tag is mandatory.
- **min_tag_length** (*int*) – The minimum length `tag` must be. By default this is `16`, meaning tag truncation is not allowed. Allowing tag truncation is strongly discouraged for most applications.

Raises: ValueError – This is raised if `len(tag) < min_tag_length` or the `initialization_vector` is too short.

An example of securely encrypting and decrypting data with `AES` in the `GCM` mode looks like:

```
import os

from cryptography.hazmat.primitives.ciphers import (
    Cipher, algorithms, modes
)

def encrypt(key, plaintext, associated_data):
    # Generate a random 96-bit IV.
    iv = os.urandom(12)

    # Construct an AES-GCM Cipher object with the given key and a
    # randomly generated IV.
    encryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv),
    ).encryptor()

    # associated_data will be authenticated but not encrypted,
    # it must also be passed in on decryption.
    encryptor.authenticate_additional_data(associated_data)

    # Encrypt the plaintext and get the associated ciphertext.
    # GCM does not require padding.
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    return (iv, ciphertext, encryptor.tag)

def decrypt(key, associated_data, iv, ciphertext, tag):
    # Construct a Cipher object, with the key, iv, and additionally the
    # GCM tag used for authenticating the message.
    decryptor = Cipher(
        algorithms.AES(key),
        modes.GCM(iv, tag),
    ).decryptor()

    # We put associated_data back in or the tag will fail to verify
    # when we finalize the decryptor.
    decryptor.authenticate_additional_data(associated_data)

    # Decryption gets us the authenticated plaintext.
    # If the tag does not match an InvalidTag exception will be raised.
    return decryptor.update(ciphertext) + decryptor.finalize()

iv, ciphertext, tag = encrypt(
    key,
    b"a secret message!",
    b"authenticated but not encrypted payload"
)

print(decrypt(
    key,
    b"authenticated but not encrypted payload",
    iv,
    ciphertext,
    tag
))
```

```
b'a secret message!'
```

class **cryptography.hazmat.primitives.ciphers.modes.XTS**(*tweak*)

New in version 2.1.

Warning

XTS mode is meant for disk encryption and should not be used in other contexts. `cryptography` only supports XTS mode with `AES`.

Note

AES XTS keys are double length. This means that to do AES-128 encryption in XTS mode you need a 256-bit key. Similarly, AES-256 requires passing a 512-bit key. AES 192 is not supported in XTS mode.

XTS (XEX-based tweaked-codebook mode with ciphertext stealing) is a mode of operation for the AES block cipher that is used for disk encryption.

This mode does not require padding.

Parameters: **tweak** (bytes-like) – The tweak is a 16 byte value typically derived from something like the disk sector number. A given `(tweak, key)` pair should not be reused, although doing so is less catastrophic than in CTR mode.

Insecure modes

Warning

These modes are insecure. New applications should never make use of them, and existing applications should strongly consider migrating away.

class **cryptography.hazmat.primitives.ciphers.modes.ECB**

ECB (Electronic Code Book) is the simplest mode of operation for block ciphers. Each block of data is encrypted in the same way. This means identical plaintext blocks will always result in identical ciphertext blocks, which can leave significant patterns in the output.

Padding is required when using this mode.

Interfaces

class `cryptography.hazmat.primitives.ciphers.CipherContext`

When calling `encryptor()` or `decryptor()` on a `Cipher` object the result will conform to the `CipherContext` interface. You can then call `update(data)` with data until you have fed everything into the context. Once that is done call `finalize()` to finish the operation and obtain the remainder of the data.

Block ciphers require that the plaintext or ciphertext always be a multiple of their block size. Because of that **padding** is sometimes required to make a message the correct size. `CipherContext` will not automatically apply any padding; you'll need to add your own. For block ciphers the recommended padding is `PKCS7`. If you are using a stream cipher mode (such as `CTR`) you don't have to worry about this.

`update`(*data*)

Parameters: **data** (bytes-like) – The data you wish to pass into the context.

Return bytes: Returns the data that was encrypted or decrypted.

Raises: cryptography.exceptions.AlreadyFinalized – See `finalize()`

When the `Cipher` was constructed in a mode that turns it into a stream cipher (e.g. `CTR`), this will return bytes immediately, however in other modes it will return chunks whose size is determined by the cipher's block size.

`update_into`(*data*, *buf*)

New in version 1.8.

Warning

This method allows you to avoid a memory copy by passing a writable buffer and reading the resulting data. You are responsible for correctly sizing the buffer and properly handling the data. This method should only be used when extremely high performance is a requirement and you will be making many small calls to `update_into`.

Parameters:

- **data** (bytes-like) – The data you wish to pass into the context.
- **buf** – A writable Python buffer that the data will be written into. This buffer should be `len(data) + n - 1` bytes where `n` is the block size (in bytes) of the cipher and mode combination.

Return int: Number of bytes written.

