Cryptography Documentation

Release 51.0.0-dev1

Individual Contributors

Aug 31, 2026

THE RECIPES LAYER

1 Installation **3**

2 Layout **5**

2.1 Fernet (symmetric encryption) . . . 5
2.2 Cobblestone (streaming symmetric encryption) . . . 9
2.3 X.509 . . . 12
2.4 Primitives . . . 100
2.5 Exceptions . . . 267
2.6 Random number generation . . . 268
2.7 ASN.1 . . . 268
2.8 Decrepit cryptography . . . 282
2.9 Installation . . . 285
2.10 Changelog . . . 289
2.11 Frequently asked questions . . . 326
2.12 Development . . . 329
2.13 Use of OpenSSL . . . 393
2.14 Security . . . 394
2.15 Known security limitations . . . 395
2.16 API stability . . . 396
2.17 Doing a release . . . 397
2.18 Community . . . 399
2.19 Glossary . . . 399
2.20 Statements . . . 400

Python Module Index **405**

Index **407**

cryptography includes both high level recipes and low level interfaces to common cryptographic algorithms such as symmetric ciphers, message digests, and key derivation functions. For example, to encrypt something with cryptography's high level symmetric encryption recipe:

```
>>> from cryptography.fernet import Fernet
>>> # Put this somewhere safe!
>>> key = Fernet.generate_key()
>>> f = Fernet(key)
>>> token = f.encrypt(b"A really secret message. Not for prying eyes.")
>>> token
b'...'
>>> f.decrypt(token)
b'A really secret message. Not for prying eyes.'
```

If you are interested in learning more about the field of cryptography, we recommend Crypto 101, by Laurens Van Houtven and The Cryptopals Crypto Challenges.

CHAPTER

ONE

INSTALLATION

To install `cryptography`:

```
$ pip install cryptography
```

```
$ uv add cryptography
```

See *Installation* for more information.

CHAPTER

TWO

LAYOUT

cryptography is broadly divided into two levels. One with safe cryptographic recipes that require little to no configuration choices. These are safe and easy to use and don't require developers to make many decisions.

The other level is low-level cryptographic primitives. These are often dangerous and can be used incorrectly. They require making decisions and having an in-depth knowledge of the cryptographic concepts at work. Because of the potential danger in working at this level, this is referred to as the "hazardous materials" or "hazmat" layer. These live in the cryptography.hazmat package, and their documentation will always contain an admonition at the top.

We recommend using the recipes layer whenever possible, and falling back to the hazmat layer only when necessary.

2.1 Fernet (symmetric encryption)

Fernet guarantees that a message encrypted using it cannot be manipulated or read without the key. Fernet is an implementation of symmetric (also known as "secret key") authenticated cryptography. Fernet also has support for implementing key rotation via *MultiFernet*.

class cryptography.fernet.**Fernet**(*key*)

This class provides both encryption and decryption facilities. This class exhibits *thread safety*.

```
>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> f = Fernet(key)
>>> token = f.encrypt(b"my deep dark secret")
>>> token
b'...'
>>> f.decrypt(token)
b'my deep dark secret'
```

Parameters

key (*bytes or str*) – A URL-safe base64-encoded 32-byte key. This **must** be kept secret. Anyone with this key is able to create and read messages.

classmethod generate_key()

Generates a fresh fernet key. Keep this some place safe! If you lose it you'll no longer be able to decrypt messages; if anyone else gains access to it, they'll be able to decrypt all of your messages, and they'll also be able to forge arbitrary messages that will be authenticated and decrypted.

encrypt(*data*)

Encrypts data passed. The result of this encryption is known as a "Fernet token" and has strong privacy and authenticity guarantees.

Parameters
data (*bytes*) – The message you would like to encrypt.

Returns bytes
A secure message that cannot be read or altered without the key. It is URL-safe base64-encoded. This is referred to as a "Fernet token".

Raises
TypeError – This exception is raised if `data` is not `bytes`.

> **Note**
>
> The encrypted message contains the current time when it was generated in *plaintext*, the time a message was created will therefore be visible to a possible attacker.

encrypt_at_time(*data*, *current_time*)

Added in version 3.0.

Encrypts data passed using explicitly passed current time. See *encrypt()* for the documentation of the `data` parameter, the return type and the exceptions raised.

The motivation behind this method is for the client code to be able to test token expiration. Since this method can be used in an insecure manner one should make sure the correct time (`int(time.time())`) is passed as `current_time` outside testing.

Parameters
current_time (*int*) – The current time.

> **Note**
>
> Similarly to *encrypt()* the encrypted message contains the timestamp in *plaintext*, in this case the timestamp is the value of the `current_time` parameter.

decrypt(*token*, *ttl=None*)

Decrypts a Fernet token. If successfully decrypted you will receive the original plaintext as the result, otherwise an exception will be raised. It is safe to use this data immediately as Fernet verifies that the data has not been tampered with prior to returning it.

Parameters

- **token** (*bytes or str*) – The Fernet token. This is the result of calling *encrypt()*.
- **ttl** (*int*) – Optionally, the number of seconds old a message may be for it to be valid. If the message is older than `ttl` seconds (from the time it was originally created) an exception will be raised. If `ttl` is not provided (or is `None`), the age of the message is not considered.

Returns bytes
The original plaintext.

Raises

- *cryptography.fernet.InvalidToken* – If the `token` is in any way invalid, this exception is raised. A token may be invalid for a number of reasons: it is older than the `ttl`, it is malformed, or it does not have a valid signature.
- **TypeError** – This exception is raised if `token` is not `bytes` or `str`.

decrypt_at_time(*token*, *ttl*, *current_time*)

Added in version 3.0.

Decrypts a token using explicitly passed current time. See *decrypt()* for the documentation of the `token` and `ttl` parameters (`ttl` is required here), the return type and the exceptions raised.

The motivation behind this method is for the client code to be able to test token expiration. Since this method can be used in an insecure manner one should make sure the correct time (`int(time.time())`) is passed as `current_time` outside testing.

Parameters
current_time (*int*) – The current time.

extract_timestamp(*token*)

Added in version 2.3.

Returns the timestamp for the token. The caller can then decide if the token is about to expire and, for example, issue a new token.

Parameters
token (*bytes or str*) – The Fernet token. This is the result of calling *encrypt()*.

Returns int
The Unix timestamp of the token.

Raises

- *cryptography.fernet.InvalidToken* – If the `token`'s signature is invalid this exception is raised.
- **TypeError** – This exception is raised if `token` is not `bytes` or `str`.

class cryptography.fernet.**MultiFernet**(*fernets*)

Added in version 0.7.

This class implements key rotation for Fernet. It takes a `list` of *Fernet* instances and implements the same API with the exception of one additional method: *MultiFernet.rotate()*:

```
>>> from cryptography.fernet import Fernet, MultiFernet
>>> key1 = Fernet(Fernet.generate_key())
>>> key2 = Fernet(Fernet.generate_key())
>>> f = MultiFernet([key1, key2])
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
```

MultiFernet performs all encryption options using the *first* key in the `list` provided. MultiFernet attempts to decrypt tokens with each key in turn. A *cryptography.fernet.InvalidToken* exception is raised if the correct key is not found in the `list` provided.

Key rotation makes it easy to replace old keys. You can add your new key at the front of the list to start encrypting new messages, and remove old keys as they are no longer needed.

Token rotation as offered by *MultiFernet.rotate()* is a best practice and manner of cryptographic hygiene designed to limit damage in the event of an undetected event and to increase the difficulty of attacks. For example, if an employee who had access to your company's fernet keys leaves, you'll want to generate new fernet key, rotate all of the tokens currently deployed using that new key, and then retire the old fernet key(s) to which the employee had access.

rotate(*msg*)

Added in version 2.2.

Rotates a token by re-encrypting it under the *MultiFernet* instance's primary key. This preserves the timestamp that was originally saved with the token. If a token has successfully been rotated then the rotated token will be returned. If rotation fails this will raise an exception.

```
>>> from cryptography.fernet import Fernet, MultiFernet
>>> key1 = Fernet(Fernet.generate_key())
>>> key2 = Fernet(Fernet.generate_key())
>>> f = MultiFernet([key1, key2])
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
>>> key3 = Fernet(Fernet.generate_key())
>>> f2 = MultiFernet([key3, key1, key2])
>>> rotated = f2.rotate(token)
>>> f2.decrypt(rotated)
b'Secret message!'
```

Parameters

msg (*bytes or str*) – The token to re-encrypt.

Returns bytes

A secure message that cannot be read or altered without the key. This is URL-safe base64-encoded. This is referred to as a "Fernet token".

Raises

- *cryptography.fernet.InvalidToken* – If a `token` is in any way invalid this exception is raised.
- **TypeError** – This exception is raised if the `msg` is not `bytes` or `str`.

class cryptography.fernet.**InvalidToken**

See *Fernet.decrypt()* for more information.

2.1.1 Using passwords with Fernet

It is possible to use passwords with Fernet. To do this, you need to run the password through a key derivation function. `cryptography` provides several such functions; it is generally recommended to use *Argon2id*.

```
>>> import base64
>>> import os
>>> from cryptography.fernet import Fernet
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.argon2 import Argon2id
>>> password = b"password"
>>> salt = os.urandom(16)
>>> kdf = Argon2id(
...     salt=salt,
...     length=32,
...     iterations=1,
```

(continues on next page)

(continued from previous page)

```
...     lanes=4,
...     memory_cost=2**21
... )
>>> key = base64.urlsafe_b64encode(kdf.derive(password))
>>> f = Fernet(key)
>>> token = f.encrypt(b"Secret message!")
>>> token
b'...'
>>> f.decrypt(token)
b'Secret message!'
```

In this scheme, the salt has to be stored in a retrievable location in order to derive the same key from the password in the future.

The *Argon2id* parameters in the above code example are based on the recommendations of IRTF RFC 9106 for general applications. For memory-constrained applications, the RFC recommends `iterations=3` and `memory_cost=2**16`. See that document for more information.

2.1.2 Implementation

Fernet is built on top of a number of standard cryptographic primitives. Specifically it uses:

- *AES* in *CBC* mode with a 128-bit key for encryption; using *PKCS7* padding.
- *HMAC* using *SHA256* for authentication.
- Initialization vectors are generated using `os.urandom()`.

For complete details consult the specification.

2.1.3 Limitations

Fernet is ideal for encrypting data that easily fits in memory. As a design feature it does not expose unauthenticated bytes. This means that the complete message contents must be available in memory, making Fernet generally unsuitable for very large files at this time.

2.2 Cobblestone (streaming symmetric encryption)

Cobblestone provides authenticated symmetric encryption of large messages — up to 4 PiB — as a stream, without ever holding the whole message in memory. It is an implementation of the C2SP chunked-encryption specification's two named instantiations: **Cobblestone-128** (SHA-512 and AES-128-GCM, the recommended choice) and **Cobblestone-256** (SHA-512 and AES-256-GCM, for environments that mandate 256-bit keys).

```
>>> from cryptography.cobblestone import (
...     Cobblestone128Decryptor, Cobblestone128Encryptor
... )
>>> key = Cobblestone128Encryptor.generate_key()
>>> encryptor = Cobblestone128Encryptor(
...     key, context=b"example-app file encryption"
... )
>>> ciphertext = encryptor.update(b"a secret message")
>>> ciphertext += encryptor.finalize()
>>> decryptor = Cobblestone128Decryptor(
...     key, context=b"example-app file encryption"
```

(continues on next page)

(continued from previous page)

```
... )
>>> decryptor.update(ciphertext) + decryptor.finalize()
b'a secret message'
```

class cryptography.cobblestone.**Cobblestone128Encryptor**(*key*, *context*)

Added in version 50.0.0.

Encrypts a single message under `key` with Cobblestone-128. Each instance must be used for exactly one message: call *update()* (or *update_into()*) any number of times, then call *finalize()* exactly once. The concatenation of the returned bytes is the ciphertext.

Parameters

- **key** (*bytes-like*) – A 16-byte key. This **must** be kept secret, and **must** be uniformly random (e.g. the output of *generate_key()* — never a password). A single key may be used to encrypt a practically unlimited number of messages.
- **context** (*bytes-like*) – Application-provided context, bound to the ciphertext. Decryption fails unless the same value is passed to *Cobblestone128Decryptor*. It is not secret, may be empty, and is not part of the ciphertext, so it must be available to the decrypting party independently. It can be used for domain separation, e.g. `b"myapp v2 backup encryption"`.

Raises

`ValueError` – If `key` is not 16 bytes.

static generate_key()

Generates a fresh 16-byte key.

Return bytes

A new key.

update(*data*)

Encrypts `data`. Data is internally buffered into 16 KiB chunks, so between 0 and `len(data) + 16 KiB` bytes of ciphertext are returned.

Parameters

data (*bytes-like*) – The data to encrypt.

Return bytes

The next portion of the ciphertext.

update_into(*data*, *buf*)

Encrypts `data`, writing the resulting ciphertext into `buf`, and returns the number of bytes written. This avoids allocating a new buffer for each call.

Parameters

- **data** (*bytes-like*) – The data to encrypt.
- **buf** (*bytes-like*) – A writable buffer to write the ciphertext into. A buffer of `len(data) + len(data) // 1024 + 16456` bytes is always large enough.

Return int

The number of bytes written to `buf`.

Raises

`ValueError` – If `buf` is too small.

finalize()

Encrypts the final chunk and returns the last portion of the ciphertext. This must always be called, and the instance cannot be used afterwards.

Return bytes
The remainder of the ciphertext.

Raises
cryptography.exceptions.AlreadyFinalized – If `finalize` has already been called.

class cryptography.cobblestone.**Cobblestone128Decryptor**(*key*, *context*)

Added in version 50.0.0.

Decrypts a single message encrypted by *Cobblestone128Encryptor* with the same `key` and `context`. Call *update()* (or *update_into()*) with the ciphertext any number of times, then call *finalize()* exactly once. The concatenation of the returned bytes is the plaintext.

Any returned plaintext is authenticated, but until *finalize()* returns successfully the message could still turn out to be truncated: an application acting on streamed plaintext before that point must be prepared to discard its work if a later call raises *InvalidTag*.

Once any method raises *InvalidTag*, the instance is permanently unusable and all further calls raise *AlreadyFinalized*.

Parameters

- **key** (*bytes-like*) – The 16-byte key the message was encrypted with.
- **context** (*bytes-like*) – The context value the message was encrypted with.

Raises
ValueError – If `key` is not 16 bytes.

update(*data*)

Processes `data`, which need not be aligned to any boundary, and returns the plaintext of all complete chunks that have been authenticated so far.

Parameters
data (*bytes-like*) – The next portion of the ciphertext.

Return bytes
The next portion of the plaintext.

Raises
cryptography.exceptions.InvalidTag – If the ciphertext was encrypted with a different key or context, or has been modified.

update_into(*data*, *buf*)

Like `update`, but writes the plaintext into `buf` and returns the number of bytes written.

Parameters

- **data** (*bytes-like*) – The next portion of the ciphertext.
- **buf** (*bytes-like*) – A writable buffer to write the plaintext into. A buffer of `len(data) + 16400` bytes is always large enough.

Return int
The number of bytes written to `buf`.

Raises

- **ValueError** – If `buf` is too small.

- *cryptography.exceptions.InvalidTag* – If the ciphertext was encrypted with a different key or context, or has been modified. Note that in this case unauthenticated data may have been written to `buf` and must not be used.

finalize()

Decrypts and authenticates the final chunk, verifying that the entire message has been processed, and returns the final portion of the plaintext. This must always be called: a successful return is what guarantees the complete message was authentic and not truncated.

Return bytes
The remainder of the plaintext.

Raises

- *cryptography.exceptions.InvalidTag* – If the ciphertext was truncated or otherwise modified.
- *cryptography.exceptions.AlreadyFinalized* – If `finalize` has already been called.

class cryptography.cobblestone.**Cobblestone256Encryptor**(*key*, *context*)

Added in version 50.0.0.

Exactly like *Cobblestone128Encryptor*, but implements Cobblestone-256: the `key` is 32 bytes and messages are encrypted with AES-256-GCM. Use this when a 256-bit key is mandated; otherwise Cobblestone-128 is recommended.

class cryptography.cobblestone.**Cobblestone256Decryptor**(*key*, *context*)

Added in version 50.0.0.

Exactly like *Cobblestone128Decryptor*, but decrypts messages produced by *Cobblestone256Encryptor* with a 32-byte key.

2.3 X.509

X.509 is a standard for public key infrastructure. `cryptography` implements X.509 in accordance with **RFC 5280**, and is principally focused on WebPKI use cases. X.509 certificates are commonly used in protocols like TLS.

In some cases we tolerate divergences from the these specifications, however these should be regarded as exceptional cases.

2.3.1 Tutorial

X.509 certificates are used to authenticate clients and servers. The most common use case is for web servers using HTTPS.

Creating a Certificate Signing Request (CSR)

When obtaining a certificate from a certificate authority (CA), the usual flow is:

1. You generate a private/public key pair.
2. You create a request for a certificate, which is signed by your key (to prove that you own that key).
3. You give your CSR to a CA (but *not* the private key).
4. The CA validates that you own the resource (e.g. domain) you want a certificate for.
5. The CA gives you a certificate, signed by them, which identifies your public key, and the resource you are authenticated for.

6. You configure your server to use that certificate, combined with your private key, to server traffic.

If you want to obtain a certificate from a typical commercial CA, here's how. First, you'll need to generate a private key, we'll generate an RSA key (these are the most common types of keys on the web right now):

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> # Generate our key
>>> key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> # Write our key to disk for safe keeping
>>> with open("path/to/store/key.pem", "wb") as f:
...     f.write(key.private_bytes(
...         encoding=serialization.Encoding.PEM,
...         format=serialization.PrivateFormat.TraditionalOpenSSL,
...         encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),
...     ))
```

If you've already generated a key you can load it with *load_pem_private_key()*.

Next we need to generate a certificate signing request. A typical CSR contains a few details:

- Information about our public key (including a signature of the entire body).
- Information about who *we* are.
- Information about what domains this certificate is for.

```
>>> from cryptography import x509
>>> from cryptography.x509.oid import NameOID
>>> from cryptography.hazmat.primitives import hashes
>>> # Generate a CSR
>>> csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
...     # Provide various details about who we are.
...     x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, "mysite.com"),
... ])).add_extension(
...     x509.SubjectAlternativeName([
...         # Describe what sites we want this certificate for.
...         x509.DNSName("mysite.com"),
...         x509.DNSName("www.mysite.com"),
...         x509.DNSName("subdomain.mysite.com"),
...     ]),
...     critical=False,
... # Sign the CSR with our private key.
... ).sign(key, hashes.SHA256())
>>> # Write our CSR out to disk.
>>> with open("path/to/csr.pem", "wb") as f:
...     f.write(csr.public_bytes(serialization.Encoding.PEM))
```

Now we can give our CSR to a CA, who will give a certificate to us in return.

Creating a self-signed certificate

While most of the time you want a certificate that has been *signed* by someone else (i.e. a certificate authority), so that trust is established, sometimes you want to create a self-signed certificate. Self-signed certificates are not issued by a certificate authority, but instead they are signed by the private key corresponding to the public key they embed.

This means that other people don't trust these certificates, but it also means they can be issued very easily. In general the only use case for a self-signed certificate is local testing, where you don't need anyone else to trust your certificate.

Like generating a CSR, we start with creating a new private key:

```
>>> # Generate our key
>>> key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> # Write our key to disk for safe keeping
>>> with open("path/to/store/key.pem", "wb") as f:
...     f.write(key.private_bytes(
...         encoding=serialization.Encoding.PEM,
...         format=serialization.PrivateFormat.TraditionalOpenSSL,
...         encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),
...     ))
```

Then we generate the certificate itself:

```
>>> import datetime
>>> # Various details about who we are. For a self-signed certificate the
>>> # subject and issuer are always the same.
>>> subject = issuer = x509.Name([
...     x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, "mysite.com"),
... ])
>>> cert = x509.CertificateBuilder().subject_name(
...     subject
... ).issuer_name(
...     issuer
... ).public_key(
...     key.public_key()
... ).serial_number(
...     x509.random_serial_number()
... ).not_valid_before(
...     datetime.datetime.now(datetime.timezone.utc)
... ).not_valid_after(
...     # Our certificate will be valid for 10 days
...     datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=10)
... ).add_extension(
...     x509.SubjectAlternativeName([x509.DNSName("localhost")]),
...     critical=False,
... # Sign our certificate with our private key
... ).sign(key, hashes.SHA256())
>>> # Write our certificate out to disk.
```

(continues on next page)

(continued from previous page)

```
>>> with open("path/to/certificate.pem", "wb") as f:
...     f.write(cert.public_bytes(serialization.Encoding.PEM))
```

And now we have a private key and certificate that can be used for local testing.

Creating a CA hierarchy

When building your own root hierarchy you need to generate a CA and then issue certificates (typically intermediates) using it. This example shows how to generate a root CA, a signing intermediate, and issues a leaf certificate off that intermediate. X.509 is a complex specification so this example will require adaptation (typically different extensions) for specific operating environments.

Note that this example does not add CRL distribution point or OCSP AIA extensions, nor does it save the key/certs to persistent storage.

```
>>> import datetime
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.x509.oid import NameOID
>>> from cryptography import x509
>>> # Generate our key
>>> root_key = ec.generate_private_key(ec.SECP256R1())
>>> subject = issuer = x509.Name([
...     x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, "PyCA Docs Root CA"),
... ])
>>> root_cert = x509.CertificateBuilder().subject_name(
...     subject
... ).issuer_name(
...     issuer
... ).public_key(
...     root_key.public_key()
... ).serial_number(
...     x509.random_serial_number()
... ).not_valid_before(
...     datetime.datetime.now(datetime.timezone.utc)
... ).not_valid_after(
...     # Our certificate will be valid for ~10 years
...     datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=365*10)
... ).add_extension(
...     x509.BasicConstraints(ca=True, path_length=None),
...     critical=True,
... ).add_extension(
...     x509.KeyUsage(
...         digital_signature=True,
...         content_commitment=False,
...         key_encipherment=False,
...         data_encipherment=False,
...         key_agreement=False,
...         key_cert_sign=True,
```

(continues on next page)

(continued from previous page)

```
...         crl_sign=True,
...         encipher_only=False,
...         decipher_only=False,
...     ),
...     critical=True,
... ).add_extension(
...     x509.SubjectKeyIdentifier.from_public_key(root_key.public_key()),
...     critical=False,
... ).sign(root_key, hashes.SHA256())
```

With a root certificate created we now want to create our intermediate.

```
>>> # Generate our intermediate key
>>> int_key = ec.generate_private_key(ec.SECP256R1())
>>> subject = x509.Name([
...     x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, "PyCA Docs Intermediate CA"),
... ])
>>> int_cert = x509.CertificateBuilder().subject_name(
...     subject
... ).issuer_name(
...     root_cert.subject
... ).public_key(
...     int_key.public_key()
... ).serial_number(
...     x509.random_serial_number()
... ).not_valid_before(
...     datetime.datetime.now(datetime.timezone.utc)
... ).not_valid_after(
...     # Our intermediate will be valid for ~3 years
...     datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=365*3)
... ).add_extension(
...     # Allow no further intermediates (path length 0)
...     x509.BasicConstraints(ca=True, path_length=0),
...     critical=True,
... ).add_extension(
...     x509.KeyUsage(
...         digital_signature=True,
...         content_commitment=False,
...         key_encipherment=False,
...         data_encipherment=False,
...         key_agreement=False,
...         key_cert_sign=True,
...         crl_sign=True,
...         encipher_only=False,
...         decipher_only=False,
...     ),
...     critical=True,
... ).add_extension(
```

(continues on next page)

(continued from previous page)

```
...     x509.SubjectKeyIdentifier.from_public_key(int_key.public_key()),
...     critical=False,
... ).add_extension(
...     x509.AuthorityKeyIdentifier.from_issuer_subject_key_identifier(
...         root_cert.extensions.get_extension_for_class(x509.SubjectKeyIdentifier).value
...     ),
...     critical=False,
... ).sign(root_key, hashes.SHA256())
```

Now we can issue an end entity certificate off this chain.

```
>>> ee_key = ec.generate_private_key(ec.SECP256R1())
>>> subject = x509.Name([
...     x509.NameAttribute(NameOID.COUNTRY_NAME, "US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, "San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, "My Company"),
... ])
>>> ee_cert = x509.CertificateBuilder().subject_name(
...     subject
... ).issuer_name(
...     int_cert.subject
... ).public_key(
...     ee_key.public_key()
... ).serial_number(
...     x509.random_serial_number()
... ).not_valid_before(
...     datetime.datetime.now(datetime.timezone.utc)
... ).not_valid_after(
...     # Our cert will be valid for 10 days
...     datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=10)
... ).add_extension(
...     x509.SubjectAlternativeName([
...         # Describe what sites we want this certificate for.
...         x509.DNSName("cryptography.io"),
...         x509.DNSName("www.cryptography.io"),
...     ]),
...     critical=False,
... ).add_extension(
...     x509.BasicConstraints(ca=False, path_length=None),
...     critical=True,
... ).add_extension(
...     x509.KeyUsage(
...         digital_signature=True,
...         content_commitment=False,
...         key_encipherment=True,
...         data_encipherment=False,
...         key_agreement=False,
...         key_cert_sign=False,
...         crl_sign=True,
...         encipher_only=False,
...         decipher_only=False,
```

(continues on next page)

(continued from previous page)

```
...     ),
...     critical=True,
... ).add_extension(
...     x509.ExtendedKeyUsage([
...         x509.ExtendedKeyUsageOID.CLIENT_AUTH,
...         x509.ExtendedKeyUsageOID.SERVER_AUTH,
...     ]),
...     critical=False,
... ).add_extension(
...     x509.SubjectKeyIdentifier.from_public_key(ee_key.public_key()),
...     critical=False,
... ).add_extension(
...     x509.AuthorityKeyIdentifier.from_issuer_subject_key_identifier(
...         int_cert.extensions.get_extension_for_class(x509.SubjectKeyIdentifier).value
...     ),
...     critical=False,
... ).sign(int_key, hashes.SHA256())
```

And finally we use the verification APIs to validate the chain.

```
>>> from cryptography.x509 import DNSName
>>> from cryptography.x509.verification import PolicyBuilder, Store
>>> store = Store([root_cert])
>>> builder = PolicyBuilder().store(store)
>>> verifier = builder.build_server_verifier(DNSName("cryptography.io"))
>>> chain = verifier.verify(ee_cert, [int_cert])
>>> len(chain)
3
```

Determining Certificate or Certificate Signing Request Key Type

Certificates and certificate signing requests can be issued with multiple key types. You can determine what the key type is by using `isinstance` checks:

```
>>> public_key = cert.public_key()
>>> if isinstance(public_key, rsa.RSAPublicKey):
...     # Do something RSA specific
... elif isinstance(public_key, ec.EllipticCurvePublicKey):
...     # Do something EC specific
... else:
...     # Remember to handle this case
```

2.3.2 Certificate Transparency

Certificate Transparency is a set of protocols specified in **RFC 6962** which allow X.509 certificates to be sent to append-only logs and have small cryptographic proofs that a certificate has been publicly logged. This allows for external auditing of the certificates that a certificate authority has issued.

class cryptography.x509.certificate_transparency.**SignedCertificateTimestamp**

Added in version 2.0.

SignedCertificateTimestamps (SCTs) are small cryptographically signed assertions that the specified certificate has been submitted to a Certificate Transparency Log, and that it will be part of the public log within some time period, this is called the "maximum merge delay" (MMD) and each log specifies its own.

version

Type
Version

The SCT version as an enumeration. Currently only one version has been specified.

log_id

Type
bytes

An opaque identifier, indicating which log this SCT is from. This is the SHA256 hash of the log's public key.

timestamp

Type
datetime.datetime

A naïve datetime representing the time in UTC at which the log asserts the certificate had been submitted to it.

entry_type

Type
LogEntryType

The type of submission to the log that this SCT is for. Log submissions can either be certificates themselves or "pre-certificates" which indicate a binding-intent to issue a certificate for the same data, with SCTs embedded in it.

signature_hash_algorithm

Added in version 38.0.0.

Type
HashAlgorithm

The hashing algorithm used by this SCT's signature.

signature_algorithm

Added in version 38.0.0.

Type
SignatureAlgorithm

The signing algorithm used by this SCT's signature.

signature

Added in version 38.0.0.

Type
bytes

The raw bytes of the signatures embedded in the SCT.

extension_bytes

Added in version 38.0.0.

Type
bytes

Any raw extension bytes.

class cryptography.x509.certificate_transparency.**Version**

Added in version 2.0.

An enumeration for SignedCertificateTimestamp versions.

v1

For version 1 SignedCertificateTimestamps.

class cryptography.x509.certificate_transparency.**LogEntryType**

Added in version 2.0.

An enumeration for SignedCertificateTimestamp log entry types.

X509_CERTIFICATE

For SCTs corresponding to X.509 certificates.

PRE_CERTIFICATE

For SCTs corresponding to pre-certificates.

class cryptography.x509.certificate_transparency.**SignatureAlgorithm**

Added in version 38.0.0.

An enumeration for SignedCertificateTimestamp signature algorithms.

These are exactly the same as SignatureAlgorithm in **RFC 5246** (TLS 1.2).

ANONYMOUS

RSA

DSA

ECDSA

2.3.3 OCSP

OCSP (Online Certificate Status Protocol) is a method of checking the revocation status of certificates. It is specified in **RFC 6960**, as well as other obsoleted RFCs.

Loading Requests

cryptography.x509.ocsp.**load_der_ocsp_request**(*data*)

Added in version 2.4.

Deserialize an OCSP request from DER encoded data.

Parameters
data (*bytes*) – The DER encoded OCSP request data.

Returns
An instance of *OCSPRequest*.

```
>>> from cryptography.x509 import ocsp
>>> ocsp_req = ocsp.load_der_ocsp_request(der_ocsp_req)
>>> print(ocsp_req.serial_number)
872625873161273451176241581705670534707360122361
```

Creating Requests

class cryptography.x509.ocsp.**OCSPRequestBuilder**

Added in version 2.4.

This class is used to create *OCSPRequest* objects.

add_certificate(*cert*, *issuer*, *algorithm*)

Adds a request using a certificate, issuer certificate, and hash algorithm. You can call this method or `add_certificate_by_hash` only once.

Parameters

- **cert** – The *Certificate* whose validity is being checked.
- **issuer** – The issuer *Certificate* of the certificate that is being checked.
- **algorithm** – A *HashAlgorithm* instance. For OCSP only *SHA1*, *SHA224*, *SHA256*, *SHA384*, and *SHA512* are allowed.

add_certificate_by_hash(*issuer_name_hash*, *issuer_key_hash*, *serial_number*, *algorithm*)

Added in version 39.0.0.

Adds a request using the issuer's name hash, key hash, the certificate serial number, and hash algorithm. You can call this method or `add_certificate` only once.

Parameters

- **issuer_name_hash** (*bytes*) – The hash of the issuer's DER encoded name using the same hash algorithm as the one specified in the `algorithm` parameter.
- **issuer_key_hash** (*bytes*) – The hash of the issuer's public key bit string DER encoding using the same hash algorithm as the one specified in the `algorithm` parameter.
- **serial_number** (*int*) – The serial number of the certificate being checked.
- **algorithm** – A *HashAlgorithm* instance. For OCSP only *SHA1*, *SHA224*, *SHA256*, *SHA384*, and *SHA512* are allowed.

add_extension(*extval*, *critical*)

Adds an extension to the request.

Parameters

- **extval** – An extension conforming to the *ExtensionType* interface.
- **critical** – Set to `True` if the extension must be understood and handled.

build()

Returns

A new *OCSPRequest*.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.hashes import SHA256
>>> from cryptography.x509 import load_pem_x509_certificate, ocsp
>>> cert = load_pem_x509_certificate(pem_cert)
>>> issuer = load_pem_x509_certificate(pem_issuer)
>>> builder = ocsp.OCSPRequestBuilder()
>>> # SHA256 is in this example because while RFC 5019 originally
>>> # required SHA1 RFC 6960 updates that to SHA256.
>>> # However, depending on your requirements you may need to use SHA1
```

(continues on next page)

(continued from previous page)

```
>>> # for compatibility reasons.
>>> builder = builder.add_certificate(cert, issuer, SHA256())
>>> req = builder.build()
>>> base64.b64encode(req.public_bytes(serialization.Encoding.DER))
b
↪'MF8wXTBbMFkwVzANBglghkgBZQMEAgEFAAQgn3BowBaoh77h17ULfkX6781dUDPD82Taj8wO1jZWhZoEINxPgjoQth3w7q4A
↪IA=='
```

Loading Responses

`cryptography.x509.ocsp.`**`load_der_ocsp_response`**(*data*)

Added in version 2.4.

Deserialize an OCSP response from DER encoded data.

Parameters

data (*bytes*) – The DER encoded OCSP response data.

Returns

An instance of *OCSPResponse*.

```
>>> from cryptography.x509 import ocsp
>>> ocsp_resp = ocsp.load_der_ocsp_response(der_ocsp_resp_unauth)
>>> print(ocsp_resp.response_status)
OCSPResponseStatus.UNAUTHORIZED
```

Creating Responses

class `cryptography.x509.ocsp.`**`OCSPResponseBuilder`**

Added in version 2.4.

This class is used to create *OCSPResponse* objects. You cannot set `produced_at` on OCSP responses at this time. Instead the field is set to current UTC time when calling `sign`. For unsuccessful statuses call the class method *build_unsuccessful()*.

add_response(*cert*, *issuer*, *algorithm*, *cert_status*, *this_update*, *next_update*, *revocation_time*, *revocation_reason*)

This method adds status information about the certificate that was requested to the response. You can call this method or `add_response_by_hash` only once.

Parameters

- **cert** – The *Certificate* whose validity is being checked.
- **issuer** – The issuer *Certificate* of the certificate that is being checked.
- **algorithm** – A *HashAlgorithm* instance. For OCSP only *SHA1*, *SHA224*, *SHA256*, *SHA384*, and *SHA512* are allowed.
- **cert_status** – An item from the *OCSPCertStatus* enumeration.
- **this_update** – A `datetime.datetime` object representing the most recent time at which the status being indicated is known by the responder to be correct. If it does not have a timezone, it is assumed to be in UTC.
- **next_update** – A `datetime.datetime` object or `None`. The time at or before which newer information will be available about the status of the certificate. If it does not have a timezone, it is assumed to be in UTC.

- **revocation_time** – A datetime.datetime object or None if the cert is not revoked. The time at which the certificate was revoked. If it does not have a timezone, it is assumed to be in UTC.
- **revocation_reason** – An item from the *ReasonFlags* enumeration or None if the cert is not revoked.

add_response_by_hash(*issuer_name_hash*, *issuer_key_hash*, *serial_number*, *algorithm*, *cert_status*, *this_update*, *next_update*, *revocation_time*, *revocation_reason*)

Added in version 45.0.0.

This method adds status information about the certificate that was requested to the response using the hash values directly, rather than requiring the full certificates. You can call this method or add_response only once.

Parameters

- **issuer_name_hash** (*bytes*) – The hash of the issuer's DER encoded name using the same hash algorithm as the one specified in the algorithm parameter.
- **issuer_key_hash** (*bytes*) – The hash of the issuer's public key bit string DER encoding using the same hash algorithm as the one specified in the algorithm parameter.
- **serial_number** (*int*) – The serial number of the certificate being checked.
- **algorithm** – A *HashAlgorithm* instance. For OCSP only *SHA1*, *SHA224*, *SHA256*, *SHA384*, and *SHA512* are allowed.
- **cert_status** – An item from the *OCSPCertStatus* enumeration.
- **this_update** – A datetime.datetime object representing the most recent time at which the status being indicated is known by the responder to be correct. If it does not have a timezone, it is assumed to be in UTC.
- **next_update** – A datetime.datetime object or None. The time at or before which newer information will be available about the status of the certificate. If it does not have a timezone, it is assumed to be in UTC.
- **revocation_time** – A datetime.datetime object or None if the cert is not revoked. The time at which the certificate was revoked. If it does not have a timezone, it is assumed to be in UTC.
- **revocation_reason** – An item from the *ReasonFlags* enumeration or None if the cert is not revoked.

certificates(*certs*)

Add additional certificates that should be used to verify the signature on the response. This is typically used when the responder utilizes an OCSP delegate.

Parameters

certs (*list*) – A list of *Certificate* objects.

responder_id(*encoding*, *responder_cert*)

Set the responderID on the OCSP response. This is the data a client will use to determine what certificate signed the response.

Parameters

- **responder_cert** – The *Certificate* object for the certificate whose private key will sign the OCSP response. If the certificate and key do not match an error will be raised when calling sign.
- **encoding** – Either *HASH* or *NAME*.

add_extension(*extval*, *critical*)

Adds an extension to the response.

Parameters

- **extval** – An extension conforming to the *ExtensionType* interface.
- **critical** – Set to True if the extension must be understood and handled.

sign(*private_key*, *algorithm*)

Creates the OCSP response that can then be serialized and sent to clients. This method will create a *SUCCESSFUL* response.

Parameters

- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey*, *EllipticCurvePrivateKey*, *Ed25519PrivateKey* or *Ed448PrivateKey* that will be used to sign the response.
- **algorithm** – The *HashAlgorithm* that will be used to generate the signature. This must be None if the private_key is an *Ed25519PrivateKey* or an *Ed448PrivateKey* and an instance of a *HashAlgorithm* otherwise. Please note that *SHA1* can not be used here, regardless of if it was used for *add_response()* or not.

Returns

A new *OCSPResponse*.

```
>>> import datetime
>>> from cryptography.hazmat.primitives import hashes, serialization
>>> from cryptography.x509 import load_pem_x509_certificate, ocsp
>>> cert = load_pem_x509_certificate(pem_cert)
>>> issuer = load_pem_x509_certificate(pem_issuer)
>>> responder_cert = load_pem_x509_certificate(pem_responder_cert)
>>> responder_key = serialization.load_pem_private_key(pem_responder_key, None)
>>> builder = ocsp.OCSPResponseBuilder()
>>> # SHA256 is in this example because while RFC 5019 originally
>>> # required SHA1 RFC 6960 updates that to SHA256.
>>> # However, depending on your requirements you may need to use SHA1
>>> # for compatibility reasons.
>>> builder = builder.add_response(
...     cert=cert, issuer=issuer, algorithm=hashes.SHA256(),
...     cert_status=ocsp.OCSPCertStatus.GOOD,
...     this_update=datetime.datetime.now(),
...     next_update=datetime.datetime.now(),
...     revocation_time=None, revocation_reason=None
... ).responder_id(
...     ocsp.OCSPResponderEncoding.HASH, responder_cert
... )
>>> response = builder.sign(responder_key, hashes.SHA256())
>>> response.certificate_status
<OCSPCertStatus.GOOD: 0>
```

classmethod build_unsuccessful(*response_status*)

Creates an unsigned OCSP response which can then be serialized and sent to clients. build_unsuccessful may only be called with a *OCSPResponseStatus* that is not SUCCESSFUL. Since this is a class method note that no other methods can or should be called as unsuccessful statuses do not encode additional data.

Returns
A new *OCSPResponse*.

```
>>> from cryptography.hazmat.primitives import hashes, serialization
>>> from cryptography.x509 import load_pem_x509_certificate, ocsp
>>> response = ocsp.OCSPResponseBuilder.build_unsuccessful(
...     ocsp.OCSPResponseStatus.UNAUTHORIZED
... )
>>> response.response_status
<OCSPResponseStatus.UNAUTHORIZED: 6>
```

Interfaces

class `cryptography.x509.ocsp.`**`OCSPRequest`**

Added in version 2.4.

An `OCSPRequest` is an object containing information about a certificate whose status is being checked.

`issuer_key_hash`

Type
bytes

The hash of the certificate issuer's key. The hash algorithm used is defined by the `hash_algorithm` property.

`issuer_name_hash`

Type
bytes

The hash of the certificate issuer's name. The hash algorithm used is defined by the `hash_algorithm` property.

`hash_algorithm`

Type
HashAlgorithm

The algorithm used to generate the `issuer_key_hash` and `issuer_name_hash`.

`serial_number`

Type
int

The serial number of the certificate to check.

`extensions`

Type
Extensions

The extensions encoded in the request.

`public_bytes`(*encoding*)

Parameters
encoding – The encoding to use. Only *DER* is supported.

Return bytes
The serialized OCSP request.

class cryptography.x509.ocsp.**OCSPResponse**

Added in version 2.4.

An `OCSPResponse` is the data provided by an OCSP responder in response to an `OCSPRequest`.

response_status

Type
OCSPResponseStatus

The status of the response.

signature_algorithm_oid

Type
ObjectIdentifier

Returns the object identifier of the signature algorithm used to sign the response. This will be one of the OIDs from *SignatureAlgorithmOID*.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

signature_hash_algorithm

Added in version 2.5.

Type
HashAlgorithm

Returns the *HashAlgorithm* which was used in signing this response. Can be `None` if signature did not use separate hash (*ED25519*, *ED448*).

signature

Type
bytes

The signature bytes.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

tbs_response_bytes

Type
bytes

The DER encoded bytes payload that is hashed and then signed. This data may be used to validate the signature on the OCSP response.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

certificates

Type
list

A list of zero or more *Certificate* objects used to help build a chain to verify the OCSP response. This situation occurs when the OCSP responder uses a delegate certificate.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

responder_key_hash

Type
bytes or None

The responder's key hash or `None` if the response has a `responder_name`.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

responder_name

Type
Name or None

The responder's `Name` or `None` if the response has a `responder_key_hash`.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

produced_at

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *produced_at_utc()*.

A naïve datetime representing the time when the response was produced.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

produced_at_utc

Added in version 43.0.0.

Type
`datetime.datetime`

A timezone-aware datetime representing the time when the response was produced.

Raises
ValueError – If `response_status` is not *SUCCESSFUL*.

certificate_status

Type
OCSPCertStatus

The status of the certificate being checked.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

revocation_time

Type
`datetime.datetime` or None

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *revocation_time_utc()*.

A naïve datetime representing the time when the certificate was revoked or `None` if the certificate has not been revoked.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

revocation_time_utc

Added in version 43.0.0.

Type
`datetime.datetime` or None

A timezone-aware datetime representing the time when the certificate was revoked or `None` if the certificate has not been revoked.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

revocation_reason

Type
ReasonFlags or None

The reason the certificate was revoked or `None` if not specified or not revoked.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

this_update

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *this_update_utc()*.

A naïve datetime representing the most recent time at which the status being indicated is known by the responder to have been correct.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

this_update_utc

Added in version 43.0.0.

Type
`datetime.datetime`

A timezone-aware datetime representing the most recent time at which the status being indicated is known by the responder to have been correct.

Raises
`ValueError` – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

next_update

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *next_update_utc()*.

A naïve datetime representing the time when newer information will be available.

Raises
`ValueError` – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

next_update_utc

Added in version 43.0.0.

Type
`datetime.datetime`

A timezone-aware datetime representing the time when newer information will be available.

Raises
`ValueError` – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

issuer_key_hash

Type
bytes

The hash of the certificate issuer's key. The hash algorithm used is defined by the `hash_algorithm` property.

Raises
`ValueError` – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

issuer_name_hash

Type
bytes

The hash of the certificate issuer's name. The hash algorithm used is defined by the `hash_algorithm` property.

Raises
`ValueError` – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

hash_algorithm

Type
HashAlgorithm

The algorithm used to generate the `issuer_key_hash` and `issuer_name_hash`.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

serial_number

Type
int

The serial number of the certificate that was checked.

Raises
ValueError – If `response_status` is not *SUCCESSFUL* or if multiple SINGLERESPs are present.

extensions

Type
Extensions

The extensions encoded in the response.

single_extensions

Added in version 2.9.

Type
Extensions

The single extensions encoded in the response.

responses

Added in version 37.0.0.

Type
an iterator over *OCSPSingleResponse*

An iterator to access individual SINGLERESP structures.

public_bytes(*encoding*)

Parameters
encoding – The encoding to use. Only *DER* is supported.

Return bytes
The serialized OCSP response.

class cryptography.x509.ocsp.**OCSPResponseStatus**

Added in version 2.4.

An enumeration of response statuses.

SUCCESSFUL

Represents a successful OCSP response.

MALFORMED_REQUEST

May be returned by an OCSP responder that is unable to parse a given request.

INTERNAL_ERROR

May be returned by an OCSP responder that is currently experiencing operational problems.

TRY_LATER

May be returned by an OCSP responder that is overloaded.

SIG_REQUIRED

May be returned by an OCSP responder that requires signed OCSP requests.

UNAUTHORIZED

May be returned by an OCSP responder when queried for a certificate for which the responder is unaware or an issuer for which the responder is not authoritative.

class `cryptography.x509.ocsp.`**OCSPCertStatus**

Added in version 2.4.

An enumeration of certificate statuses in an OCSP response.

GOOD

The value for a certificate that is not revoked.

REVOKED

The certificate being checked is revoked.

UNKNOWN

The certificate being checked is not known to the OCSP responder.

class `cryptography.x509.ocsp.`**OCSPResponderEncoding**

Added in version 2.4.

An enumeration of `responderID` encodings that can be passed to *`responder_id()`*.

HASH

Encode the hash of the public key whose corresponding private key signed the response.

NAME

Encode the X.509 `Name` of the certificate whose private key signed the response.

class `cryptography.x509.ocsp.`**OCSPSingleResponse**

Added in version 37.0.0.

A class representing a single certificate response bundled into a larger OCSPResponse. Accessed via OCSPResponse.responses.

certificate_status

Type
`OCSPCertStatus`

The status of the certificate being checked.

revocation_time

Type
`datetime.datetime` or None

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *revocation_time_utc()*.

A naïve datetime representing the time when the certificate was revoked or `None` if the certificate has not been revoked.

revocation_time_utc

Added in version 43.0.0.

Type
`datetime.datetime` or None

A timezone-aware datetime representing the time when the certificate was revoked or `None` if the certificate has not been revoked.

revocation_reason

Type
ReasonFlags or None

The reason the certificate was revoked or `None` if not specified or not revoked.

this_update

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *this_update_utc()*.

A naïve datetime representing the most recent time at which the status being indicated is known by the responder to have been correct.

this_update_utc

Added in version 43.0.0.