Raises:

- NotImplementedError – This is raised if the version of `cffi` used is too old (this can happen on older PyPy releases).
- ValueError – This is raised if the supplied buffer is too small.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
>>> key = os.urandom(32)
>>> iv = os.urandom(16)
>>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
>>> encryptor = cipher.encryptor()
>>> # the buffer needs to be at least len(data) + n - 1 where n is cipher/mode
block size in bytes
>>> buf = bytearray(31)
>>> len_encrypted = encryptor.update_into(b"a secret message", buf)
>>> # get the ciphertext from the buffer reading only the bytes written to it
(len_encrypted)
>>> ct = bytes(buf[:len_encrypted]) + encryptor.finalize()
>>> decryptor = cipher.decryptor()
>>> len_decrypted = decryptor.update_into(ct, buf)
>>> # get the plaintext from the buffer reading only the bytes written
(len_decrypted)
>>> bytes(buf[:len_decrypted]) + decryptor.finalize()
b'a secret message'
```

`finalize()`

Return bytes: Returns the remainder of the data.

Raises: ValueError – This is raised when the data provided isn't a multiple of the algorithm's block size.

Once `finalize` is called this object can no longer be used and `update()` and `finalize()` will raise an `AlreadyFinalized` exception.

class `cryptography.hazmat.primitives.ciphers.AEADCipherContext`

When calling `encryptor` or `decryptor` on a `Cipher` object with an AEAD mode (e.g. `GCM`) the result will conform to the `AEADCipherContext` and `CipherContext` interfaces. If it is an encryption or decryption context it will additionally be an `AEADEncryptionContext` or `AEADDecryptionContext` instance, respectively. `AEADCipherContext` contains an additional method `authenticate_additional_data()` for adding additional authenticated but unencrypted data (see note below). You should call this before calls to `update`. When you are done call `finalize` to finish the operation.

Note

In AEAD modes all data passed to `update()` will be both encrypted and authenticated. Do not pass encrypted data to the `authenticate_additional_data()` method. It is meant solely for additional data you may want to authenticate but leave unencrypted.

> **`authenticate_additional_data`**(*data*)
>
> **Parameters:** **data** (bytes-like) – Any data you wish to authenticate but not encrypt.
>
> **Raises:** `AlreadyFinalized`

---

class **`cryptography.hazmat.primitives.ciphers.AEADEncryptionContext`**

When creating an encryption context using `encryptor` on a `Cipher` object with an AEAD mode such as `GCM` an object conforming to both the `AEADEncryptionContext` and `AEADCipherContext` interfaces will be returned. This interface provides one additional attribute `tag`. `tag` can only be obtained after `finalize` has been called.

> **`tag`**
>
> **Return bytes:** Returns the tag value as bytes.
>
> **Raises:** `NotYetFinalized` if called before the context is finalized.

---

class **`cryptography.hazmat.primitives.ciphers.AEADDecryptionContext`**

New in version 1.9.

When creating an encryption context using `decryptor` on a `Cipher` object with an AEAD mode such as `GCM` an object conforming to both the `AEADDecryptionContext` and `AEADCipherContext` interfaces will be returned. This interface provides one additional method `finalize_with_tag()` that allows passing the authentication tag for validation after the ciphertext has been decrypted.

> **`finalize_with_tag`**(*tag*)
>
> **Parameters:** **tag** (*bytes*) – The tag bytes to verify after decryption.
>
> **Return bytes:** Returns the remainder of the data.

Raises: ValueError – This is raised when the data provided isn't a multiple of the algorithm's block size, if `min_tag_length` is less than 4, or if `len(tag) < min_tag_length`. `min_tag_length` is an argument to the `GCM` constructor.

If the authentication tag was not already supplied to the constructor of the `GCM` mode object, this method must be used instead of `finalize()`.

---

class **`cryptography.hazmat.primitives.ciphers.CipherAlgorithm`**

A named symmetric encryption algorithm.

`name`

Type: str

The standard name for the mode, for example, "AES", "Camellia", or "Blowfish".

`key_size`

Type: int

The number of bits in the key being used.

---

class **`cryptography.hazmat.primitives.ciphers.BlockCipherAlgorithm`**

A block cipher algorithm.

`block_size`

Type: int

The number of bits in a block.

Interfaces used by the symmetric cipher modes described in Symmetric Encryption Modes.

---

class **`cryptography.hazmat.primitives.ciphers.modes.Mode`**

A named cipher mode.

`name`

Type: str

This should be the standard shorthand name for the mode, for example Cipher-Block Chaining mode is "CBC".

The name may be used by a backend to influence the operation of a cipher in conjunction with the algorithm's name.

`validate_for_algorithm`(*algorithm*)

Parameters: **algorithm** (*cryptography.hazmat.primitives.ciphers.CipherAlgorithm*) –

Checks that the combination of this mode with the provided algorithm meets any necessary invariants. This should raise an exception if they are not met.

For example, the `CBC` mode uses this method to check that the provided initialization vector's length matches the block size of the algorithm.

---

class
cryptography.hazmat.primitives.ciphers.modes.ModeWithInitializationVector

A cipher mode with an initialization vector.

`initialization_vector`

Type: bytes-like

Exact requirements of the initialization are described by the documentation of individual modes.

---

class **cryptography.hazmat.primitives.ciphers.modes.ModeWithNonce**

A cipher mode with a nonce.

`nonce`

Type: bytes-like

Exact requirements of the nonce are described by the documentation of individual modes.

---

class
cryptography.hazmat.primitives.ciphers.modes.ModeWithAuthenticationTag

A cipher mode with an authentication tag.

`tag`

Type: bytes-like

Exact requirements of the tag are described by the documentation of individual modes.

class **cryptography.hazmat.primitives.ciphers.modes.ModeWithTweak**

New in version 2.1.

A cipher mode with a tweak.

tweak

Type: bytes-like

Exact requirements of the tweak are described by the documentation of individual modes.

Exceptions

class **cryptography.exceptions.InvalidTag**

This is raised if an authenticated encryption tag fails to verify during decryption.