Type
`datetime.datetime`

A timezone-aware datetime representing the most recent time at which the status being indicated is known by the responder to have been correct.

next_update

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *next_update_utc()*.

A naïve datetime representing the time when newer information will be available.

next_update_utc

Added in version 43.0.0.

Type
datetime.datetime

A timezone-aware datetime representing the time when newer information will be available.

issuer_key_hash

Type
bytes

The hash of the certificate issuer's key. The hash algorithm used is defined by the `hash_algorithm` property.

issuer_name_hash

Type
bytes

The hash of the certificate issuer's name. The hash algorithm used is defined by the `hash_algorithm` property.

hash_algorithm

Type
HashAlgorithm

The algorithm used to generate the `issuer_key_hash` and `issuer_name_hash`.

serial_number

Type
int

The serial number of the certificate that was checked.

2.3.4 X.509 Verification

Support for X.509 certificate verification, also known as path validation or chain building.

Example usage, with certifi providing the root of trust:

```
>>> from cryptography.x509 import Certificate, DNSName, load_pem_x509_certificates
>>> from cryptography.x509.verification import PolicyBuilder, Store
>>> import certifi
>>> from datetime import datetime, timezone
>>> with open(certifi.where(), "rb") as pems:
...     store = Store(load_pem_x509_certificates(pems.read()))
>>> builder = PolicyBuilder().store(store)
>>> # See the documentation on `time` below for more details. If
>>> # significant time passes between creating a verifier and performing a
>>> # verification, you may encounter issues with certificate expiration.
>>> verification_time = datetime.now(timezone.utc)
>>> builder = builder.time(verification_time)
>>> verifier = builder.build_server_verifier(DNSName("cryptography.io"))
>>> # NOTE: peer and untrusted_intermediates are Certificate and
```

(continues on next page)

(continued from previous page)

```
>>> #       list[Certificate] respectively, and should be loaded from the
>>> #       application context that needs them verified, such as a
>>> #       TLS socket.
>>> chain = verifier.verify(peer, untrusted_intermediates)
```

class cryptography.x509.verification.**Store**(*certs*)

Added in version 42.0.0.

A Store is an opaque set of public keys and subject identifiers that are considered trusted *a priori*. Stores are typically created from the host OS's root of trust, from a well-known source such as a browser CA bundle, or from a small set of manually pre-trusted entities.

Parameters
certs – A list of one or more *cryptography.x509.Certificate* instances.

class cryptography.x509.verification.**Subject**

Added in version 42.0.0.

Type alias: A union of all subject types supported: *cryptography.x509.general_name.DNSName*, *cryptography.x509.general_name.IPAddress*.

class cryptography.x509.verification.**VerifiedClient**

Added in version 43.0.0.

Changed in version 45.0.0: Made `subjects` optional with the addition of custom extension policies.

subjects

Type
list of *GeneralName* or None

The subjects presented in the verified client's Subject Alternative Name extension or `None` if the extension is not present.

chain

Type
A list of *Certificate*, in leaf-first order

The chain of certificates that forms the valid chain to the client certificate.

class cryptography.x509.verification.**ClientVerifier**

Added in version 43.0.0.

A ClientVerifier verifies client certificates.

It contains and describes various pieces of configurable path validation logic, such as how deep prospective validation chains may go, which signature algorithms are allowed, and so forth.

ClientVerifier instances cannot be constructed directly; *PolicyBuilder* must be used.

policy

Type
Policy

The policy used by the verifier. Can be used to access verification time, maximum chain depth, etc.

store

Type
Store

The verifier's trust store.

verify(*leaf*, *intermediates*)

Performs path validation on `leaf`, returning a valid path if one exists. The path is returned in leaf-first order: the first member is `leaf`, followed by the intermediates used (if any), followed by a member of the `store`.

Parameters

- **leaf** – The leaf *Certificate* to validate
- **intermediates** – A `list` of intermediate *Certificate* to attempt to use

Returns
A new instance of *VerifiedClient*

Raises

- ***VerificationError*** – If a valid chain cannot be constructed
- ***UnsupportedGeneralNameType*** – If a valid chain exists, but contains an unsupported general name type

class cryptography.x509.verification.**ServerVerifier**

Added in version 42.0.0.

A ServerVerifier verifies server certificates.

It contains and describes various pieces of configurable path validation logic, such as which subject to expect, how deep prospective validation chains may go, which signature algorithms are allowed, and so forth.

ServerVerifier instances cannot be constructed directly; *PolicyBuilder* must be used.

policy

Type
Policy

The policy used by the verifier. Can be used to access verification time, maximum chain depth, etc.

store

Type
Store

The verifier's trust store.

verify(*leaf*, *intermediates*)

Performs path validation on `leaf`, returning a valid path if one exists. The path is returned in leaf-first order: the first member is `leaf`, followed by the intermediates used (if any), followed by a member of the `store`.

Parameters

- **leaf** – The leaf *Certificate* to validate
- **intermediates** – A `list` of intermediate *Certificate* to attempt to use

Returns
A list containing a valid chain from `leaf` to a member of *ServerVerifier.store*.

Raises
VerificationError – If a valid chain cannot be constructed

class cryptography.x509.verification.**VerificationError**

Added in version 42.0.0.

The error raised when path validation fails.

class cryptography.x509.verification.**PolicyBuilder**

Added in version 42.0.0.

Changed in version 45.0.0: Added the `extension_policies` method. Removed the `new_` prefix from all parameter names.

A PolicyBuilder provides a builder-style interface for constructing a Verifier.

time(*time*)

Sets the verifier's verification time.

If not called explicitly, this is set to `datetime.datetime.now()` when *build_server_verifier()* or *build_client_verifier()* is called.

Parameters
time – The `datetime.datetime` to use in the verifier

Returns
A new instance of *PolicyBuilder*

store(*store*)

Sets the verifier's trust store.

Parameters
store – The *Store* to use in the verifier

Returns
A new instance of *PolicyBuilder*

max_chain_depth(*max_chain_depth*)

Sets the verifier's maximum chain building depth.

This depth behaves tracks the length of the intermediate CA chain: a maximum depth of zero means that the leaf must be directly issued by a member of the store, a depth of one means no more than one intermediate CA, and so forth. Note that self-issued intermediates don't count against the chain depth, per RFC 5280.

Parameters
max_chain_depth – The maximum depth to allow in the verifier

Returns
A new instance of *PolicyBuilder*

extension_policies(** (Keyword-only parameters separator (PEP 3102))*, *ee_policy*, *ca_policy*)

Added in version 45.0.0.

Sets the EE and CA extension policies for the verifier. The default policies used are those returned by *ExtensionPolicy.webpki_defaults_ee()* and *ExtensionPolicy.webpki_defaults_ca()*.

> **Warning**
>
> If the PolicyBuilder will be used to build a *ServerVerifier*, the EE extension policy *must require* the *SubjectAlternativeName* extension to be present. All CA extension policies *must require* the *BasicConstraints* extension to be present.

Parameters

- **ca_policy** (ExtensionPolicy) – The CA extension policy to use.
- **ee_policy** (ExtensionPolicy) – The EE extension policy to use.

Returns
A new instance of *PolicyBuilder*

build_server_verifier(*subject*)

Builds a verifier for verifying server certificates.

Parameters
subject – A *Subject* to use in the verifier

Returns
An instance of *ServerVerifier*

build_client_verifier()

Added in version 43.0.0.

Builds a verifier for verifying client certificates.

> **Warning**
>
> This API is not suitable for website (i.e. server) certificate verification. You **must** use *build_server_verifier()* for server verification.

Returns
An instance of *ClientVerifier*

class cryptography.x509.verification.**ExtensionPolicy**

Added in version 45.0.0.

ExtensionPolicy provides a set of static methods to construct predefined extension policies, and a builder-style interface for modifying them.

> **Note**
>
> Calling any of the builder methods (*require_not_present()*, *may_be_present()*, or *require_present()*) multiple times with the same extension type will raise an exception.

Changed in version 49.0.0: Any extension type may now be used with the builder methods.

> **Note**
>
> If a present extension's value cannot be parsed into a known Python object, it is passed to the validator callback as an `UnrecognizedExtension`.

static permit_all()

Creates an ExtensionPolicy that does not put any constraints on a certificate's extensions. This can serve as a base for a fully custom extension policy.

Returns
An instance of `ExtensionPolicy`

static webpki_defaults_ca()

Creates an ExtensionPolicy for CA certificates, based on CA/B Forum guidelines.

This is the default CA extension policy used by `PolicyBuilder`.

Returns
An instance of `ExtensionPolicy`

static webpki_defaults_ee()

Creates an ExtensionPolicy for EE certificates, based on CA/B Forum guidelines.

This is the default EE extension policy used by `PolicyBuilder`.

Returns
An instance of `ExtensionPolicy`

require_not_present(*extension_type*)

Specifies that the extension identified by *extension_type* must not be present (must be absent).

Parameters
extension_type – The extension type of the extension that must not be present.

Returns
An instance of `ExtensionPolicy`

may_be_present(*extension_type*, *criticality*, *validator_cb*)

Specifies that the extension identified by *extension_type* is optional. If it is present, it must conform to the given criticality constraint. An optional validator callback may be provided.

If a validator callback is provided, the callback will be invoked when `ClientVerifier.verify()` or `ServerVerifier.verify()` is called on a verifier that uses the extension policy. For details on the callback signature, see `MaybeExtensionValidatorCallback`.

Parameters

- **extension_type** – A concrete class derived from `ExtensionType` indicating which extension may be present.
- **criticality** (`Criticality`) – The criticality of the extension
- **validator_cb** (`MaybeExtensionValidatorCallback` or None) – An optional Python callback to validate the extension value. Must accept extensions of type *extension_type*.

Returns
An instance of `ExtensionPolicy`

require_present(*extension_type*, *criticality*, *validator_cb*)

Specifies that the extension identified by *extension_type* ` must be present and conform to the given criticality constraint. An optional validator callback may be provided.

If a validator callback is provided, the callback will be invoked when `ClientVerifier.verify()` or `ServerVerifier.verify()` is called on a verifier that uses the extension policy. For details on the callback signature, see `PresentExtensionValidatorCallback`.

Parameters

- **extension_type** – A concrete class derived from `ExtensionType` indicating which extension is required to be present.
- **criticality** (`Criticality`) – The criticality of the extension
- **validator_cb** (`PresentExtensionValidatorCallback` or None) – An optional Python callback to validate the extension value. Must accept extensions of type *extension_type*.

Returns

An instance of `ExtensionPolicy`

class cryptography.x509.verification.**Criticality**

Added in version 45.0.0.

An enumeration of criticality constraints for certificate extensions.

CRITICAL

The extension must be marked as critical.

AGNOSTIC

The extension may be marked either as critical or non-critical.

NON_CRITICAL

The extension must not be marked as critical.

class cryptography.x509.verification.**Policy**

Added in version 45.0.0.

Represents a policy for certificate verification. Passed to extension validator callbacks and accessible via `ClientVerifier` and `ServerVerifier`.

max_chain_depth

The maximum chain depth (as described in `PolicyBuilder.max_chain_depth()`).

Type

int

subject

The subject used during verification. Will be None if the verifier is a `ClientVerifier`.

Type

x509.verification.Subject or None

validation_time

The validation time.

Type

datetime.datetime

extended_key_usage

The Extended Key Usage required by the policy.

Type
x509.ObjectIdentifier

minimum_rsa_modulus

The minimum RSA modulus size required by the policy.

Type
int

type cryptography.x509.verification.**MaybeExtensionValidatorCallback** = Callable[[*Policy*, *Certificate*, *ExtensionType* | None], None]

Added in version 45.0.0.

A Python callback that validates an extension that may or may not be present. If the extension is not present, the callback will be invoked with *ext* set to *None*.

To fail the validation, the callback must raise an exception.

Parameters

- **policy** (Policy) – The verification policy.
- **certificate** (Certificate) – The certificate being verified.
- **extension** (ExtensionType *or None*) – The extension value or *None* if the extension is not present.

Returns
An extension validator callback must return *None*. If the validation fails, the validator must raise an exception.

type cryptography.x509.verification.**PresentExtensionValidatorCallback** = Callable[[*Policy*, *Certificate*, *ExtensionType*], None]

Added in version 45.0.0.

A Python callback that validates an extension that must be present.

To fail the validation, the callback must raise an exception.

Parameters

- **policy** (Policy) – The verification policy.
- **certificate** (Certificate) – The certificate being verified.
- **extension** (ExtensionType) – The extension value.

Returns
An extension validator callback must return *None*. If the validation fails, the validator must raise an exception.

2.3.5 X.509 Reference

Loading Certificates

cryptography.x509.**load_pem_x509_certificate**(*data*)

Added in version 0.7.

Deserialize a certificate from PEM encoded data. PEM certificates are base64 decoded and have delimiters that look like `-----BEGIN CERTIFICATE-----`.

Parameters
data (*bytes*) – The PEM encoded certificate data.

Returns
An instance of *Certificate*.

```
>>> from cryptography import x509
>>> cert = x509.load_pem_x509_certificate(pem_data)
>>> cert.serial_number
2
```

`cryptography.x509.load_pem_x509_certificates(data)`

Added in version 39.0.0.

Deserialize one or more certificates from PEM encoded data.

This is like *load_pem_x509_certificate()*, but allows for loading multiple certificates (as adjacent PEMs) at once.

Parameters
data (*bytes*) – One or more PEM-encoded certificates.

Returns
list of *Certificate*

Raises
ValueError – If there isn't at least one certificate, or if any certificate is malformed.

`cryptography.x509.load_der_x509_certificate(data)`

Added in version 0.7.

Deserialize a certificate from DER encoded data. DER is a binary format and is commonly found in files with the `.cer` extension (although file extensions are not a guarantee of encoding type).

Parameters
data (*bytes*) – The DER encoded certificate data.

Returns
An instance of *Certificate*.

Raises
ValueError – If a certificate cannot be parsed from the provided data.

Loading Certificate Revocation Lists

`cryptography.x509.load_pem_x509_crl(data)`

Added in version 1.1.

Deserialize a certificate revocation list (CRL) from PEM encoded data. PEM requests are base64 decoded and have delimiters that look like `-----BEGIN X509 CRL-----`.

Parameters
data (*bytes*) – The PEM encoded request data.

Returns
An instance of *CertificateRevocationList*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> crl = x509.load_pem_x509_crl(pem_crl_data)
```

(continues on next page)

(continued from previous page)

```
>>> crl.signature_hash_algorithm == hashes.SHA256()
True
```

`cryptography.x509.load_der_x509_crl(data)`

Added in version 1.1.

Deserialize a certificate revocation list (CRL) from DER encoded data. DER is a binary format.

Parameters
data (*bytes*) – The DER encoded request data.

Returns
An instance of *CertificateRevocationList*.

Loading Certificate Signing Requests

`cryptography.x509.load_pem_x509_csr(data)`

Added in version 0.9.

Deserialize a certificate signing request (CSR) from PEM encoded data. PEM requests are base64 decoded and have delimiters that look like `-----BEGIN CERTIFICATE REQUEST-----`. This format is also known as PKCS#10.

Parameters
data (*bytes*) – The PEM encoded request data.

Returns
An instance of *CertificateSigningRequest*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> csr = x509.load_pem_x509_csr(pem_req_data)
>>> csr.signature_hash_algorithm == hashes.SHA256()
True
```

`cryptography.x509.load_der_x509_csr(data)`

Added in version 0.9.

Deserialize a certificate signing request (CSR) from DER encoded data. DER is a binary format and is not commonly used with CSRs.

Parameters
data (*bytes*) – The DER encoded request data.

Returns
An instance of *CertificateSigningRequest*.

X.509 Certificate Object

class `cryptography.x509.Certificate`

Added in version 0.7.

version

Type
Version

The certificate version as an enumeration. Version 3 certificates are the latest version and also the only type you should see in practice.

Raises
cryptography.x509.InvalidVersion – If the version in the certificate is not a known *X.509 version*.

```
>>> cert.version
<Version.v3: 2>
```

fingerprint(*algorithm*)

Parameters
algorithm – The *HashAlgorithm* that will be used to generate the fingerprint.

Return bytes
The fingerprint using the supplied hash algorithm, as bytes.

```
>>> from cryptography.hazmat.primitives import hashes
>>> cert.fingerprint(hashes.SHA256())
b'\x86\xd2\x187Gc\xfc\xe7}[+E9\x8d\xb4\x8f\x10\xe5S\xda\x18u\xbe}a\x03\x08[\xac\
↪xa04?'
```

serial_number

Type
int

The serial as a Python integer.

```
>>> cert.serial_number
2
```

public_key()

The public key associated with the certificate.

Returns
One of *CertificatePublicKeyTypes*.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> public_key = cert.public_key()
>>> isinstance(public_key, rsa.RSAPublicKey)
True
```

public_key_algorithm_oid

Added in version 43.0.0.

Type
ObjectIdentifier

Returns the *ObjectIdentifier* of the public key algorithm found inside the certificate. This will be one of the OIDs from *PublicKeyAlgorithmOID*.

```
>>> cert.public_key_algorithm_oid
<ObjectIdentifier(oid=1.2.840.113549.1.1.1, name=rsaEncryption)>
```

not_valid_before

Type
datetime.datetime

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *not_valid_before_utc()*.

A naïve datetime representing the beginning of the validity period for the certificate in UTC. This value is inclusive.

```
>>> cert.not_valid_before
datetime.datetime(2010, 1, 1, 8, 30)
```

not_valid_before_utc

Added in version 42.0.0.

Type
datetime.datetime

A timezone-aware datetime representing the beginning of the validity period for the certificate in UTC. This value is inclusive.

```
>>> cert.not_valid_before_utc
datetime.datetime(2010, 1, 1, 8, 30, tzinfo=datetime.timezone.utc)
```

not_valid_after

Type
datetime.datetime

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *not_valid_after_utc()*.

A naïve datetime representing the end of the validity period for the certificate in UTC. This value is inclusive.

```
>>> cert.not_valid_after
datetime.datetime(2030, 12, 31, 8, 30)
```

not_valid_after_utc

Added in version 42.0.0.

Type
datetime.datetime

A timezone-aware datetime representing the end of the validity period for the certificate in UTC. This value is inclusive.

```
>>> cert.not_valid_after_utc
datetime.datetime(2030, 12, 31, 8, 30, tzinfo=datetime.timezone.utc)
```

issuer

Added in version 0.8.

Type
Name

The *Name* of the issuer.

subject

Added in version 0.8.

Type
Name

The *Name* of the subject.

signature_hash_algorithm

Type
HashAlgorithm

Returns the *HashAlgorithm* which was used in signing this certificate. Can be `None` if signature did not use separate hash (*ED25519*, *ED448*).

```
>>> from cryptography.hazmat.primitives import hashes
>>> cert.signature_hash_algorithm == hashes.SHA256()
True
```

signature_algorithm_oid

Added in version 1.6.

Type
ObjectIdentifier

Returns the *ObjectIdentifier* of the signature algorithm used to sign the certificate. This will be one of the OIDs from *SignatureAlgorithmOID*.

```
>>> cert.signature_algorithm_oid
<ObjectIdentifier(oid=1.2.840.113549.1.1.11, name=sha256WithRSAEncryption)>
```

signature_algorithm_parameters

Added in version 41.0.0.

Returns the parameters of the signature algorithm used to sign the certificate. For RSA signatures it will return either a *PKCS1v15* or *PSS* object.

For ECDSA signatures it will return an *ECDSA*.

For EdDSA and DSA signatures it will return `None`.

These objects can be used to verify signatures on the certificate.

Returns
None, *PKCS1v15*, *PSS*, or *ECDSA*

```
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> pss_cert = x509.load_pem_x509_certificate(rsa_pss_pem_cert)
>>> isinstance(pss_cert.signature_algorithm_parameters, padding.PSS)
True
```

extensions

Type
Extensions

The extensions encoded in the certificate.

Raises

- *cryptography.x509.DuplicateExtension* – If more than one extension of the same type is found within the certificate.
- *cryptography.x509.UnsupportedGeneralNameType* – If an extension contains a general name that is not supported.

```
>>> for ext in cert.extensions:
...     print(ext)
<Extension(oid=<ObjectIdentifier(oid=2.5.29.35, name=authorityKeyIdentifier)>,␣
↪critical=False, value=<AuthorityKeyIdentifier(key_identifier=b'\xe4}_\xd1\\\
↪x95\x86\x08,\x05\xae\xbeu\xb6e\xa7\xd9]\xa8f', authority_cert_issuer=None,␣
↪authority_cert_serial_number=None)>)>
<Extension(oid=<ObjectIdentifier(oid=2.5.29.14, name=subjectKeyIdentifier)>,␣
↪critical=False, value=<SubjectKeyIdentifier(digest=b'X\x01\x84$\x1b\xbc+R\
↪x94J=\xa5\x10r\x14Q\xf5\xaf:\xc9')>)>
<Extension(oid=<ObjectIdentifier(oid=2.5.29.15, name=keyUsage)>, critical=True,␣
↪value=<KeyUsage(digital_signature=False, content_commitment=False, key_
↪encipherment=False, data_encipherment=False, key_agreement=False, key_cert_
↪sign=True, crl_sign=True, encipher_only=False, decipher_only=False)>)>
<Extension(oid=<ObjectIdentifier(oid=2.5.29.32, name=certificatePolicies)>,␣
↪critical=False, value=<CertificatePolicies([<PolicyInformation(policy_
↪identifier=<ObjectIdentifier(oid=2.16.840.1.101.3.2.1.48.1, name=Unknown OID)>
↪, policy_qualifiers=None)>])>)>
<Extension(oid=<ObjectIdentifier(oid=2.5.29.19, name=basicConstraints)>,␣
↪critical=True, value=<BasicConstraints(ca=True, path_length=None)>)>
```

`signature`

Added in version 1.2.

Type
bytes

The bytes of the certificate's signature.

`tbs_certificate_bytes`

Added in version 1.2.

type
bytes

The DER encoded bytes payload (as defined by **RFC 5280**) that is hashed and then signed by the private key of the certificate's issuer. This data may be used to validate a signature, but use extreme caution as certificate validation is a complex problem that involves much more than just signature checks.

To validate the signature on a certificate you can do the following. Note: This only verifies that the certificate was signed with the private key associated with the public key provided and does not perform any of the other checks needed for secure certificate validation. Additionally, this example will only work for RSA public keys with `PKCS1v15` signatures, and so it can't be used for general purpose signature verification.

```
>>> from cryptography.hazmat.primitives.serialization import load_pem_
↪public_key
```

(continues on next page)

(continued from previous page)

```
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> issuer_public_key = load_pem_public_key(pem_issuer_public_key)
>>> cert_to_check = x509.load_pem_x509_certificate(pem_data_to_check)
>>> issuer_public_key.verify(
...     cert_to_check.signature,
...     cert_to_check.tbs_certificate_bytes,
...     # Depends on the algorithm used to create the certificate
...     padding.PKCS1v15(),
...     cert_to_check.signature_hash_algorithm,
... )
```

An *InvalidSignature* exception will be raised if the signature fails to verify.

verify_directly_issued_by(*issuer*)

Added in version 40.0.0.

Parameters

issuer (*Certificate*) – The issuer certificate to check against.

> **Warning**
>
> This method verifies that the certificate issuer name matches the issuer subject name and that the certificate is signed by the issuer's private key. **No other validation is performed.** Callers are responsible for performing any additional validations required for their use case (e.g. checking the validity period, whether the signer is allowed to issue certificates, that the issuing certificate has a strong public key, etc).

Validates that the certificate is signed by the provided issuer and that the issuer's subject name matches the issuer name of the certificate.

Returns

None

Raises

- **ValueError** – If the issuer name on the certificate does not match the subject name of the issuer or the signature algorithm is unsupported.
- **TypeError** – If the issuer does not have a supported public key type.
- *cryptography.exceptions.InvalidSignature* – If the signature fails to verify.

tbs_precertificate_bytes

Added in version 38.0.0.

Type

bytes

Raises

ValueError – If the certificate doesn't have the expected Certificate Transparency extensions.

The DER encoded bytes payload (as defined by **RFC 6962**) that is hashed and then signed by the private key of the pre-certificate's issuer. This data may be used to validate a Signed Certificate Timestamp's signature, but use extreme caution as SCT validation is a complex problem that involves much more than just signature checks.

This method is primarily useful in the context of programs that interact with and verify the products of Certificate Transparency logs, as specified in **RFC 6962**. If you are not directly interacting with a Certificate Transparency log, this method unlikely to be what you want. To make unintentional misuse less likely, it raises a `ValueError` if the underlying certificate does not contain the expected Certificate Transparency extensions.

public_bytes(*encoding*)

Added in version 1.0.

Parameters
encoding – The *Encoding* that will be used to serialize the certificate.

Return bytes
The data that can be written to a file or sent over the network to be verified by clients.

X.509 CRL (Certificate Revocation List) Object

class cryptography.x509.**CertificateRevocationList**

Added in version 1.0.

A CertificateRevocationList is an object representing a list of revoked certificates. The object is iterable and will yield the RevokedCertificate objects stored in this CRL.

```
>>> len(crl)
1
>>> revoked_certificate = crl[0]
>>> type(revoked_certificate)
<class '...RevokedCertificate'>
>>> for r in crl:
...     print(r.serial_number)
0
```

fingerprint(*algorithm*)

Parameters
algorithm – The *HashAlgorithm* that will be used to generate the fingerprint.

Return bytes
The fingerprint using the supplied hash algorithm, as bytes.

```
>>> from cryptography.hazmat.primitives import hashes
>>> crl.fingerprint(hashes.SHA256())
b'\xe3\x1d\xb5P\x18\x9ed\x9f\x16O\x9dm\xc1>\x8c\xca\xb1\xc6x?T\x9f\xe9t_\x1d\
↪x8dF8V\xf78'
```

get_revoked_certificate_by_serial_number(*serial_number*)

Added in version 2.3.

Parameters
serial_number – The serial as a Python integer.

Returns
RevokedCertificate if the `serial_number` is present in the CRL or `None` if it is not.

signature_hash_algorithm

Type
HashAlgorithm

Returns the *HashAlgorithm* which was used in signing this CRL. Can be `None` if signature did not use separate hash (*ED25519*, *ED448*).

```
>>> from cryptography.hazmat.primitives import hashes
>>> crl.signature_hash_algorithm == hashes.SHA256()
True
```

signature_algorithm_oid

Added in version 1.6.

Type
ObjectIdentifier

Returns the *ObjectIdentifier* of the signature algorithm used to sign the CRL. This will be one of the OIDs from *SignatureAlgorithmOID*.

```
>>> crl.signature_algorithm_oid
<ObjectIdentifier(oid=1.2.840.113549.1.1.11, name=sha256WithRSAEncryption)>
```

signature_algorithm_parameters

Added in version 42.0.0.

Returns the parameters of the signature algorithm used to sign the certificate revocation list. For RSA signatures it will return either a *PKCS1v15* or *PSS* object.

For ECDSA signatures it will return an *ECDSA*.

For EdDSA and DSA signatures it will return `None`.

These objects can be used to verify the CRL signature.

Returns
None, *PKCS1v15*, *PSS*, or *ECDSA*

issuer

Type
Name

The *Name* of the issuer.

```
>>> crl.issuer
<Name(CN=cryptography.io,C=US)>
```

next_update

Type
`datetime.datetime`

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *next_update_utc()*.

A naïve datetime representing when the next update to this CRL is expected.

```
>>> crl.next_update
datetime.datetime(2016, 1, 1, 0, 0)
```

next_update_utc

Added in version 42.0.0.

Type
datetime.datetime

A timezone-aware datetime representing when the next update to this CRL is expected.

```
>>> crl.next_update_utc
datetime.datetime(2016, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)
```

last_update

Type
datetime.datetime

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *last_update_utc()*.

A naïve datetime representing when this CRL was last updated.

```
>>> crl.last_update
datetime.datetime(2015, 1, 1, 0, 0)
```

last_update_utc

Added in version 42.0.0.

Type
datetime.datetime

A timezone-aware datetime representing when this CRL was last updated.

```
>>> crl.last_update_utc
datetime.datetime(2015, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)
```

extensions

Type
Extensions

The extensions encoded in the CRL.

signature

Added in version 1.2.

Type
bytes

The bytes of the CRL's signature.

tbs_certlist_bytes

Added in version 1.2.

Type
bytes

The DER encoded bytes payload (as defined by **RFC 5280**) that is hashed and then signed by the private key of the CRL's issuer. This data may be used to validate a signature, but use extreme caution as CRL validation is a complex problem that involves much more than just signature checks.

public_bytes(*encoding*)

Added in version 1.2.

Parameters

encoding – The *Encoding* that will be used to serialize the certificate revocation list.

Return bytes

The data that can be written to a file or sent over the network and used as part of a certificate verification process.

is_signature_valid(*public_key*)

Added in version 2.1.

> **Warning**
>
> Checking the validity of the signature on the CRL is insufficient to know if the CRL should be trusted. More details are available in **RFC 5280**.

Returns True if the CRL signature is correct for given public key, False otherwise.

X.509 Certificate Builder

class cryptography.x509.**CertificateBuilder**

Added in version 1.0.

> **Note**
>
> All methods, except *sign()* and *create_unsigned()*, return a **new** CertificateBuilder instance with the corresponding updated value. They do not modify the existing builder in place.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.x509.oid import NameOID
>>> import datetime
>>> one_day = datetime.timedelta(1, 0, 0)
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> public_key = private_key.public_key()
>>> builder = x509.CertificateBuilder()
>>> builder = builder.subject_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'cryptography.io'),
... ]))
>>> builder = builder.issuer_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'cryptography.io'),
... ]))
>>> builder = builder.not_valid_before(datetime.datetime.today() - one_day)
```

(continues on next page)

(continued from previous page)

```
>>> builder = builder.not_valid_after(datetime.datetime.today() + (one_day * 30))
>>> builder = builder.serial_number(x509.random_serial_number())
>>> builder = builder.public_key(public_key)
>>> builder = builder.add_extension(
...     x509.SubjectAlternativeName(
...         [x509.DNSName('cryptography.io')]
...     ),
...     critical=False
... )
>>> builder = builder.add_extension(
...     x509.BasicConstraints(ca=False, path_length=None), critical=True,
... )
>>> certificate = builder.sign(
...     private_key=private_key, algorithm=hashes.SHA256(),
... )
>>> isinstance(certificate, x509.Certificate)
True
```

issuer_name(*name*)

Sets the issuer's distinguished name.

Parameters

name – The *Name* that describes the issuer (CA).

Returns

A new *CertificateBuilder* with the updated issuer name.

subject_name(*name*)

Sets the subject's distinguished name.

Parameters

name – The *Name* that describes the subject.

Returns

A new *CertificateBuilder* with the updated subject name.

public_key(*public_key*, *, *rsa_padding=None*)

Sets the subject's public key.

Changed in version 49.0.0: Added the `rsa_padding` keyword-only parameter.

Parameters

- **public_key** – The subject's public key. This can be one of *CertificatePublicKeyTypes*.
- **rsa_padding** – This keyword argument is only valid with *RSAPublicKey*. The caller can pass an uninstantiated *PSS* class or `None`. If `PSS` is passed the `id-RSASSA-PSS` OID (with no parameters) will be encoded into the public key, which marks it as usable for only PSS signatures. This does not affect the certificate's signature, only the public key encoding within the certificate.

Returns

A new *CertificateBuilder* with the updated public key.

serial_number(*serial_number*)

Sets the certificate's serial number (an integer). The CA's policy determines how it attributes serial numbers to certificates. This number must uniquely identify the certificate given the issuer. CABForum Guidelines

require entropy in the serial number to provide protection against hash collision attacks. For more information on secure random number generation, see *Random number generation*.

Parameters

serial_number – Integer number that will be used by the CA to identify this certificate (most notably during certificate revocation checking). Users should consider using `random_serial_number()` when possible.

Returns

A new `CertificateBuilder` with the updated serial number.

not_valid_before(*time*)

Sets the certificate's activation time. This is the time from which clients can start trusting the certificate. It may be different from the time at which the certificate was created.

Parameters

time – The `datetime.datetime` object (in UTC) that marks the activation time for the certificate. The certificate may not be trusted clients if it is used before this time.

Returns

A new `CertificateBuilder` with the updated activation time.

not_valid_after(*time*)

Sets the certificate's expiration time. This is the time from which clients should no longer trust the certificate. The CA's policy will determine how long the certificate should remain in use.

Parameters

time – The `datetime.datetime` object (in UTC) that marks the expiration time for the certificate. The certificate may not be trusted clients if it is used after this time.

Returns

A new `CertificateBuilder` with the updated expiration time.

add_extension(*extval*, *critical*)

Adds an X.509 extension to the certificate.

Parameters

- **extval** – An extension conforming to the `ExtensionType` interface.
- **critical** – Set to `True` if the extension must be understood and handled by whoever reads the certificate.

Returns

A new `CertificateBuilder` with the additional extension.

sign(*private_key*, *algorithm*, *, *rsa_padding=None*, *ecdsa_deterministic=None*)

Sign the certificate using the CA's private key.

Parameters

- **private_key** – The key that will be used to sign the certificate, one of `CertificateIssuerPrivateKeyTypes`.
- **algorithm** – The `HashAlgorithm` that will be used to generate the signature. This must be `None` if the `private_key` is an `Ed25519PrivateKey`, an `Ed448PrivateKey`, or an ML-DSA key (one of `MLDSA44PrivateKey`, `MLDSA65PrivateKey`, or `MLDSA87PrivateKey`), and an instance of a `HashAlgorithm` otherwise.
- **rsa_padding** (`None`, `PKCS1v15`, or `PSS`) – Added in version 41.0.0.

  This is a keyword-only argument. If `private_key` is an `RSAPrivateKey` then this can be set to either `PKCS1v15` or `PSS` to sign with those respective paddings. If this is `None` then

RSA keys will default to `PKCS1v15` padding. All other key types **must** not pass a value other than `None`.

- **ecdsa_deterministic** (`None`, `bool`) – Added in version 45.0.0.

  This is a keyword-only argument. If `private_key` is an `EllipticCurvePrivateKey` then this can be set to true to use deterministic signing, as defined in **RFC 6979**. This only impacts the signing process, verification is not affected (the verification process is the same for both deterministic and non-deterministic signed messages). All other key types **must** not pass a value other than `None`.

Returns
Certificate

create_unsigned()

Added in version 50.0.0.

Creates an unsigned certificate, per **RFC 9925**.

Returns
Certificate

X.509 CSR (Certificate Signing Request) Object

class cryptography.x509.**CertificateSigningRequest**

Added in version 0.9.

public_key()

The public key associated with the request.

Returns
One of *CertificatePublicKeyTypes*.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> public_key = csr.public_key()
>>> isinstance(public_key, rsa.RSAPublicKey)
True
```

public_key_algorithm_oid

Added in version 43.0.0.

Type
ObjectIdentifier

Returns the *ObjectIdentifier* of the public key algorithm found inside the certificate. This will be one of the OIDs from *PublicKeyAlgorithmOID*.

```
>>> csr.public_key_algorithm_oid
<ObjectIdentifier(oid=1.2.840.113549.1.1.1, name=rsaEncryption)>
```

subject

Type
Name

The *Name* of the subject.

signature_hash_algorithm

Type
HashAlgorithm

Returns the *HashAlgorithm* which was used in signing this request. Can be None if signature did not use separate hash (*ED25519*, *ED448*).

```
>>> from cryptography.hazmat.primitives import hashes
>>> csr.signature_hash_algorithm == hashes.SHA256()
True
```

signature_algorithm_oid

Added in version 1.6.

Type
ObjectIdentifier

Returns the *ObjectIdentifier* of the signature algorithm used to sign the request. This will be one of the OIDs from *SignatureAlgorithmOID*.

```
>>> csr.signature_algorithm_oid
<ObjectIdentifier(oid=1.2.840.113549.1.1.11, name=sha256WithRSAEncryption)>
```

signature_algorithm_parameters

Added in version 42.0.0.

Returns the parameters of the signature algorithm used to sign the certificate signing request. For RSA signatures it will return either a *PKCS1v15* or *PSS* object.

For ECDSA signatures it will return an *ECDSA*.

For EdDSA and DSA signatures it will return None.

These objects can be used to verify signatures on the signing request.

Returns
None, *PKCS1v15*, *PSS*, or *ECDSA*

extensions

Type
Extensions

The extensions encoded in the certificate signing request.

Raises

- *cryptography.x509.DuplicateExtension* – If more than one extension of the same type is found within the certificate signing request.
- *cryptography.x509.UnsupportedGeneralNameType* – If an extension contains a general name that is not supported.

attributes

Added in version 36.0.0.

Type
Attributes

The attributes encoded in the certificate signing request.

public_bytes(*encoding*)

Added in version 1.0.

Parameters

encoding – The *Encoding* that will be used to serialize the certificate request.

Return bytes

The data that can be written to a file or sent over the network to be signed by the certificate authority.

signature

Added in version 1.2.

Type

bytes

The bytes of the certificate signing request's signature.

tbs_certrequest_bytes

Added in version 1.2.

Type

bytes

The DER encoded bytes payload (as defined by **RFC 2986**) that is hashed and then signed by the private key (corresponding to the public key embedded in the CSR). This data may be used to validate the CSR signature.

is_signature_valid

Added in version 1.3.

Returns True if the CSR signature is correct, False otherwise.

X.509 Certificate Revocation List Builder

class cryptography.x509.**CertificateRevocationListBuilder**

Added in version 1.2.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.x509.oid import NameOID
>>> import datetime
>>> one_day = datetime.timedelta(1, 0, 0)
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> builder = x509.CertificateRevocationListBuilder()
>>> builder = builder.issuer_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'cryptography.io CA'),
... ]))
>>> builder = builder.last_update(datetime.datetime.today())
>>> builder = builder.next_update(datetime.datetime.today() + one_day)
>>> revoked_cert = x509.RevokedCertificateBuilder().serial_number(
...     333
... ).revocation_date(
```

(continues on next page)

(continued from previous page)

```
...     datetime.datetime.today()
... ).build()
>>> builder = builder.add_revoked_certificate(revoked_cert)
>>> crl = builder.sign(
...     private_key=private_key, algorithm=hashes.SHA256(),
... )
>>> len(crl)
1
```

issuer_name(*name*)

Sets the issuer's distinguished name.

Parameters

name – The *Name* that describes the issuer (CA).

last_update(*time*)

Sets this CRL's activation time. This is the time from which clients can start trusting this CRL. It may be different from the time at which this CRL was created. This is also known as the `thisUpdate` time.

Parameters

time – The `datetime.datetime` object (in UTC) that marks the activation time for this CRL. The CRL may not be trusted if it is used before this time.

next_update(*time*)

Sets this CRL's next update time. This is the time by which a new CRL will be issued. The CA is allowed to issue a new CRL before this date, however clients are not required to check for it.

Parameters

time – The `datetime.datetime` object (in UTC) that marks the next update time for this CRL.

add_extension(*extval*, *critical*)

Adds an X.509 extension to this CRL.

Parameters

- **extval** – An extension with the *ExtensionType* interface.
- **critical** – Set to `True` if the extension must be understood and handled by whoever reads the CRL.

add_revoked_certificate(*revoked_certificate*)

Adds a revoked certificate to this CRL.

Parameters

revoked_certificate – An instance of *RevokedCertificate*. These can be obtained from an existing CRL or created with *RevokedCertificateBuilder*.

sign(*private_key*, *algorithm*, *, *rsa_padding=None*, *ecdsa_deterministic=None*)

Sign this CRL using the CA's private key.

Parameters

- **private_key** – The private key that will be used to sign the certificate, one of *CertificateIssuerPrivateKeyTypes*.
- **algorithm** – The *HashAlgorithm* that will be used to generate the signature. This must be `None` if the `private_key` is an *Ed25519PrivateKey*, an *Ed448PrivateKey*,

or an ML-DSA key (one of *MLDSA44PrivateKey*, *MLDSA65PrivateKey*, or *MLDSA87PrivateKey*), and an instance of a *HashAlgorithm* otherwise.

- **rsa_padding** (None, *PKCS1v15*, or *PSS*) – Added in version 42.0.0.

  This is a keyword-only argument. If `private_key` is an `RSAPrivateKey` then this can be set to either *PKCS1v15* or *PSS* to sign with those respective paddings. If this is `None` then RSA keys will default to `PKCS1v15` padding. All other key types **must** not pass a value other than `None`.

- **ecdsa_deterministic** (None, bool) – Added in version 45.0.0.

  This is a keyword-only argument. If `private_key` is an `EllipticCurvePrivateKey` then this can be set to true to use deterministic signing, as defined in **RFC 6979**. This only impacts the signing process, verification is not affected (the verification process is the same for both deterministic and non-deterministic signed messages). All other key types **must** not pass a value other than `None`.

Returns
CertificateRevocationList

X.509 Revoked Certificate Object

class cryptography.x509.**RevokedCertificate**

Added in version 1.0.

serial_number

Type
int

An integer representing the serial number of the revoked certificate.

```
>>> revoked_certificate.serial_number
0
```

revocation_date

Type
datetime.datetime

> **Warning**
>
> This property is deprecated and will be removed in a future version. Please switch to the timezone-aware variant *revocation_date_utc()*.

A naïve datetime representing the date this certificates was revoked.

```
>>> revoked_certificate.revocation_date
datetime.datetime(2015, 1, 1, 0, 0)
```

revocation_date_utc

Added in version 42.0.0.

Type
datetime.datetime

A timezone-aware datetime representing the date this certificates was revoked.

```
>>> revoked_certificate.revocation_date_utc
datetime.datetime(2015, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)
```

extensions

Type
Extensions

The extensions encoded in the revoked certificate.

```
>>> for ext in revoked_certificate.extensions:
...     print(ext)
<Extension(oid=<ObjectIdentifier(oid=2.5.29.24, name=invalidityDate)>,␣
↪critical=False, value=<InvalidityDate(invalidity_date=2015-01-01 00:00:00)>)>
<Extension(oid=<ObjectIdentifier(oid=2.5.29.21, name=cRLReason)>,␣
↪critical=False, value=<CRLReason(reason=ReasonFlags.key_compromise)>)>
```

X.509 Revoked Certificate Builder

class cryptography.x509.**RevokedCertificateBuilder**

This class is used to create *RevokedCertificate* objects that can be used with the *CertificateRevocationListBuilder*.

Added in version 1.2.

```
>>> from cryptography import x509
>>> import datetime
>>> builder = x509.RevokedCertificateBuilder()
>>> builder = builder.revocation_date(datetime.datetime.today())
>>> builder = builder.serial_number(3333)
>>> revoked_certificate = builder.build()
>>> isinstance(revoked_certificate, x509.RevokedCertificate)
True
```

serial_number(*serial_number*)

Sets the revoked certificate's serial number.

Parameters
serial_number – Integer number that is used to identify the revoked certificate.

revocation_date(*time*)

Sets the certificate's revocation date.

Parameters
time – The `datetime.datetime` object (in UTC) that marks the revocation time for the certificate.

add_extension(*extval*, *critical*)

Adds an X.509 extension to this revoked certificate.

Parameters

- **extval** – An instance of one of the *CRL entry extensions*.
- **critical** – Set to `True` if the extension must be understood and handled.

build()

Create a revoked certificate object.

Returns
RevokedCertificate

X.509 CSR (Certificate Signing Request) Builder Object

class cryptography.x509.**CertificateSigningRequestBuilder**

Added in version 1.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.x509.oid import AttributeOID, NameOID
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> builder = x509.CertificateSigningRequestBuilder()
>>> builder = builder.subject_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'cryptography.io'),
... ]))
>>> builder = builder.add_extension(
...     x509.BasicConstraints(ca=False, path_length=None), critical=True,
... )
>>> builder = builder.add_attribute(
...     AttributeOID.CHALLENGE_PASSWORD, b"changeit"
... )
>>> request = builder.sign(
...     private_key, hashes.SHA256()
... )
>>> isinstance(request, x509.CertificateSigningRequest)
True
```

subject_name(*name*)

Parameters
name – The *Name* of the certificate subject.

Returns
A new *CertificateSigningRequestBuilder*.

add_extension(*extension*, *critical*)

Parameters

- **extension** – An extension conforming to the *ExtensionType* interface.
- **critical** – Set to *True* if the extension must be understood and handled by whoever reads the certificate.

Returns
A new *CertificateSigningRequestBuilder*.

add_attribute(*oid*, *value*)

Added in version 3.0.

Parameters

- **oid** – An *ObjectIdentifier* instance.
- **value** (*bytes*) – The value of the attribute.

Returns
A new *CertificateSigningRequestBuilder*.

sign(*private_key*, *algorithm*, *, *rsa_padding=None*, *ecdsa_deterministic=None*)

Parameters

- **private_key** – The private key that will be used to sign the request. When the request is signed by a certificate authority, the private key's associated public key will be stored in the resulting certificate. One of *CertificateIssuerPrivateKeyTypes*.
- **algorithm** – The *HashAlgorithm* that will be used to generate the request signature. This must be `None` if the `private_key` is an *Ed25519PrivateKey*, an *Ed448PrivateKey*, or an ML-DSA key (one of *MLDSA44PrivateKey*, *MLDSA65PrivateKey*, or *MLDSA87PrivateKey*), and an instance of a *HashAlgorithm* otherwise.
- **rsa_padding** (`None`, *PKCS1v15*, or *PSS*) – Added in version 42.0.0.

  This is a keyword-only argument. If `private_key` is an `RSAPrivateKey` then this can be set to either *PKCS1v15* or *PSS* to sign with those respective paddings. If this is `None` then RSA keys will default to `PKCS1v15` padding. All other key types **must** not pass a value other than `None`.
- **ecdsa_deterministic** (`None`, `bool`) – Added in version 45.0.0.

  This is a keyword-only argument. If `private_key` is an `EllipticCurvePrivateKey` then this can be set to true to use deterministic signing, as defined in **RFC 6979**. This only impacts the signing process, verification is not affected (the verification process is the same for both deterministic and non-deterministic signed messages). All other key types **must** not pass a value other than `None`.

Returns
A new *CertificateSigningRequest*.

class cryptography.x509.**Name**

Added in version 0.8.

An X509 Name is an ordered list of attributes. The object is iterable to get every attribute or you can use *Name.get_attributes_for_oid()* to obtain the specific type you want. Names are sometimes represented as a slash or comma delimited string (e.g. `/CN=mydomain.com/O=My Org/C=US` or `CN=mydomain.com,O=My Org,C=US`).

Technically, a Name is a list of *sets* of attributes, called *Relative Distinguished Names* or *RDNs*, although multi-valued RDNs are rarely encountered. The iteration order of values within a multi-valued RDN is preserved. If you need to handle multi-valued RDNs, the `rdns` property gives access to an ordered list of *RelativeDistinguishedName* objects.

A Name can be initialized with an iterable of *NameAttribute* (the common case where each RDN has a single attribute) or an iterable of *RelativeDistinguishedName* objects (in the rare case of multi-valued RDNs).

```
>>> len(cert.subject)
3
>>> for attribute in cert.subject:
...     print(attribute)
<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.6, name=countryName)>, value='US')>
```

(continues on next page)

(continued from previous page)

```
<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.10, name=organizationName)>, value=
↪'Test Certificates 2011')>
<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, value='Good CA
↪')>
```

rdns

Added in version 1.6.

Type
list of *RelativeDistinguishedName*

classmethod from_bytes(*data*)

Added in version 49.0.0.

Parse a DER encoded name (the inverse of *public_bytes()*).

Parameters
data (*bytes*) – The DER encoded name.

Returns
A *Name* parsed from data.

```
>>> x509.Name.from_bytes(
...     b"0\x1a1\x180\x16\x06\x03U\x04\x03\x0c\x0fcryptography.io"
... )
<Name(CN=cryptography.io)>
```

classmethod from_rfc4514_string(*data*, *attr_name_overrides=None*)

Parameters

- **data** (*str*) – An **RFC 4514** string.
- **attr_name_overrides** – Specify custom OID to name mappings, which can be used to match vendor-specific extensions. See *NameOID* for common attribute OIDs.

Returns
A *Name* parsed from data.

```
>>> x509.Name.from_rfc4514_string("CN=cryptography.io")
<Name(CN=cryptography.io)>
>>> x509.Name.from_rfc4514_string("E=pyca@cryptography.io", {"E": NameOID.EMAIL_
↪ADDRESS})
<Name(1.2.840.113549.1.9.1=pyca@cryptography.io)>
```

get_attributes_for_oid(*oid*)

Parameters
oid – An *ObjectIdentifier* instance.

Returns
A list of *NameAttribute* instances that match the OID provided. If nothing matches an empty list will be returned.

```
>>> cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)
[<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, value=
↪'Good CA')>]
```

public_bytes()

Added in version 1.6.

Return bytes
The DER encoded name.

rfc4514_string(*attr_name_overrides=None*)

Added in version 2.5.

Changed in version 36.0.0: Added `attr_name_overrides` parameter.

Format the given name as a **RFC 4514** Distinguished Name string, for example `CN=mydomain.com,O=My Org,C=US`.

By default, attributes `CN`, `L`, `ST`, `O`, `OU`, `C`, `STREET`, `DC`, `UID` are represented by their short name. Unrecognized attributes are formatted as dotted OID strings.

Example:

```
>>> name = x509.Name([
...     x509.NameAttribute(NameOID.EMAIL_ADDRESS, "santa@north.pole"),
...     x509.NameAttribute(NameOID.COMMON_NAME, "Santa Claus"),
... ])
>>> name.rfc4514_string()
'CN=Santa Claus,1.2.840.113549.1.9.1=santa@north.pole'
>>> name.rfc4514_string({NameOID.EMAIL_ADDRESS: "E"})
'CN=Santa Claus,E=santa@north.pole'
```

Parameters
attr_name_overrides (Dict-like mapping from *ObjectIdentifier* to `str`) – Specify custom OID to name mappings, which can be used to match vendor-specific extensions. See *NameOID* for common attribute OIDs.

Return type
str

class cryptography.x509.**Version**

Added in version 0.7.

An enumeration for X.509 versions.

v1

For version 1 X.509 certificates.

v3

For version 3 X.509 certificates.

class cryptography.x509.**NameAttribute**

Added in version 0.8.

An X.509 name consists of a list of *RelativeDistinguishedName* instances, which consist of a set of *NameAttribute* instances.

oid

Type
ObjectIdentifier

The attribute OID.

value

Type
str or bytes

The value of the attribute. This will generally be a `str`, the only times it can be a `bytes` is when *oid* is `X500_UNIQUE_IDENTIFIER`.

rfc4514_attribute_name

Added in version 35.0.0.

Type
str

The **RFC 4514** short attribute name (for example "CN"), or the OID dotted string if a short name is unavailable.

rfc4514_string(*attr_name_overrides=None*)

Added in version 2.5.

Changed in version 36.0.0: Added `attr_name_overrides` parameter.

Return str
Format the given attribute as a **RFC 4514** Distinguished Name string.

Parameters
attr_name_overrides (Dict-like mapping from *ObjectIdentifier* to `str`) – Specify custom OID to name mappings, which can be used to match vendor-specific extensions.

class cryptography.x509.**RelativeDistinguishedName**(*attributes*)

Added in version 1.6.

A relative distinguished name is a non-empty set of name attributes. The object is iterable to get every attribute, preserving the original order. Passing duplicate attributes to the constructor raises `ValueError`.

get_attributes_for_oid(*oid*)

Parameters
oid – An *ObjectIdentifier* instance.

Returns
A list of *NameAttribute* instances that match the OID provided. The list should contain zero or one values.

rfc4514_string(*attr_name_overrides=None*)

Added in version 2.5.

Changed in version 36.0.0: Added `attr_name_overrides` parameter.

Return str
Format the given RDN set as a **RFC 4514** Distinguished Name string.

Parameters
attr_name_overrides (Dict-like mapping from *ObjectIdentifier* to `str`) – Specify custom OID to name mappings, which can be used to match vendor-specific extensions.

class cryptography.x509.**ObjectIdentifier**

Added in version 0.8.

Object identifiers (frequently seen abbreviated as OID) identify the type of a value (see: *NameAttribute*).

dotted_string

Type
str

The dotted string value of the OID (e.g. `"2.5.4.3"`)

General Name Classes

class cryptography.x509.**GeneralName**

Added in version 0.9.

This is the generic interface that all the following classes are registered against.

class cryptography.x509.**RFC822Name**(*value*)

Added in version 0.9.

Changed in version 3.1: *U-label* support has been removed. Encode them to *A-label* before use.

This corresponds to an email address. For example, `user@example.com`.

Parameters
value – The email address. If the address contains an internationalized domain name then it must be encoded to an *A-label* string before being passed.

Raises
ValueError – If the provided string is not an *A-label*.

value

Type
str

class cryptography.x509.**DNSName**(*value*)

Added in version 0.9.

Changed in version 3.1: *U-label* support has been removed. Encode them to *A-label* before use.

This corresponds to a domain name. For example, `cryptography.io`.

Parameters
value – The domain name. If it is an internationalized domain name then it must be encoded to an *A-label* string before being passed.

Raises
ValueError – If the provided string is not an *A-label*.

type
str

value

Type
str

class cryptography.x509.**DirectoryName**(*value*)

Added in version 0.9.

This corresponds to a directory name.

value

Type
Name

class cryptography.x509.**UniformResourceIdentifier**(*value*)

Added in version 0.9.

Changed in version 3.1: *U-label* support has been removed. Encode them to *A-label* before use.

This corresponds to a uniform resource identifier. For example, `https://cryptography.io`.

Parameters
value – The URI. If it contains an internationalized domain name then it must be encoded to an *A-label* string before being passed.

Raises
ValueError – If the provided string is not an *A-label*.

value

Type
str

class cryptography.x509.**IPAddress**(*value*)

Added in version 0.9.

This corresponds to an IP address.

value

Type
`IPv4Address`, `IPv6Address`, `IPv4Network`, or `IPv6Network`.

class cryptography.x509.**RegisteredID**(*value*)

Added in version 0.9.

This corresponds to a registered ID.

value

Type
ObjectIdentifier

class cryptography.x509.**OtherName**(*type_id*, *value*)

Added in version 1.0.

This corresponds to an `otherName`. An `otherName` has a type identifier and a value represented in binary DER format.

type_id

Type
ObjectIdentifier

value

Type
bytes

X.509 Extensions

class cryptography.x509.**Extensions**

Added in version 0.9.

An X.509 Extensions instance is an ordered list of extensions. The object is iterable to get every extension.

get_extension_for_oid(*oid*)

> **Note**
>
> For type-checking purposes, this method returns `Extension[ExtensionType]`, which is a generic type. To access methods specific to a particular extension class, you'll need to add a type assertion or a cast. When you know the specific extension type you need, prefer *get_extension_for_class()* which returns a properly typed `Extension[ExtensionTypeVar]` and allows direct access to extension-specific methods.

Parameters
oid – An *ObjectIdentifier* instance.

Returns
An instance of *Extension*.

Raises
cryptography.x509.ExtensionNotFound – If the certificate does not have the extension requested.

```
>>> from cryptography.x509.oid import ExtensionOID
>>> cert.extensions.get_extension_for_oid(ExtensionOID.BASIC_CONSTRAINTS)
<Extension(oid=<ObjectIdentifier(oid=2.5.29.19, name=basicConstraints)>,␣
↪critical=True, value=<BasicConstraints(ca=True, path_length=None)>)>
```

get_extension_for_class(*extclass*)

Added in version 1.1.

Parameters
extclass – An extension class.

Returns
An instance of *Extension*.

Raises
cryptography.x509.ExtensionNotFound – If the certificate does not have the extension requested.

```
>>> from cryptography import x509
>>> cert.extensions.get_extension_for_class(x509.BasicConstraints)
<Extension(oid=<ObjectIdentifier(oid=2.5.29.19, name=basicConstraints)>,␣
↪critical=True, value=<BasicConstraints(ca=True, path_length=None)>)>
```

class cryptography.x509.**Extension**

Added in version 0.9.

oid

Type
ObjectIdentifier

One of the *ExtensionOID* OIDs.

critical

Type
bool

Determines whether a given extension is critical or not. **RFC 5280** requires that "A certificate-using system MUST reject the certificate if it encounters a critical extension it does not recognize or a critical extension that contains information that it cannot process".

value

Returns an instance of the extension type corresponding to the OID.

class cryptography.x509.**ExtensionType**

Added in version 1.0.

This is the interface against which all the following extension types are registered.

oid

Type
ObjectIdentifier

Returns the OID associated with the given extension type.

public_bytes()

Added in version 36.0.0.

Return bytes
A bytes string representing the extension's DER encoded value.

class cryptography.x509.**KeyUsage**(*digital_signature*, *content_commitment*, *key_encipherment*, *data_encipherment*, *key_agreement*, *key_cert_sign*, *crl_sign*, *encipher_only*, *decipher_only*)

Added in version 0.9.

The key usage extension defines the purpose of the key contained in the certificate. The usage restriction might be employed when a key that could be used for more than one operation is to be restricted.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *KEY_USAGE*.

digital_signature

Type
bool

This purpose is set to true when the subject public key is used for verifying digital signatures, other than signatures on certificates (`key_cert_sign`) and CRLs (`crl_sign`).

content_commitment

Type
bool

This purpose is set to true when the subject public key is used for verifying digital signatures, other than signatures on certificates (`key_cert_sign`) and CRLs (`crl_sign`). It is used to provide a non-repudiation service that protects against the signing entity falsely denying some action. In the case of later conflict, a reliable third party may determine the authenticity of the signed data. This was called `non_repudiation` in older revisions of the X.509 specification.

key_encipherment

Type
bool

This purpose is set to true when the subject public key is used for enciphering private or secret keys.

data_encipherment

Type
bool

This purpose is set to true when the subject public key is used for directly enciphering raw user data without the use of an intermediate symmetric cipher.

key_agreement

Type
bool

This purpose is set to true when the subject public key is used for key agreement. For example, when a Diffie-Hellman key is to be used for key management, then this purpose is set to true.

key_cert_sign

Type
bool

This purpose is set to true when the subject public key is used for verifying signatures on public key certificates. If this purpose is set to true then `ca` must be true in the *`BasicConstraints`* extension.

crl_sign

Type
bool

This purpose is set to true when the subject public key is used for verifying signatures on certificate revocation lists.

encipher_only

Type
bool

When this purposes is set to true and the `key_agreement` purpose is also set, the subject public key may be used only for enciphering data while performing key agreement.

Raises
`ValueError` – This is raised if accessed when `key_agreement` is false.

decipher_only

Type
bool

When this purposes is set to true and the `key_agreement` purpose is also set, the subject public key may be used only for deciphering data while performing key agreement.

Raises
ValueError – This is raised if accessed when `key_agreement` is false.

class `cryptography.x509.`**`BasicConstraints`**(*ca*, *path_length*)

Added in version 0.9.

Basic constraints is an X.509 extension type that defines whether a given certificate is allowed to sign additional certificates and what path length restrictions may exist.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *BASIC_CONSTRAINTS*.

ca

Type
bool

Whether the certificate can sign certificates.

path_length

Type
int or None

The maximum path length for certificates subordinate to this certificate. This attribute only has meaning if `ca` is true. If `ca` is true then a path length of None means there's no restriction on the number of subordinate CAs in the certificate chain. If it is zero or greater then it defines the maximum length for a subordinate CA's certificate chain. For example, a `path_length` of 1 means the certificate can sign a subordinate CA, but the subordinate CA is not allowed to create subordinates with `ca` set to true.

class `cryptography.x509.`**`ExtendedKeyUsage`**(*usages*)

Added in version 0.9.

This extension indicates one or more purposes for which the certified public key may be used, in addition to or in place of the basic purposes indicated in the key usage extension. The object is iterable to obtain the list of *ExtendedKeyUsageOID* OIDs present.

Parameters
usages (*list*) – A list of *ExtendedKeyUsageOID* OIDs.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *EXTENDED_KEY_USAGE*.

class `cryptography.x509.`**`OCSPNoCheck`**

Added in version 1.0.

This presence of this extension indicates that an OCSP client can trust a responder for the lifetime of the responder's certificate. CAs issuing such a certificate should realize that a compromise of the responder's key is as serious as the compromise of a CA key used to sign CRLs, at least for the validity period of this certificate. CA's may choose to issue this type of certificate with a very short lifetime and renew it frequently. This extension is only relevant when the certificate is an authorized OCSP responder.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *OCSP_NO_CHECK*.

class cryptography.x509.**TLSFeature**(*features*)

Added in version 2.1.

The TLS Feature extension is defined in **RFC 7633** and is used in certificates for OCSP Must-Staple. The object is iterable to get every element.

Parameters
features (*list*) – A list of features to enable from the *TLSFeatureType* enum. At this time only `status_request` or `status_request_v2` are allowed.

oid

Type
ObjectIdentifier

Returns *TLS_FEATURE*.

class cryptography.x509.**TLSFeatureType**

Added in version 2.1.

An enumeration of TLS Feature types.

status_request

This feature type is defined in **RFC 6066** and, when embedded in an X.509 certificate, signals to the client that it should require a stapled OCSP response in the TLS handshake. Commonly known as OCSP Must-Staple in certificates.

status_request_v2

This feature type is defined in **RFC 6961**. This value is not commonly used and if you want to enable OCSP Must-Staple you should use `status_request`.

class cryptography.x509.**NameConstraints**(*permitted_subtrees*, *excluded_subtrees*)

Added in version 1.0.

The name constraints extension, which only has meaning in a CA certificate, defines a name space within which all subject names in certificates issued beneath the CA certificate must (or must not) be in. For specific details on the way this extension should be processed see **RFC 5280**.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *NAME_CONSTRAINTS*.

permitted_subtrees

Type
list of *GeneralName* objects or None

The set of permitted name patterns. If a name matches this and an element in `excluded_subtrees` it is invalid. At least one of `permitted_subtrees` and `excluded_subtrees` will be non-None.

excluded_subtrees

Type
list of *GeneralName* objects or None

Any name matching a restriction in the `excluded_subtrees` field is invalid regardless of information appearing in the `permitted_subtrees`. At least one of `permitted_subtrees` and `excluded_subtrees` will be non-None.

class cryptography.x509.**AuthorityKeyIdentifier**(*key_identifier*, *authority_cert_issuer*, *authority_cert_serial_number*)

Added in version 0.9.

The authority key identifier extension provides a means of identifying the public key corresponding to the private key used to sign a certificate. This extension is typically used to assist in determining the appropriate certificate chain. For more information about generation and use of this extension see **RFC 5280 Section 4.2.1.1**.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *AUTHORITY_KEY_IDENTIFIER*.

key_identifier

Type
bytes

A value derived from the public key used to verify the certificate's signature.

authority_cert_issuer

Type
A list of *GeneralName* instances or None

The *GeneralName* (one or multiple) of the issuer's issuer.

authority_cert_serial_number

Type
int or None

The serial number of the issuer's issuer.

classmethod from_issuer_public_key(*public_key*)

Added in version 1.0.

> **Note**
>
> This method should be used if the issuer certificate does not contain a *SubjectKeyIdentifier*. Otherwise, use *from_issuer_subject_key_identifier()*.

Creates a new AuthorityKeyIdentifier instance using the public key provided to generate the appropriate digest. This should be the **issuer's public key**. The resulting object will contain *key_identifier*, but *authority_cert_issuer* and *authority_cert_serial_number* will be None. The generated `key_identifier` is the SHA1 hash of the `subjectPublicKey` ASN.1 bit string. This is the first recommendation in **RFC 5280** section 4.2.1.2.

Parameters
public_key – One of *CertificateIssuerPublicKeyTypes*.

```
>>> from cryptography import x509
>>> issuer_cert = x509.load_pem_x509_certificate(pem_data)
>>> x509.AuthorityKeyIdentifier.from_issuer_public_key(issuer_cert.public_key())
<AuthorityKeyIdentifier(key_identifier=b'X\x01\x84$\x1b\xbc+R\x94J=\xa5\x10r\
↪x14Q\xf5\xaf:\xc9', authority_cert_issuer=None, authority_cert_serial_
↪number=None)>
```

classmethod from_issuer_subject_key_identifier(*ski*)

Added in version 1.3.

> **Note**
>
> This method should be used if the issuer certificate contains a *SubjectKeyIdentifier*. Otherwise, use *from_issuer_public_key()*.

Creates a new AuthorityKeyIdentifier instance using the SubjectKeyIdentifier from the issuer certificate. The resulting object will contain *key_identifier*, but *authority_cert_issuer* and *authority_cert_serial_number* will be None.

Parameters
ski – The *SubjectKeyIdentifier* from the issuer certificate.

```
>>> from cryptography import x509
>>> issuer_cert = x509.load_pem_x509_certificate(pem_data)
>>> ski_ext = issuer_cert.extensions.get_extension_for_class(x509.
↪SubjectKeyIdentifier)
>>> x509.AuthorityKeyIdentifier.from_issuer_subject_key_identifier(ski_ext.
↪value)
<AuthorityKeyIdentifier(key_identifier=b'X\x01\x84$\x1b\xbc+R\x94J=\xa5\x10r\
↪x14Q\xf5\xaf:\xc9', authority_cert_issuer=None, authority_cert_serial_
↪number=None)>
```

class cryptography.x509.**SubjectKeyIdentifier**(*digest*)

Added in version 0.9.

The subject key identifier extension provides a means of identifying certificates that contain a particular public key.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *SUBJECT_KEY_IDENTIFIER*.

key_identifier

Added in version 35.0.0.

Type
bytes

The binary value of the identifier.

digest

Type
bytes

The binary value of the identifier. An alias of `key_identifier`.

classmethod from_public_key(*public_key*)

Added in version 1.0.

Creates a new SubjectKeyIdentifier instance using the public key provided to generate the appropriate digest. This should be the public key that is in the certificate. The generated digest is the SHA1 hash of the `subjectPublicKey` ASN.1 bit string. This is the first recommendation in **RFC 5280** section 4.2.1.2.

Parameters
public_key – One of *CertificatePublicKeyTypes*.

```
>>> from cryptography import x509
>>> csr = x509.load_pem_x509_csr(pem_req_data)
>>> x509.SubjectKeyIdentifier.from_public_key(csr.public_key())
<SubjectKeyIdentifier(digest=b'\x8c"\x98\xe2\xb5\xbf]\xe8*2\xf8\xd2\'?\x00\xd2\
↪xc7#\xe4c')>
```

class cryptography.x509.**SubjectAlternativeName**(*general_names*)

Added in version 0.9.

Subject alternative name is an X.509 extension that provides a list of *general name* instances that provide a set of identities for which the certificate is valid. The object is iterable to get every element.

Parameters
general_names (*list*) – A list of *GeneralName* instances.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *SUBJECT_ALTERNATIVE_NAME*.

get_values_for_type(*type*)

Parameters
type – A *GeneralName* instance. This is one of the *general name classes*.

Returns
A list of values extracted from the matched general names. The type of the returned values depends on the *GeneralName*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.x509.oid import ExtensionOID
>>> cert = x509.load_pem_x509_certificate(cryptography_cert_pem)
>>> # Get the subjectAltName extension from the certificate
>>> ext = cert.extensions.get_extension_for_oid(ExtensionOID.SUBJECT_
↪ALTERNATIVE_NAME)
>>> # Get the dNSName entries from the SAN extension
>>> ext.value.get_values_for_type(x509.DNSName)
['www.cryptography.io', 'cryptography.io']
```

class cryptography.x509.**IssuerAlternativeName**(*general_names*)

Added in version 1.0.

Issuer alternative name is an X.509 extension that provides a list of *general name* instances that provide a set of identities for the certificate issuer. The object is iterable to get every element.

Parameters
general_names (*list*) – A list of *GeneralName* instances.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *ISSUER_ALTERNATIVE_NAME*.

get_values_for_type(*type*)

Parameters
type – A *GeneralName* instance. This is one of the *general name classes*.

Returns
A list of values extracted from the matched general names.

class cryptography.x509.**PrecertificateSignedCertificateTimestamps**(*scts*)

Added in version 2.0.

This extension contains *SignedCertificateTimestamp* instances which were issued for the pre-certificate corresponding to this certificate. These can be used to verify that the certificate is included in a public Certificate Transparency log.

It is an iterable containing one or more *SignedCertificateTimestamp* objects.

Parameters
scts (*list*) – A `list` of *SignedCertificateTimestamp* objects.

oid

Type
ObjectIdentifier

Returns *PRECERT_SIGNED_CERTIFICATE_TIMESTAMPS*.

class cryptography.x509.**PrecertPoison**

Added in version 2.4.

This extension indicates that the certificate should not be treated as a certificate for the purposes of validation, but is instead for submission to a certificate transparency log in order to obtain SCTs which will be embedded in a *PrecertificateSignedCertificateTimestamps* extension on the final certificate.

oid

Type
ObjectIdentifier

Returns *PRECERT_POISON*.

class cryptography.x509.**PrivateKeyUsagePeriod**(*not_before*, *not_after*)

:canonical: cryptography.x509.extensions.PrivateKeyUsagePeriod

Added in version 45.0.0.

This extension defines the period during which the private key corresponding to the certificate's public key may be used. Either `not_before` or `not_after` must be provided.

Parameters

- **not_before** – A `datetime.datetime` object or `None`. Specifies the earliest time the private key can be used.
- **not_after** – A `datetime.datetime` object or `None`. Specifies the latest time the private key can be used.

not_before

A `datetime.datetime` object or `None`. Represents the earliest time the private key can be used.

not_after

A `datetime.datetime` object or `None`. Represents the latest time the private key can be used.

public_bytes()

Returns the encoded bytes of the extension.

return
A `bytes` object containing the encoded extension.

oid

Type
ObjectIdentifier

Returns *PRIVATE_KEY_USAGE_PERIOD*.

class cryptography.x509.**SignedCertificateTimestamps**(*scts*)

Added in version 3.0.

This extension contains *SignedCertificateTimestamp* instances. These can be used to verify that the certificate is included in a public Certificate Transparency log. This extension is only found in OCSP responses. For SCTs in an X.509 certificate see *PrecertificateSignedCertificateTimestamps*.

It is an iterable containing one or more *SignedCertificateTimestamp* objects.

Parameters
scts (*list*) – A `list` of *SignedCertificateTimestamp* objects.

oid

Type
ObjectIdentifier

Returns *SIGNED_CERTIFICATE_TIMESTAMPS*.

class cryptography.x509.**DeltaCRLIndicator**(*crl_number*)

Added in version 2.1.

The delta CRL indicator is a CRL extension that identifies a CRL as being a delta CRL. Delta CRLs contain updates to revocation information previously distributed, rather than all the information that would appear in a complete CRL.

Parameters
crl_number (*int*) – The CRL number of the complete CRL that the delta CRL is updating.

oid

Type
ObjectIdentifier

Returns *DELTA_CRL_INDICATOR*.

crl_number

Type
int

class cryptography.x509.**AuthorityInformationAccess**(*descriptions*)

Added in version 0.9.

The authority information access extension indicates how to access information and services for the issuer of the certificate in which the extension appears. Information and services may include online validation services (such as OCSP) and issuer data. It is an iterable, containing one or more *AccessDescription* instances.

Parameters
descriptions (*list*) – A list of *AccessDescription* objects.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *AUTHORITY_INFORMATION_ACCESS*.

class cryptography.x509.**SubjectInformationAccess**(*descriptions*)

Added in version 3.0.

The subject information access extension indicates how to access information and services for the subject of the certificate in which the extension appears. When the subject is a CA, information and services may include certificate validation services and CA policy data. When the subject is an end entity, the information describes the type of services offered and how to access them. It is an iterable, containing one or more *AccessDescription* instances.

Parameters
descriptions (*list*) – A list of *AccessDescription* objects.

oid

Type
ObjectIdentifier

Returns *SUBJECT_INFORMATION_ACCESS*.

class cryptography.x509.**AccessDescription**(*access_method*, *access_location*)

Added in version 0.9.

access_method

Type
ObjectIdentifier

The access method defines what the `access_location` means. It must be *OCSP* or *CA_ISSUERS* when used with *AuthorityInformationAccess* or *CA_REPOSITORY* when used with *SubjectInformationAccess*.

If it is *OCSP* the access location will be where to obtain OCSP information for the certificate. If it is *CA_ISSUERS* the access location will provide additional information about the issuing certificate. Finally, if it is *CA_REPOSITORY* the access location will be the location of the CA's repository.

access_location

Type
GeneralName

Where to access the information defined by the access method.

class cryptography.x509.**FreshestCRL**(*distribution_points*)

Added in version 2.1.

The freshest CRL extension (also known as Delta CRL Distribution Point) identifies how delta CRL information is obtained. It is an iterable, containing one or more *DistributionPoint* instances.

Parameters
distribution_points (*list*) – A list of *DistributionPoint* instances.

oid

Type
ObjectIdentifier

Returns *FRESHEST_CRL*.

class cryptography.x509.**CRLDistributionPoints**(*distribution_points*)

Added in version 0.9.

The CRL distribution points extension identifies how CRL information is obtained. It is an iterable, containing one or more *DistributionPoint* instances.

Parameters
distribution_points (*list*) – A list of *DistributionPoint* instances.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *CRL_DISTRIBUTION_POINTS*.

class cryptography.x509.**DistributionPoint**(*full_name*, *relative_name*, *reasons*, *crl_issuer*)

Added in version 0.9.

full_name

Type
list of *GeneralName* instances or None

This field describes methods to retrieve the CRL. At most one of `full_name` or `relative_name` will be non-None.

relative_name

Type
RelativeDistinguishedName or None

This field describes methods to retrieve the CRL relative to the CRL issuer. At most one of `full_name` or `relative_name` will be non-None.

Changed in version 1.6: Changed from *Name* to *RelativeDistinguishedName*.

crl_issuer

Type
list of *GeneralName* instances or None

Information about the issuer of the CRL.

reasons

Type
frozenset of *ReasonFlags* or None

The reasons a given distribution point may be used for when performing revocation checks.

class cryptography.x509.**ReasonFlags**

Added in version 0.9.

An enumeration for CRL reasons.

unspecified

It is unspecified why the certificate was revoked. This reason cannot be used as a reason flag in a *DistributionPoint*.

key_compromise

This reason indicates that the private key was compromised.

ca_compromise

This reason indicates that the CA issuing the certificate was compromised.

affiliation_changed

This reason indicates that the subject's name or other information has changed.

superseded

This reason indicates that a certificate has been superseded.

cessation_of_operation

This reason indicates that the certificate is no longer required.

certificate_hold

This reason indicates that the certificate is on hold.

privilege_withdrawn

This reason indicates that the privilege granted by this certificate have been withdrawn.

aa_compromise

When an attribute authority has been compromised.

remove_from_crl

This reason indicates that the certificate was on hold and should be removed from the CRL. This reason cannot be used as a reason flag in a *DistributionPoint*.

class cryptography.x509.**InhibitAnyPolicy**(*skip_certs*)

Added in version 1.0.

The inhibit `anyPolicy` extension indicates that the special OID *ANY_POLICY*, is not considered an explicit match for other *CertificatePolicies* except when it appears in an intermediate self-issued CA certificate. The value indicates the number of additional non-self-issued certificates that may appear in the path before *ANY_POLICY* is no longer permitted. For example, a value of one indicates that *ANY_POLICY* may be processed in certificates issued by the subject of this certificate, but not in additional certificates in the path.

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *INHIBIT_ANY_POLICY*.

skip_certs

Type
int

class cryptography.x509.**PolicyConstraints**

Added in version 1.3.

The policy constraints extension is used to inhibit policy mapping or require that each certificate in a chain contain an acceptable policy identifier. For more information about the use of this extension see **RFC 5280**.

oid

Type
ObjectIdentifier

Returns *POLICY_CONSTRAINTS*.

require_explicit_policy

Type
int or None

If this field is not None, the value indicates the number of additional certificates that may appear in the chain before an explicit policy is required for the entire path. When an explicit policy is required, it is necessary for all certificates in the chain to contain an acceptable policy identifier in the certificate policies extension. An acceptable policy identifier is the identifier of a policy required by the user of the certification path or the identifier of a policy that has been declared equivalent through policy mapping.

inhibit_policy_mapping

Type
int or None

If this field is not None, the value indicates the number of additional certificates that may appear in the chain before policy mapping is no longer permitted. For example, a value of one indicates that policy mapping may be processed in certificates issued by the subject of this certificate, but not in additional certificates in the chain.

class cryptography.x509.**CRLNumber**(*crl_number*)

Added in version 1.2.

The CRL number is a CRL extension that conveys a monotonically increasing sequence number for a given CRL scope and CRL issuer. This extension allows users to easily determine when a particular CRL supersedes another CRL. **RFC 5280** requires that this extension be present in conforming CRLs.

oid

Type
ObjectIdentifier

Returns *CRL_NUMBER*.

crl_number

Type
int

class cryptography.x509.**IssuingDistributionPoint**(*full_name*, *relative_name*, *only_contains_user_certs*, *only_contains_ca_certs*, *only_some_reasons*, *indirect_crl*, *only_contains_attribute_certs*)

Added in version 2.5.

Issuing distribution point is a CRL extension that identifies the CRL distribution point and scope for a particular CRL. It indicates whether the CRL covers revocation for end entity certificates only, CA certificates only, attribute certificates only, or a limited set of reason codes. For specific details on the way this extension should be processed see **RFC 5280**.

oid

Type
ObjectIdentifier

Returns *ISSUING_DISTRIBUTION_POINT*.

only_contains_user_certs

Type
bool

Set to True if the CRL this extension is embedded within only contains information about user certificates.

only_contains_ca_certs

Type
bool

Set to True if the CRL this extension is embedded within only contains information about CA certificates.

indirect_crl

Type
bool

Set to True if the CRL this extension is embedded within includes certificates issued by one or more authorities other than the CRL issuer.

only_contains_attribute_certs

Type
bool

Set to True if the CRL this extension is embedded within only contains information about attribute certificates.

only_some_reasons

Type
frozenset of *ReasonFlags* or None

The reasons for which the issuing distribution point is valid. None indicates that it is valid for all reasons.

full_name

Type
list of *GeneralName* instances or None

This field describes methods to retrieve the CRL. At most one of `full_name` or `relative_name` will be non-None.

relative_name

Type
RelativeDistinguishedName or None

This field describes methods to retrieve the CRL relative to the CRL issuer. At most one of `full_name` or `relative_name` will be non-None.

class cryptography.x509.**UnrecognizedExtension**

Added in version 1.2.

A generic extension class used to hold the raw value of extensions that `cryptography` does not know how to parse. This can also be used when creating new certificates, CRLs, or OCSP requests and responses to encode extensions that `cryptography` does not know how to generate.

oid

Type
ObjectIdentifier

Returns the OID associated with this extension.

value

Type
bytes

Returns the DER encoded bytes payload of the extension.

class cryptography.x509.**MSCertificateTemplate**(*template_id*, *major_version*, *minor_version*)

Added in version 41.0.0.

The Microsoft certificate template extension is a proprietary Microsoft PKI extension that is used to provide information about the template associated with the certificate.

oid

Type
ObjectIdentifier

Returns *MS_CERTIFICATE_TEMPLATE*.

template_id

Type
ObjectIdentifier

major_version

Type
int or None

minor_version

Type
int or None

class cryptography.x509.**CertificatePolicies**(*policies*)

Added in version 0.9.

The certificate policies extension is an iterable, containing one or more *PolicyInformation* instances.

Parameters
policies (*list*) – A list of *PolicyInformation* instances.

As an example of how `CertificatePolicies` might be used, if you wanted to check if a certificated contained the CAB Forum's "domain-validated" policy, you might write code like:

```
def contains_domain_validated(policies):
    return any(
        policy.policy_identifier.dotted_string == "2.23.140.1.2.1"
        for policy in policies
    )
```

oid

Added in version 1.0.

Type
ObjectIdentifier

Returns *CERTIFICATE_POLICIES*.

class cryptography.x509.**Admissions**(*authority*, *admissions*)

Added in version 44.0.0.

The admissions extension contains information on registration and professional admission, as specified by Common PKI v2. It is an iterable, containing one or more *Admission* instances.

oid

Type
ObjectIdentifier

Returns *ADMISSIONS*.

authority

Type
GeneralName or None

An optional identifier of the institution who granted the admissions. This serves as the default value for the admission authority in a single *Admission* if it is not specified there.

Certificate Policies Classes

These classes may be present within a *CertificatePolicies* instance.

class cryptography.x509.**PolicyInformation**(*policy_identifier*, *policy_qualifiers*)

Added in version 0.9.

Contains a policy identifier and an optional list of qualifiers.

policy_identifier

Type
ObjectIdentifier

policy_qualifiers

Type
list

A list consisting of `str` and/or `UserNotice` objects. If the value is `str` it is a pointer to the practice statement published by the certificate authority. If it is a user notice it is meant for display to the relying party when the certificate is used.

class `cryptography.x509.`**`UserNotice`**(*notice_reference*, *explicit_text*)

Added in version 0.9.

User notices are intended for display to a relying party when a certificate is used. In practice, few if any UIs expose this data and it is a rarely encoded component.

`notice_reference`

Type
`NoticeReference` or None

The notice reference field names an organization and identifies, by number, a particular statement prepared by that organization.

`explicit_text`

This field includes an arbitrary textual statement directly in the certificate.

Type
str

class `cryptography.x509.`**`NoticeReference`**(*organization*, *notice_numbers*)

Notice reference can name an organization and provide information about notices related to the certificate. For example, it might identify the organization name and notice number 1. Application software could have a notice file containing the current set of notices for the named organization; the application would then extract the notice text from the file and display it. In practice this is rarely seen.

Added in version 0.9.

`organization`

Type
str

`notice_numbers`

Type
list

A list of integers.

Admissions Classes

These classes may be present within an `Admissions` instance.

class `cryptography.x509.`**`Admission`**(*admission_authority*, *naming_authority*, *profession_infos*)

Added in version 44.0.0.

Contains professional information and optionally the authorization information.

`admission_authority`

Type
`GeneralName` or None

An optional identifier of the institution who granted the admission.

naming_authority

Type
NamingAuthority or None

An optional identifier of the institution who is administering the information of the professions in this admission. This serves as the default value for the naming authority in a single *ProfessionInfo* if it is not specified there.

profession_infos

Type
list

An information on the professions that are part of this admission. This is a list of *ProfessionInfo* objects.

class cryptography.x509.**ProfessionInfo**(*naming_authority*, *profession_items*, *profession_oids*, *registration_number*, *add_profession_info*)

Added in version 44.0.0.

Contains the information for a single profession in the admission.

naming_authority

Type
NamingAuthority or None

An optional identifier of the institution who is administering the information of this profession.

profession_items

Type
list

One or more text strings identifying the profession.

profession_oids

Type
list or None

An optional list of *ObjectIdentifier* elements. Each element in the list corresponds to the resp. text string in the *profession_items* list.

registration_number

Type
str or None

An optional registration number for the profession.

add_profession_info

Type
bytes or None

Optional additional application-specific information in DER-encoded form.

class cryptography.x509.**NamingAuthority**(*id*, *url*, *text*)

Added in version 44.0.0.

Identifies an institution who is responsible for the administration of title registers in an admission. The naming authority can be identified by an object identifier in the field *id*, by the text in the field *text*, by a URL address in the field *url*, or by a combination of them.

id

Type
ObjectIdentifier or None

url

Type
str or None

text

Type
str or None

CRL Entry Extensions

These extensions are only valid within a *RevokedCertificate* object.

class cryptography.x509.**CertificateIssuer**(*general_names*)

Added in version 1.2.

The certificate issuer is an extension that is only valid inside *RevokedCertificate* objects. If the `indirectCRL` property of the parent CRL's IssuingDistributionPoint extension is set, then this extension identifies the certificate issuer associated with the revoked certificate. The object is iterable to get every element.

Parameters
general_names (*list*) – A list of *GeneralName* instances.

oid

Type
ObjectIdentifier

Returns *CERTIFICATE_ISSUER*.

get_values_for_type(*type*)

Parameters
type – A *GeneralName* instance. This is one of the *general name classes*.

Returns
A list of values extracted from the matched general names. The type of the returned values depends on the *GeneralName*.

class cryptography.x509.**CRLReason**(*reason*)

Added in version 1.2.

CRL reason (also known as `reasonCode`) is an extension that is only valid inside *RevokedCertificate* objects. It identifies a reason for the certificate revocation.

Parameters
reason – An element from *ReasonFlags*.

oid

Type
ObjectIdentifier

Returns *CRL_REASON*.

reason

Type
An element from *ReasonFlags*

class cryptography.x509.**InvalidityDate**(*invalidity_date*)

Added in version 1.2.

Invalidity date is an extension that is only valid inside *RevokedCertificate* objects. It provides the date on which it is known or suspected that the private key was compromised or that the certificate otherwise became invalid. This date may be earlier than the revocation date in the CRL entry, which is the date at which the CA processed the revocation.

Parameters
invalidity_date – The datetime.datetime when it is known or suspected that the private key was compromised.

oid

Type
ObjectIdentifier

Returns *INVALIDITY_DATE*.

invalidity_date

Type
datetime.datetime

invalidity_date_utc

Added in version 43.0.0.

Type
datetime.datetime

The invalidity date in UTC as a timezone-aware datetime object.

OCSP Extensions

class cryptography.x509.**OCSPNonce**(*nonce*)

Added in version 2.4.

OCSP nonce is an extension that is only valid inside *OCSPRequest* and *OCSPResponse* objects. The nonce cryptographically binds a request and a response to prevent replay attacks. In practice nonces are rarely used in OCSP due to the desire to precompute OCSP responses at large scale.

oid

Type
ObjectIdentifier

Returns *NONCE*.

nonce

Type
bytes

class cryptography.x509.**OCSPAcceptableResponses**(*response*)

Added in version 41.0.0.

OCSP acceptable responses is an extension that is only valid inside *OCSPRequest* objects. This allows an OCSP client to tell the server what types of responses it supports. In practice this is rarely used, because there is only one kind of OCSP response in wide use.

oid

Type
ObjectIdentifier

Returns *ACCEPTABLE_RESPONSES*.

nonce

Type
bytes

X.509 Request Attributes

class cryptography.x509.**Attributes**

Added in version 36.0.0.

An Attributes instance is an ordered list of attributes. The object is iterable to get every attribute. Each returned element is an *Attribute*.

get_attribute_for_oid(*oid*)

Added in version 36.0.0.

Parameters
oid – An *ObjectIdentifier* instance.

Returns
The *Attribute* or an exception if not found.

Raises
cryptography.x509.AttributeNotFound – If the request does not have the attribute requested.

class cryptography.x509.**Attribute**

Added in version 36.0.0.

An attribute associated with an X.509 request.

oid

Type
ObjectIdentifier

Returns the object identifier for the attribute.

value

Type
bytes

Returns the value of the attribute.

Object Identifiers

X.509 elements are frequently identified by `ObjectIdentifier` instances. The following common OIDs are available as constants.

class `cryptography.x509.oid.`**`NameOID`**

These OIDs are typically seen in X.509 names.

Added in version 1.0.

`COMMON_NAME`

Corresponds to the dotted string `"2.5.4.3"`. Historically the domain name would be encoded here for server certificates. **RFC 2818** deprecates this practice and names of that type should now be located in a `SubjectAlternativeName` extension.

`COUNTRY_NAME`

Corresponds to the dotted string `"2.5.4.6"`.

`LOCALITY_NAME`

Corresponds to the dotted string `"2.5.4.7"`.

`STATE_OR_PROVINCE_NAME`

Corresponds to the dotted string `"2.5.4.8"`.

`STREET_ADDRESS`

Added in version 1.6.

Corresponds to the dotted string `"2.5.4.9"`.

`ORGANIZATION_IDENTIFIER`

Added in version 42.0.0.

Corresponds to the dotted string `"2.5.4.97"`.

`ORGANIZATION_NAME`

Corresponds to the dotted string `"2.5.4.10"`.

`ORGANIZATIONAL_UNIT_NAME`

Corresponds to the dotted string `"2.5.4.11"`.

`SERIAL_NUMBER`

Corresponds to the dotted string `"2.5.4.5"`. This is distinct from the serial number of the certificate itself (which can be obtained with `serial_number`).

`SURNAME`

Corresponds to the dotted string `"2.5.4.4"`.

`GIVEN_NAME`

Corresponds to the dotted string `"2.5.4.42"`.

`TITLE`

Corresponds to the dotted string `"2.5.4.12"`.

`INITIALS`

Added in version 41.0.0.

Corresponds to the dotted string `"2.5.4.43"`.

`GENERATION_QUALIFIER`

Corresponds to the dotted string `"2.5.4.44"`.

X500_UNIQUE_IDENTIFIER

Added in version 1.6.

Corresponds to the dotted string `"2.5.4.45"`.

DN_QUALIFIER

Corresponds to the dotted string `"2.5.4.46"`. This specifies disambiguating information to add to the relative distinguished name of an entry. See **RFC 2256**.

PSEUDONYM

Corresponds to the dotted string `"2.5.4.65"`.

USER_ID

Added in version 1.6.

Corresponds to the dotted string `"0.9.2342.19200300.100.1.1"`.

DOMAIN_COMPONENT

Corresponds to the dotted string `"0.9.2342.19200300.100.1.25"`. A string holding one component of a domain name. See **RFC 4519**.

EMAIL_ADDRESS

Corresponds to the dotted string `"1.2.840.113549.1.9.1"`.

JURISDICTION_COUNTRY_NAME

Corresponds to the dotted string `"1.3.6.1.4.1.311.60.2.1.3"`.

JURISDICTION_LOCALITY_NAME

Corresponds to the dotted string `"1.3.6.1.4.1.311.60.2.1.1"`.

JURISDICTION_STATE_OR_PROVINCE_NAME

Corresponds to the dotted string `"1.3.6.1.4.1.311.60.2.1.2"`.

BUSINESS_CATEGORY

Corresponds to the dotted string `"2.5.4.15"`.

POSTAL_ADDRESS

Added in version 1.6.

Corresponds to the dotted string `"2.5.4.16"`.

POSTAL_CODE

Added in version 1.6.

Corresponds to the dotted string `"2.5.4.17"`.

UNSIGNED

Added in version 50.0.0.

Corresponds to the dotted string `"1.3.6.1.5.5.7.25.1"`. Used in unsigned certificates. See **RFC 9925**.

UNSTRUCTURED_NAME

Added in version 3.0.

Corresponds to the dotted string `"1.2.840.113549.1.9.2"`.

class `cryptography.x509.oid.`**SignatureAlgorithmOID**

Added in version 1.0.

RSA_WITH_MD5

Corresponds to the dotted string `"1.2.840.113549.1.1.4"`. This is an MD5 digest signed by an RSA key.

RSA_WITH_SHA1

Corresponds to the dotted string `"1.2.840.113549.1.1.5"`. This is a SHA1 digest signed by an RSA key.

RSA_WITH_SHA224

Corresponds to the dotted string `"1.2.840.113549.1.1.14"`. This is a SHA224 digest signed by an RSA key.

RSA_WITH_SHA256

Corresponds to the dotted string `"1.2.840.113549.1.1.11"`. This is a SHA256 digest signed by an RSA key.

RSA_WITH_SHA384

Corresponds to the dotted string `"1.2.840.113549.1.1.12"`. This is a SHA384 digest signed by an RSA key.

RSA_WITH_SHA512

Corresponds to the dotted string `"1.2.840.113549.1.1.13"`. This is a SHA512 digest signed by an RSA key.

RSA_WITH_SHA3_224

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.13"`. This is a SHA3-224 digest signed by an RSA key.

RSA_WITH_SHA3_256

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.14"`. This is a SHA3-256 digest signed by an RSA key.

RSA_WITH_SHA3_384

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.15"`. This is a SHA3-384 digest signed by an RSA key.

RSA_WITH_SHA3_512

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.16"`. This is a SHA3-512 digest signed by an RSA key.

RSASSA_PSS

Added in version 2.3.

Corresponds to the dotted string `"1.2.840.113549.1.1.10"`. This is signed by an RSA key using the Probabilistic Signature Scheme (PSS) padding from **RFC 4055**. The hash function and padding are defined by signature algorithm parameters.

ECDSA_WITH_SHA1

Corresponds to the dotted string `"1.2.840.10045.4.1"`. This is a SHA1 digest signed by an ECDSA key.

ECDSA_WITH_SHA224

Corresponds to the dotted string `"1.2.840.10045.4.3.1"`. This is a SHA224 digest signed by an ECDSA key.

ECDSA_WITH_SHA256

Corresponds to the dotted string `"1.2.840.10045.4.3.2"`. This is a SHA256 digest signed by an ECDSA key.

ECDSA_WITH_SHA384

Corresponds to the dotted string `"1.2.840.10045.4.3.3"`. This is a SHA384 digest signed by an ECDSA key.

ECDSA_WITH_SHA512

Corresponds to the dotted string `"1.2.840.10045.4.3.4"`. This is a SHA512 digest signed by an ECDSA key.

ECDSA_WITH_SHA3_224

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.9"`. This is a SHA3-224 digest signed by an ECDSA key.

ECDSA_WITH_SHA3_256

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.10"`. This is a SHA3-256 digest signed by an ECDSA key.

ECDSA_WITH_SHA3_384

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.11"`. This is a SHA3-384 digest signed by an ECDSA key.

ECDSA_WITH_SHA3_512

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.12"`. This is a SHA3-512 digest signed by an ECDSA key.

DSA_WITH_SHA1

Corresponds to the dotted string `"1.2.840.10040.4.3"`. This is a SHA1 digest signed by a DSA key.

DSA_WITH_SHA224

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.1"`. This is a SHA224 digest signed by a DSA key.

DSA_WITH_SHA256

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.2"`. This is a SHA256 digest signed by a DSA key.

DSA_WITH_SHA384

Added in version 36.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.3"`. This is a SHA384 digest signed by a DSA key.

DSA_WITH_SHA512

Added in version 36.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.4"`. This is a SHA512 digest signed by a DSA key.

ED25519

Added in version 2.8.

Corresponds to the dotted string `"1.3.101.112"`. This is a signature using an ed25519 key.

ED448

Added in version 2.8.

Corresponds to the dotted string `"1.3.101.113"`. This is a signature using an ed448 key.

ML_DSA_44

Added in version 49.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.17"`. This is a signature using an ML-DSA-44 key.

ML_DSA_65

Added in version 49.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.18"`. This is a signature using an ML-DSA-65 key.

ML_DSA_87

Added in version 49.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.19"`. This is a signature using an ML-DSA-87 key.

UNSIGNED

Added in version 50.0.0.

Corresponds to the dotted string `"1.3.6.1.5.5.7.6.36"`. This indicates that there is no signature. See **RFC 9925**.

class `cryptography.x509.oid.`**ExtendedKeyUsageOID**

Added in version 1.0.

SERVER_AUTH

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.1"`. This is used to denote that a certificate may be used for TLS web server authentication.

CLIENT_AUTH

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.2"`. This is used to denote that a certificate may be used for TLS web client authentication.

CODE_SIGNING

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.3"`. This is used to denote that a certificate may be used for code signing.

EMAIL_PROTECTION

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.4"`. This is used to denote that a certificate may be used for email protection.

TIME_STAMPING

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.8"`. This is used to denote that a certificate may be used for time stamping.

OCSP_SIGNING

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.9"`. This is used to denote that a certificate may be used for signing OCSP responses.

ANY_EXTENDED_KEY_USAGE

Added in version 2.0.

Corresponds to the dotted string `"2.5.29.37.0"`. This is used to denote that a certificate may be used for _any_ purposes. However, **RFC 5280** additionally notes that applications that require the presence of a particular purpose _MAY_ reject certificates that include the `anyExtendedKeyUsage` OID but not the particular OID expected for the application. Therefore, the presence of this OID does not mean a given application will accept the certificate for all purposes.

SMARTCARD_LOGON

Added in version 35.0.0.

Corresponds to the dotted string `"1.3.6.1.4.1.311.20.2.2"`. This is used to denote that a certificate may be used for `PKINIT` access on Windows.

KERBEROS_PKINIT_KDC

Added in version 35.0.0.

Corresponds to the dotted string `"1.3.6.1.5.2.3.5"`. This is used to denote that a certificate may be used as a Kerberos domain controller certificate authorizing `PKINIT` access. For more information see **RFC 4556**.

IPSEC_IKE

Added in version 37.0.0.

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.17"`. This is used to denote that a certificate may be assigned to an IPSEC SA, and can be used by the assignee to initiate an IPSec Internet Key Exchange (IKE). For more information see **RFC 4945**.

BUNDLE_SECURITY

Added in version 45.0.0.

Corresponds to the dotted string `"1.3.6.1.5.5.7.3.35"`. This is used to denote that a certificate is used by a Bundle Protocol Node to secure data either in transit (e.g. via TLS/TCPCL) or at rest (e.g. via BPSec). For more information see **RFC 9172** and **RFC 9174**.

CERTIFICATE_TRANSPARENCY

Added in version 38.0.0.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.4"`. This is used to denote that a certificate may be used as a pre-certificate signing certificate for Certificate Transparency log operation purposes. For more information see **RFC 6962**.

class `cryptography.x509.oid.`**OtherNameFormOID**

Added in version 45.0.0.

PERMANENT_IDENTIFIER

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.3"`. This is used to correlate multiple certificates which relate to the same entity, as identified by this Other Name value. The Other Name value is encoded as sequence of optional UTF-8 value and optional OID assigner. For more information see **RFC 4043**.

HW_MODULE_NAME

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.4"`. This is used to identify hardware module components when protecting firmware packages. The Other Name value is encoded as sequence of OID hardware-type and octet-string serial number. For more information see **RFC 4108**.

DNS_SRV

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.7"`. This is used to identify service names using qualified DNS name of the form `_Service.Name`. The Other Name value is encoded as IA5 text. For more information see **RFC 4985**.

NAI_REALM

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.8"`. This is used to identify realms for RADIUS dynamic peer discovery using Network Access Identifier (NAI) values. The Other Name value is encoded as UTF-8 text. For more information see **RFC 7585**.

SMTP_UTF8_MAILBOX

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.9"`. This is used to identify an internationalized email address associated with an entity. The Other Name value is encoded as UTF-8 text. For more information see **RFC 9598**.

ACP_NODE_NAME

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.10"`. This is used to identify a single node within an Autonomic Control Plane (ACP). The Other Name value is encoded as IA5 text. For more information see **RFC 8994**.

BUNDLE_EID

Corresponds to the dotted string `"1.3.6.1.5.5.7.8.11"`. This is used to contain the text form of an endpoint identifier (EID) for the Bundle Protocol Version 7. The Other Name value is encoded as IA5 text. For more information see **RFC 9171** and **RFC 9174**.

class cryptography.x509.oid.**AuthorityInformationAccessOID**

Added in version 1.0.

OCSP

Corresponds to the dotted string `"1.3.6.1.5.5.7.48.1"`. Used as the identifier for OCSP data in *AccessDescription* objects.

CA_ISSUERS

Corresponds to the dotted string `"1.3.6.1.5.5.7.48.2"`. Used as the identifier for CA issuer data in *AccessDescription* objects.

class cryptography.x509.oid.**SubjectInformationAccessOID**

Added in version 3.0.

CA_REPOSITORY

Corresponds to the dotted string `"1.3.6.1.5.5.7.48.5"`. Used as the identifier for CA repository data in *AccessDescription* objects.

class cryptography.x509.oid.**CertificatePoliciesOID**

Added in version 1.0.

CPS_QUALIFIER

Corresponds to the dotted string `"1.3.6.1.5.5.7.2.1"`.

CPS_USER_NOTICE

Corresponds to the dotted string `"1.3.6.1.5.5.7.2.2"`.

ANY_POLICY

Corresponds to the dotted string `"2.5.29.32.0"`.

class cryptography.x509.oid.**ExtensionOID**

Added in version 1.0.

BASIC_CONSTRAINTS

Corresponds to the dotted string `"2.5.29.19"`. The identifier for the *BasicConstraints* extension type.

KEY_USAGE

Corresponds to the dotted string `"2.5.29.15"`. The identifier for the *KeyUsage* extension type.

SUBJECT_ALTERNATIVE_NAME

Corresponds to the dotted string `"2.5.29.17"`. The identifier for the *SubjectAlternativeName* extension type.

ISSUER_ALTERNATIVE_NAME

Corresponds to the dotted string `"2.5.29.18"`. The identifier for the *IssuerAlternativeName* extension type.

SUBJECT_KEY_IDENTIFIER

Corresponds to the dotted string `"2.5.29.14"`. The identifier for the *SubjectKeyIdentifier* extension type.

NAME_CONSTRAINTS

Corresponds to the dotted string `"2.5.29.30"`. The identifier for the *NameConstraints* extension type.

CRL_DISTRIBUTION_POINTS

Corresponds to the dotted string `"2.5.29.31"`. The identifier for the *CRLDistributionPoints* extension type.

CERTIFICATE_POLICIES

Corresponds to the dotted string `"2.5.29.32"`. The identifier for the *CertificatePolicies* extension type.

AUTHORITY_KEY_IDENTIFIER

Corresponds to the dotted string `"2.5.29.35"`. The identifier for the *AuthorityKeyIdentifier* extension type.

EXTENDED_KEY_USAGE

Corresponds to the dotted string `"2.5.29.37"`. The identifier for the *ExtendedKeyUsage* extension type.

AUTHORITY_INFORMATION_ACCESS

Corresponds to the dotted string `"1.3.6.1.5.5.7.1.1"`. The identifier for the *AuthorityInformationAccess* extension type.

SUBJECT_INFORMATION_ACCESS

Added in version 3.0.

Corresponds to the dotted string `"1.3.6.1.5.5.7.1.11"`. The identifier for the *SubjectInformationAccess* extension type.

INHIBIT_ANY_POLICY

Corresponds to the dotted string `"2.5.29.54"`. The identifier for the *InhibitAnyPolicy* extension type.

OCSP_NO_CHECK

Corresponds to the dotted string `"1.3.6.1.5.5.7.48.1.5"`. The identifier for the *OCSPNoCheck* extension type.

TLS_FEATURE

Corresponds to the dotted string `"1.3.6.1.5.5.7.1.24"`. The identifier for the *TLSFeature* extension type.

CRL_NUMBER

Corresponds to the dotted string `"2.5.29.20"`. The identifier for the `CRLNumber` extension type. This extension only has meaning for certificate revocation lists.

DELTA_CRL_INDICATOR

Added in version 2.1.

Corresponds to the dotted string `"2.5.29.27"`. The identifier for the `DeltaCRLIndicator` extension type. This extension only has meaning for certificate revocation lists.

PRECERT_SIGNED_CERTIFICATE_TIMESTAMPS

Added in version 1.9.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.2"`.

PRECERT_POISON

Added in version 2.4.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.3"`.

PRIVATE_KEY_USAGE_PERIOD

Added in version 45.0.0.

Corresponds to the dotted string `"2.5.29.16"`.

SIGNED_CERTIFICATE_TIMESTAMPS

Added in version 3.0.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.5"`.

POLICY_CONSTRAINTS

Corresponds to the dotted string `"2.5.29.36"`. The identifier for the *PolicyConstraints* extension type.

FRESHEST_CRL

Corresponds to the dotted string `"2.5.29.46"`. The identifier for the *FreshestCRL* extension type.

ISSUING_DISTRIBUTION_POINT

Added in version 2.4.

Corresponds to the dotted string `"2.5.29.28"`.

POLICY_MAPPINGS

Corresponds to the dotted string `"2.5.29.33"`.

SUBJECT_DIRECTORY_ATTRIBUTES

Corresponds to the dotted string `"2.5.29.9"`.

MS_CERTIFICATE_TEMPLATE

Added in version 41.0.0.

Corresponds to the dotted string `"1.3.6.1.4.1.311.21.7"`.

ADMISSIONS

Added in version 44.0.0.

Corresponds to the dotted string `"1.3.36.8.3.3"`.

class `cryptography.x509.oid.`**CRLEntryExtensionOID**

Added in version 1.2.

CERTIFICATE_ISSUER

Corresponds to the dotted string `"2.5.29.29"`.

CRL_REASON

Corresponds to the dotted string `"2.5.29.21"`.

INVALIDITY_DATE

Corresponds to the dotted string `"2.5.29.24"`.

class cryptography.x509.oid.**OCSPExtensionOID**

Added in version 2.4.

NONCE

Corresponds to the dotted string "1.3.6.1.5.5.7.48.1.2".

ACCEPTABLE_RESPONSES

Added in version 41.0.0.

Corresponds to the dotted string "1.3.6.1.5.5.7.48.1.4".

class cryptography.x509.oid.**AttributeOID**

Added in version 3.0.

CHALLENGE_PASSWORD

Corresponds to the dotted string "1.2.840.113549.1.9.7".

UNSTRUCTURED_NAME

Corresponds to the dotted string "1.2.840.113549.1.9.2".

class cryptography.x509.oid.**PublicKeyAlgorithmOID**

Added in version 43.0.0.

DSA

Corresponds to the dotted string "1.2.840.10040.4.1". This is a *DSAPublicKey* public key.

EC_PUBLIC_KEY

Corresponds to the dotted string "1.2.840.10045.2.1". This is a *EllipticCurvePublicKey* public key.

RSAES_PKCS1_v1_5

Corresponds to the dotted string "1.2.840.113549.1.1.1". This is a *RSAPublicKey* public key with *PKCS1v15* padding.

RSASSA_PSS

Corresponds to the dotted string "1.2.840.113549.1.1.10". This is a *RSAPublicKey* public key with *PSS* padding.

X25519

Corresponds to the dotted string "1.3.101.110". This is a *X25519PublicKey* public key.

X448

Corresponds to the dotted string "1.3.101.111". This is a *X448PublicKey* public key.

ED25519

Corresponds to the dotted string "1.3.101.112". This is a *Ed25519PublicKey* public key.

ED448

Corresponds to the dotted string "1.3.101.113". This is a *Ed448PublicKey* public key.

ML_DSA_44

Added in version 49.0.0.

Corresponds to the dotted string "2.16.840.1.101.3.4.3.17". This is a *MLDSA44PublicKey* public key.

ML_DSA_65

Added in version 49.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.18"`. This is a *MLDSA65PublicKey* public key.

ML_DSA_87

Added in version 49.0.0.

Corresponds to the dotted string `"2.16.840.1.101.3.4.3.19"`. This is a *MLDSA87PublicKey* public key.

Helper Functions

`cryptography.x509.random_serial_number()`

Added in version 1.6.

Generates a random serial number suitable for use when constructing certificates.

Exceptions

class `cryptography.x509.InvalidVersion`

This is raised when an X.509 certificate has an invalid version number.

parsed_version

Type
int

Returns the raw version that was parsed from the certificate.

class `cryptography.x509.DuplicateExtension`

This is raised when more than one X.509 extension of the same type is found within a certificate.

oid

Type
ObjectIdentifier

Returns the OID.

class `cryptography.x509.ExtensionNotFound`

This is raised when calling *Extensions.get_extension_for_oid()* with an extension OID that is not present in the certificate.

oid

Type
ObjectIdentifier

Returns the OID.

class `cryptography.x509.AttributeNotFound`

This is raised when calling *Attributes.get_attribute_for_oid()* with an attribute OID that is not present in the request.

oid

Type
ObjectIdentifier

Returns the OID.

class cryptography.x509.**UnsupportedGeneralNameType**

This is raised when a certificate contains an unsupported general name type in an extension.

type

Type
int

The integer value of the unsupported type. The complete list of types can be found in **RFC 5280 Section 4.2.1.6**.

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.

2.4 Primitives

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.

2.4.1 Authenticated encryption

Authenticated encryption with associated data (AEAD) are encryption schemes which provide both confidentiality and integrity for their ciphertext. They also support providing integrity for associated data which is not encrypted.

class cryptography.hazmat.primitives.ciphers.aead.**ChaCha20Poly1305**(*key*)

Added in version 2.0.

The ChaCha20Poly1305 construction is defined in **RFC 7539** section 2.8. It is a stream cipher combined with a MAC that offers strong integrity guarantees.

Parameters
key (*bytes-like*) – A 32-byte key. This **must** be kept secret.

Raises
cryptography.exceptions.UnsupportedAlgorithm – If the version of OpenSSL does not support ChaCha20Poly1305.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
>>> data = b"a secret message"
>>> aad = b"authenticated but unencrypted data"
>>> key = ChaCha20Poly1305.generate_key()
>>> chacha = ChaCha20Poly1305(key)
>>> nonce = os.urandom(12)
>>> ct = chacha.encrypt(nonce, data, aad)
>>> chacha.decrypt(nonce, ct, aad)
b'a secret message'
```

classmethod generate_key()

Securely generates a random ChaCha20Poly1305 key.

Returns bytes
A 32 byte key.

`encrypt`(*nonce*, *data*, *associated_data*)

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts the `data` provided and authenticates the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **`nonce`** (*bytes-like*) – A 12 byte value. **NEVER REUSE A NONCE** with a key.
- **`data`** (*bytes-like*) – The data to encrypt.
- **`associated_data`** (*bytes-like*) – Additional data that should be authenticated with the key, but does not need to be encrypted. Can be `None`.

Returns bytes
The ciphertext bytes with the 16 byte tag appended.

Raises
`OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

`encrypt_into`(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **`nonce`** (*bytes-like*) – A 12 byte value. **NEVER REUSE A NONCE** with a key.
- **`data`** (*bytes-like*) – The data to encrypt.
- **`associated_data`** (*bytes-like*) – Additional data that should be authenticated with the key, but does not need to be encrypted. Can be `None`.
- **`buf`** – A writable *bytes-like* object that must be exactly `len(data) + 16` bytes. The ciphertext with the 16 byte tag appended will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) + 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- `OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

decrypt(*nonce*, *data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **nonce** (*bytes-like*) – A 12 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

Returns bytes
The original plaintext.

Raises
cryptography.exceptions.InvalidTag – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

decrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A 12 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) - 16` bytes. The plaintext will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) - 16`).

Raises

- **ValueError** – If the buffer is not the correct size.
- *cryptography.exceptions.InvalidTag* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

class cryptography.hazmat.primitives.ciphers.aead.**AESGCM**(*key*)

Added in version 2.0.

> **Note**
>
> This class only supports 128-bit tags. If you need tag truncation (which is generally **not recommended**) you should use the *GCM* class with *Cipher*.

The AES-GCM construction is composed of the *AES* block cipher utilizing Galois Counter Mode (GCM).

Parameters

key (*bytes-like*) – A 128, 192, or 256-bit key. This **must** be kept secret.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import AESGCM
>>> data = b"a secret message"
>>> aad = b"authenticated but unencrypted data"
>>> key = AESGCM.generate_key(bit_length=128)
>>> aesgcm = AESGCM(key)
>>> nonce = os.urandom(12)
>>> ct = aesgcm.encrypt(nonce, data, aad)
>>> aesgcm.decrypt(nonce, ct, aad)
b'a secret message'
```

classmethod generate_key(*bit_length*)

Securely generates a random AES-GCM key.

Parameters

bit_length – The bit length of the key to generate. Must be 128, 192, or 256.

Returns bytes

The generated key.

encrypt(*nonce*, *data*, *associated_data*)

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **nonce** (*bytes-like*) – NIST recommends a 96-bit IV length for best performance but it can be up to 2^{64} - 1 *bits*. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.

Returns bytes

The ciphertext bytes with the 16 byte tag appended.

Raises

`OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

encrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – NIST recommends a 96-bit IV length for best performance but it can be up to 2^{64} - 1 *bits*. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) + 16` bytes. The ciphertext with the 16 byte tag appended will be written to this buffer.

Returns int

The number of bytes written to the buffer (always `len(data) + 16`).

Raises

- **ValueError** – If the buffer is not the correct size.
- **OverflowError** – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

decrypt(*nonce*, *data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **nonce** (*bytes-like*) – NIST recommends a 96-bit IV length for best performance but it can be up to 2^{64} - 1 *bits*. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

Returns bytes

The original plaintext.

Raises

cryptography.exceptions.InvalidTag – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

decrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – NIST recommends a 96-bit IV length for best performance but it can be up to 2^{64} - 1 *bits*. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

- **buf** – A writable *bytes-like* object that must be exactly `len(data) - 16` bytes. The plaintext will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) - 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- *`cryptography.exceptions.InvalidTag`* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

class `cryptography.hazmat.primitives.ciphers.aead.`**`AESGCMSIV`**(*key*)

Added in version 42.0.0.

The AES-GCM-SIV construction is defined in **RFC 8452** and is composed of the *`AES`* block cipher utilizing Galois Counter Mode (GCM) and a synthetic initialization vector (SIV).

Parameters
key (*bytes-like*) – A 128, 192, or 256-bit key. This **must** be kept secret.

Raises
`cryptography.exceptions.UnsupportedAlgorithm` – If the version of OpenSSL does not support AES-GCM-SIV.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import AESGCMSIV
>>> data = b"a secret message"
>>> aad = b"authenticated but unencrypted data"
>>> key = AESGCMSIV.generate_key(bit_length=128)
>>> aesgcmsiv = AESGCMSIV(key)
>>> nonce = os.urandom(12)
>>> ct = aesgcmsiv.encrypt(nonce, data, aad)
>>> aesgcmsiv.decrypt(nonce, ct, aad)
b'a secret message'
```

classmethod **`generate_key`**(*bit_length*)

Securely generates a random AES-GCM-SIV key.

Parameters
bit_length – The bit length of the key to generate. Must be 128, 192, or 256.

Returns bytes
The generated key.

`encrypt`(*nonce*, *data*, *associated_data*)

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **nonce** (*bytes-like*) – A 12-byte value.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.

Returns bytes
The ciphertext bytes with the 16 byte tag appended.

Raises
`OverflowError` – If `data` or `associated_data` is larger than 2^{32} - 1 bytes.

`encrypt_into`(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A 12-byte value.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) + 16` bytes. The ciphertext with the 16 byte tag appended will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) + 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- `OverflowError` – If `data` or `associated_data` is larger than 2^{32} - 1 bytes.

`decrypt`(*nonce*, *data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **nonce** (*bytes-like*) – A 12-byte value.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

Returns bytes
The original plaintext.

Raises
`cryptography.exceptions.InvalidTag` – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

`decrypt_into`(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A 12-byte value.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).

- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) - 16` bytes. The plaintext will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) - 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- *cryptography.exceptions.InvalidTag* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

class cryptography.hazmat.primitives.ciphers.aead.**AESOCB3**(*key*)

Added in version 36.0.0.

The OCB3 construction is defined in RFC 7253. It is an AEAD mode that offers strong integrity guarantees and good performance.

Parameters
key (*bytes-like*) – A 128, 192, or 256-bit key. This **must** be kept secret.

Raises
cryptography.exceptions.UnsupportedAlgorithm – If the version of OpenSSL does not support AES-OCB3.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import AESOCB3
>>> data = b"a secret message"
>>> aad = b"authenticated but unencrypted data"
>>> key = AESOCB3.generate_key(bit_length=128)
>>> aesocb = AESOCB3(key)
>>> nonce = os.urandom(12)
>>> ct = aesocb.encrypt(nonce, data, aad)
>>> aesocb.decrypt(nonce, ct, aad)
b'a secret message'
```

classmethod generate_key(*bit_length*)

Securely generates a random AES-OCB3 key.

Parameters
bit_length – The bit length of the key to generate. Must be 128, 192, or 256.

Returns bytes
The generated key.

encrypt(*nonce*, *data*, *associated_data*)

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **nonce** (*bytes-like*) – A 12-15 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.

Returns bytes
The ciphertext bytes with the 16 byte tag appended.

Raises
`OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

encrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A 12-15 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) + 16` bytes. The ciphertext with the 16 byte tag appended will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) + 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- `OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

decrypt(*nonce*, *data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **nonce** (*bytes-like*) – A 12 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

Returns bytes
The original plaintext.

Raises

cryptography.exceptions.InvalidTag – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

decrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A 12-15 byte value. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) - 16` bytes. The plaintext will be written to this buffer.

Returns int

The number of bytes written to the buffer (always `len(data) - 16`).

Raises

- **ValueError** – If the buffer is not the correct size.
- *cryptography.exceptions.InvalidTag* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

class cryptography.hazmat.primitives.ciphers.aead.**AESSIV**(*key*)

Added in version 37.0.0.

The SIV (synthetic initialization vector) construction is defined in **RFC 5297**. Depending on how it is used, SIV allows either deterministic authenticated encryption or nonce-based, misuse-resistant authenticated encryption.

Parameters

key (*bytes-like*) – A 256, 384, or 512-bit key (double sized from typical AES). This **must** be kept secret.

Raises

cryptography.exceptions.UnsupportedAlgorithm – If the version of OpenSSL does not support AES-SIV.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import AESSIV
>>> data = b"a secret message"
>>> nonce = os.urandom(16)
>>> aad = [b"authenticated but unencrypted data", nonce]
>>> key = AESSIV.generate_key(bit_length=512)  # AES256 requires 512-bit keys for␣
↪SIV
>>> aessiv = AESSIV(key)
>>> ct = aessiv.encrypt(data, aad)
>>> aessiv.decrypt(ct, aad)
b'a secret message'
```

classmethod generate_key(*bit_length*)

Securely generates a random AES-SIV key.

Parameters

bit_length – The bit length of the key to generate. Must be 256, 384, or 512. AES-SIV splits the key into an encryption and MAC key, so these lengths correspond to AES 128, 192, and 256.

Returns bytes

The generated key.

encrypt(*data*, *associated_data*)

> **Note**
>
> SIV performs nonce-based authenticated encryption when a component of the associated data is a nonce. The final associated data in the list is used for the nonce.
>
> Random nonces should have at least 128-bits of entropy. If a nonce is reused with SIV authenticity is retained and confidentiality is only compromised to the extent that an attacker can determine that the same plaintext (and same associated data) was protected with the same nonce and key.
>
> If you do not supply a nonce encryption is deterministic and the same (plaintext, key) pair will always produce the same ciphertext.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*list*) – An optional `list` of `bytes-like objects`. This is additional data that should be authenticated with the key, but is not encrypted. Can be `None`. In SIV mode the final element of this list is treated as a `nonce`.

Returns bytes

The ciphertext bytes with the 16 byte tag **prepended**.

Raises

OverflowError – If `data` or an `associated_data` element is larger than 2^{31} - 1 bytes.

encrypt_into(*data*, *associated_data*, *buf*)

Added in version 47.0.0.

> **Note**
>
> SIV performs nonce-based authenticated encryption when a component of the associated data is a nonce. The final associated data in the list is used for the nonce.
>
> Random nonces should have at least 128-bits of entropy. If a nonce is reused with SIV authenticity is retained and confidentiality is only compromised to the extent that an attacker can determine that the same plaintext (and same associated data) was protected with the same nonce and key.
>
> If you do not supply a nonce encryption is deterministic and the same (plaintext, key) pair will always produce the same ciphertext.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*list*) – An optional `list` of `bytes-like objects`. This is additional data that should be authenticated with the key, but is not encrypted. Can be `None`. In SIV mode the final element of this list is treated as a `nonce`.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) + 16` bytes. The ciphertext with the 16 byte tag **prepended** will be written to this buffer.

Returns int

The number of bytes written to the buffer (always `len(data) + 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- `OverflowError` – If `data` or an `associated_data` element is larger than 2^{31} - 1 bytes.

decrypt(*data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **data** (*bytes*) – The data to decrypt (with tag **prepended**).
- **associated_data** (*list*) – An optional `list` of `bytes-like objects`. This is additional data that should be authenticated with the key, but is not encrypted. Can be `None` if none was used during encryption.

Returns bytes

The original plaintext.

Raises

cryptography.exceptions.InvalidTag – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key or associated data are wrong.

decrypt_into(*data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **data** (*bytes*) – The data to decrypt (with tag **prepended**).
- **associated_data** (*list*) – An optional `list` of `bytes-like objects`. This is additional data that should be authenticated with the key, but is not encrypted. Can be `None` if none was used during encryption.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) - 16` bytes. The plaintext will be written to this buffer.

Returns int

The number of bytes written to the buffer (always `len(data) - 16`).

Raises

- `ValueError` – If the buffer is not the correct size.
- *`cryptography.exceptions.InvalidTag`* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key or associated data are wrong.

class `cryptography.hazmat.primitives.ciphers.aead.`**`AESCCM`**(*key*, *tag_length=16*)

Added in version 2.0.

The AES-CCM construction is composed of the *`AES`* block cipher utilizing Counter with CBC-MAC (CCM) (specified in **RFC 3610**).

Parameters

- **`key`** (*bytes-like*) – A 128, 192, or 256-bit key. This **must** be kept secret.
- **`tag_length`** (*`int`*) – The length of the authentication tag. This defaults to 16 bytes and it is **strongly** recommended that you do not make it shorter unless absolutely necessary. Valid tag lengths are 4, 6, 8, 10, 12, 14, and 16.

Raises

`cryptography.exceptions.UnsupportedAlgorithm` – If the version of OpenSSL does not support AES-CCM.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers.aead import AESCCM
>>> data = b"a secret message"
>>> aad = b"authenticated but unencrypted data"
>>> key = AESCCM.generate_key(bit_length=128)
>>> aesccm = AESCCM(key)
>>> nonce = os.urandom(13)
>>> ct = aesccm.encrypt(nonce, data, aad)
>>> aesccm.decrypt(nonce, ct, aad)
b'a secret message'
```

classmethod **`generate_key`**(*bit_length*)

Securely generates a random AES-CCM key.

Parameters

`bit_length` – The bit length of the key to generate. Must be 128, 192, or 256.

Returns bytes

The generated key.

`encrypt`(*nonce*, *data*, *associated_data*)

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output of this can be passed directly to the `decrypt` method.

Parameters

- **nonce** (*bytes-like*) – A value of between 7 and 13 bytes. The maximum length is determined by the length of the ciphertext you are encrypting and must satisfy the condition: `len(data) < 2 ** (8 * (15 - len(nonce)))` **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.

Returns bytes
The ciphertext bytes with the tag appended.

Raises
`OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.

encrypt_into(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

> **Warning**
>
> Reuse of a `nonce` with a given `key` compromises the security of any message with that `nonce` and `key` pair.

Encrypts and authenticates the `data` provided as well as authenticating the `associated_data`. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A value of between 7 and 13 bytes. The maximum length is determined by the length of the ciphertext you are encrypting and must satisfy the condition: `len(data) < 2 ** (8 * (15 - len(nonce)))` **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to encrypt.
- **associated_data** (*bytes-like*) – Additional data that should be authenticated with the key, but is not encrypted. Can be `None`.
- **buf** – A writable *bytes-like* buffer into which the ciphertext and tag will be written. This must be exactly `len(data) + tag_length` bytes long.

Returns int
The number of bytes written to the buffer (this is always `len(data) + tag_length`).

Raises

- `OverflowError` – If `data` or `associated_data` is larger than 2^{31} - 1 bytes.
- `ValueError` – If `buf` is not the correct length.

decrypt(*nonce*, *data*, *associated_data*)

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail.

Parameters

- **nonce** (*bytes-like*) – A value of between 7 and 13 bytes. This is the same value used when you originally called encrypt. **NEVER REUSE A NONCE** with a key.

- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.

Returns bytes
The original plaintext.

Raises
cryptography.exceptions.InvalidTag – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

`decrypt_into`(*nonce*, *data*, *associated_data*, *buf*)

Added in version 47.0.0.

Decrypts the `data` and authenticates the `associated_data`. If you called encrypt with `associated_data` you must pass the same `associated_data` in decrypt or the integrity check will fail. The output is written into the `buf` parameter.

Parameters

- **nonce** (*bytes-like*) – A value of between 7 and 13 bytes. This is the same value used when you originally called encrypt. **NEVER REUSE A NONCE** with a key.
- **data** (*bytes-like*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes-like*) – Additional data to authenticate. Can be `None` if none was passed during encryption.
- **buf** – A writable *bytes-like* object that must be exactly `len(data) - tag_length` bytes. The plaintext will be written to this buffer.

Returns int
The number of bytes written to the buffer (always `len(data) - tag_length`).

Raises

- `ValueError` – If the buffer is not the correct size.
- *cryptography.exceptions.InvalidTag* – If the authentication tag doesn't validate this exception will be raised. This will occur when the ciphertext has been changed, but will also occur when the key, nonce, or associated data are wrong.

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.

2.4.2 Asymmetric algorithms

Asymmetric cryptography is a branch of cryptography where a secret key can be divided into two parts, a *public key* and a *private key*. The public key can be given to anyone, trusted or not, while the private key must be kept secret (just like the key in symmetric cryptography).

Asymmetric cryptography has two primary use cases: authentication and confidentiality. Using asymmetric cryptography, messages can be signed with a private key, and then anyone with the public key is able to verify that the message was created by someone possessing the corresponding private key. This can be combined with a proof of identity system to know what entity (person or group) actually owns that private key, providing authentication.

Encryption with asymmetric cryptography works in a slightly different way from symmetric encryption. Someone with the public key is able to encrypt a message, providing confidentiality, and then only the person in possession of the private key is able to decrypt it.

> 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.

ML-DSA signing

ML-DSA is a post-quantum digital signature algorithm based on module lattices, standardized in FIPS 204.

Signing & Verification

```
>>> from cryptography.hazmat.primitives.asymmetric.mldsa import MLDSA65PrivateKey
>>> private_key = MLDSA65PrivateKey.generate()
>>> signature = private_key.sign(b"my authenticated message")
>>> public_key = private_key.public_key()
>>> public_key.verify(signature, b"my authenticated message")
```

Context-based Signing & Verification

ML-DSA supports context strings to bind additional information to signatures. The context can be up to 255 bytes and is used to differentiate signatures in different contexts or protocols.

```
>>> from cryptography.hazmat.primitives.asymmetric.mldsa import MLDSA65PrivateKey
>>> private_key = MLDSA65PrivateKey.generate()
>>> context = b"email-signature-v1"
>>> signature = private_key.sign(b"my authenticated message", context)
>>> public_key = private_key.public_key()
>>> # Verification requires the same context
>>> public_key.verify(signature, b"my authenticated message", context)
```

External mu

ML-DSA supports an "external mu" mode, where the message representative `mu` is computed separately from signing and verification (see *`MLDSA65PrivateKey.sign_mu()`* and *`MLDSA65PublicKey.verify_mu()`*). *`MLDSAMuHasher`* computes `mu` incrementally, which is useful when the message is large or only available as a stream.

```
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA65PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> hasher = mldsa.MLDSAMuHasher(public_key)
>>> hasher.update(b"my authenticated ")
>>> hasher.update(b"message")
>>> mu = hasher.finalize()
>>> signature = private_key.sign_mu(mu)
>>> public_key.verify_mu(signature, mu)
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSA44PrivateKey**

Added in version 47.0.0.

classmethod generate()

Generate an ML-DSA-44 private key.

Returns
MLDSA44PrivateKey

Raises
cryptography.exceptions.UnsupportedAlgorithm – If ML-DSA-44 is not supported by the backend `cryptography` is using.

classmethod from_seed_bytes(*data*)

Load an ML-DSA-44 private key from seed bytes.

Parameters
data (*bytes-like*) – 32 byte seed.

Returns
MLDSA44PrivateKey

Raises

- **ValueError** – If the seed is not 32 bytes.
- *cryptography.exceptions.UnsupportedAlgorithm* – If ML-DSA-44 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA44PrivateKey.generate()
>>> seed = private_key.private_bytes_raw()
>>> same_key = mldsa.MLDSA44PrivateKey.from_seed_bytes(seed)
```

public_key()

Returns
MLDSA44PublicKey

sign(*data*, *context=None*)

Sign the data using ML-DSA-44. An optional context string can be provided.

Parameters

- **data** (*bytes-like*) – The data to sign.
- **context** (*bytes-like* or `None`) – An optional context string (up to 255 bytes).

Returns bytes
The signature (2420 bytes).

Raises
ValueError – If the context is longer than 255 bytes.

sign_mu(*mu*)

Added in version 49.0.0.

Sign a precomputed `mu` (message representative) using ML-DSA-44, the "external mu" variant from FIPS 204. `mu` can be computed incrementally with *MLDSAMuHasher*.

Parameters
mu (*bytes-like*) – The 64-byte message representative.

Returns bytes
The signature (2420 bytes).

Raises
ValueError – If mu is not 64 bytes.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

This method only returns the serialization of the seed form of the private key, never the expanded one.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the encoding is *Raw* then format must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

This method only returns the seed form of the private key (32 bytes).

Return bytes
Raw key (32-byte seed).

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSA44PublicKey**

Added in version 47.0.0.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 1312 byte public key.

Returns
MLDSA44PublicKey

Raises

- **ValueError** – If the public key is not 1312 bytes.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If ML-DSA-44 is not supported by the backend cryptography is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA44PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
```

(continues on next page)

(continued from previous page)

```
... )
>>> loaded_public_key = mldsa.MLDSA44PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
1312-byte raw public key.

verify(*signature*, *data*, *context=None*)

Verify a signature using ML-DSA-44. If a context string was used during signing, the same context must be provided for verification to succeed.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The data to verify.
- **context** (*bytes-like* or `None`) – An optional context string (up to 255 bytes) that was used during signing.

Returns
None

Raises

- ***cryptography.exceptions.InvalidSignature*** – Raised when the signature cannot be verified.
- **ValueError** – If the context is longer than 255 bytes.

verify_mu(*signature*, *mu*)

Added in version 49.0.0.

Verify a signature over a precomputed `mu` (message representative), the "external mu" variant from FIPS 204. `mu` can be computed incrementally with *MLDSAMuHasher*.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **mu** (*bytes-like*) – The 64-byte message representative.

Returns
None

Raises

- *cryptography.exceptions.InvalidSignature* – Raised when the signature cannot be verified.
- ValueError – If mu is not 64 bytes.

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSA65PrivateKey**

Added in version 47.0.0.

classmethod generate()

Generate an ML-DSA-65 private key.

Returns
MLDSA65PrivateKey

Raises
cryptography.exceptions.UnsupportedAlgorithm – If ML-DSA-65 is not supported by the backend cryptography is using.

classmethod from_seed_bytes(*data*)

Load an ML-DSA-65 private key from seed bytes.

Parameters
data (*bytes-like*) – 32 byte seed.

Returns
MLDSA65PrivateKey

Raises

- ValueError – If the seed is not 32 bytes.
- *cryptography.exceptions.UnsupportedAlgorithm* – If ML-DSA-65 is not supported by the backend cryptography is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA65PrivateKey.generate()
>>> seed = private_key.private_bytes_raw()
>>> same_key = mldsa.MLDSA65PrivateKey.from_seed_bytes(seed)
```

public_key()

Returns
MLDSA65PublicKey

sign(*data*, *context=None*)

Sign the data using ML-DSA-65. An optional context string can be provided.

Parameters

- **data** (*bytes-like*) – The data to sign.
- **context** (*bytes-like* or None) – An optional context string (up to 255 bytes).

Returns bytes
The signature (3309 bytes).

Raises
ValueError – If the context is longer than 255 bytes.

sign_mu(*mu*)

Added in version 49.0.0.

Sign a precomputed `mu` (message representative) using ML-DSA-65, the "external mu" variant from FIPS 204. `mu` can be computed incrementally with *MLDSAMuHasher*.

Parameters
mu (*bytes-like*) – The 64-byte message representative.

Returns bytes
The signature (3309 bytes).

Raises
ValueError – If `mu` is not 64 bytes.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

This method only returns the serialization of the seed form of the private key, never the expanded one.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

This method only returns the seed form of the private key (32 bytes).

Return bytes
Raw key (32-byte seed).

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSA65PublicKey**

Added in version 47.0.0.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 1952 byte public key.

Returns
MLDSA65PublicKey

Raises

- **ValueError** – If the public key is not 1952 bytes.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If ML-DSA-65 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA65PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = mldsa.MLDSA65PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes

The public key bytes.

public_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes

1952-byte raw public key.

verify(*signature*, *data*, *context=None*)

Verify a signature using ML-DSA-65. If a context string was used during signing, the same context must be provided for verification to succeed.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The data to verify.
- **context** (*bytes-like* or `None`) – An optional context string (up to 255 bytes) that was used during signing.

Returns

None

Raises

- ***cryptography.exceptions.InvalidSignature*** – Raised when the signature cannot be verified.
- **ValueError** – If the context is longer than 255 bytes.

verify_mu(*signature*, *mu*)

Added in version 49.0.0.

Verify a signature over a precomputed `mu` (message representative), the "external mu" variant from FIPS 204. `mu` can be computed incrementally with *MLDSAMuHasher*.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **mu** (*bytes-like*) – The 64-byte message representative.

Returns
None

Raises

- *cryptography.exceptions.InvalidSignature* – Raised when the signature cannot be verified.
- **ValueError** – If mu is not 64 bytes.

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSA87PrivateKey**

Added in version 47.0.0.

classmethod generate()

Generate an ML-DSA-87 private key.

Returns
MLDSA87PrivateKey

Raises
cryptography.exceptions.UnsupportedAlgorithm – If ML-DSA-87 is not supported by the backend cryptography is using.

classmethod from_seed_bytes(*data*)

Load an ML-DSA-87 private key from seed bytes.

Parameters
data (*bytes-like*) – 32 byte seed.

Returns
MLDSA87PrivateKey

Raises

- **ValueError** – If the seed is not 32 bytes.
- *cryptography.exceptions.UnsupportedAlgorithm* – If ML-DSA-87 is not supported by the backend cryptography is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA87PrivateKey.generate()
>>> seed = private_key.private_bytes_raw()
>>> same_key = mldsa.MLDSA87PrivateKey.from_seed_bytes(seed)
```

public_key()

Returns
MLDSA87PublicKey

sign(*data*, *context=None*)

Sign the data using ML-DSA-87. An optional context string can be provided.

Parameters

- **data** (*bytes-like*) – The data to sign.
- **context** (*bytes-like* or None) – An optional context string (up to 255 bytes).

Returns bytes
The signature (4627 bytes).

Raises
`ValueError` – If the context is longer than 255 bytes.

`sign_mu`(*mu*)

Added in version 49.0.0.

Sign a precomputed `mu` (message representative) using ML-DSA-87, the "external mu" variant from FIPS 204. `mu` can be computed incrementally with `MLDSAMuHasher`.

Parameters
mu (*bytes-like*) – The 64-byte message representative.

Returns bytes
The signature (4627 bytes).

Raises
`ValueError` – If `mu` is not 64 bytes.

`private_bytes`(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (`PEM`, `DER`, or `Raw`) and format (`PKCS8` or `Raw`) are chosen to define the exact serialization.

This method only returns the serialization of the seed form of the private key, never the expanded one.

Parameters

- **encoding** – A value from the `Encoding` enum.
- **format** – A value from the `PrivateFormat` enum. If the `encoding` is `Raw` then `format` must be `Raw` , otherwise it must be `PKCS8`.
- **encryption_algorithm** – An instance of an object conforming to the `KeySerializationEncryption` interface.

Return bytes
Serialized key.

`private_bytes_raw`()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling `private_bytes()` with `Raw` encoding, `Raw` format, and `NoEncryption`.

This method only returns the seed form of the private key (32 bytes).

Return bytes
Raw key (32-byte seed).

class `cryptography.hazmat.primitives.asymmetric.mldsa.`**MLDSA87PublicKey**

Added in version 47.0.0.

classmethod **from_public_bytes**(*data*)

Parameters
data (*bytes*) – 2592 byte public key.

Returns
`MLDSA87PublicKey`

Raises

- `ValueError` – If the public key is not 2592 bytes.
- `cryptography.exceptions.UnsupportedAlgorithm` – If ML-DSA-87 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import mldsa
>>> private_key = mldsa.MLDSA87PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = mldsa.MLDSA87PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes

The public key bytes.

public_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes

2592-byte raw public key.

verify(*signature*, *data*, *context=None*)

Verify a signature using ML-DSA-87. If a context string was used during signing, the same context must be provided for verification to succeed.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The data to verify.
- **context** (*bytes-like* or `None`) – An optional context string (up to 255 bytes) that was used during signing.

Returns

None

Raises

- ***cryptography.exceptions.InvalidSignature*** – Raised when the signature cannot be verified.
- **ValueError** – If the context is longer than 255 bytes.

verify_mu(*signature*, *mu*)

Added in version 49.0.0.

Verify a signature over a precomputed `mu` (message representative), the "external mu" variant from FIPS 204. `mu` can be computed incrementally with *MLDSAMuHasher*.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **mu** (*bytes-like*) – The 64-byte message representative.

Returns
None

Raises

- *cryptography.exceptions.InvalidSignature* – Raised when the signature cannot be verified.
- ValueError – If mu is not 64 bytes.

class cryptography.hazmat.primitives.asymmetric.mldsa.**MLDSAMuHasher**(*public_key*, *context=None*)

Added in version 50.0.0.

Incrementally computes the ML-DSA mu (message representative) for the "external mu" signing and verification APIs. The message is supplied incrementally with *update()*, so it need not be buffered in memory.

Parameters

- **public_key** (An ML-DSA public key, e.g. *MLDSA65PublicKey*.) – The ML-DSA public key the mu is bound to.
- **context** (*bytes-like* or None) – An optional context string (up to 255 bytes). It must match the context later passed to *MLDSA65PublicKey.verify()*.

Raises

- TypeError – If public_key is not an ML-DSA public key.
- ValueError – If the context is longer than 255 bytes.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the backend does not support computing mu.

update(*data*)

Parameters
data (*bytes-like*) – The bytes to hash into mu.

Raises
cryptography.exceptions.AlreadyFinalized – If *finalize()* has already been called.

copy()

Copy the current state so that mu can be computed over a common prefix and then diverging suffixes.

Returns
A new *MLDSAMuHasher* with the same internal state.

Raises
cryptography.exceptions.AlreadyFinalized – If *finalize()* has already been called.

finalize()

Finalize the hasher and return the computed mu. After calling this the hasher can no longer be used.

Return bytes
The 64-byte message representative mu.

Raises
cryptography.exceptions.AlreadyFinalized – If *finalize()* has already been called.

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.

ML-KEM key encapsulation

ML-KEM is a post-quantum key encapsulation mechanism based on module lattices, standardized in FIPS 203.

Encapsulation & Decapsulation

```
>>> from cryptography.hazmat.primitives.asymmetric.mlkem import MLKEM768PrivateKey
>>> private_key = MLKEM768PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> shared_secret, ciphertext = public_key.encapsulate()
>>> recovered_secret = private_key.decapsulate(ciphertext)
>>> shared_secret == recovered_secret
True
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.mlkem.**MLKEM768PrivateKey**

Added in version 47.0.0.

classmethod generate()

Generate an ML-KEM-768 private key.

Returns
MLKEM768PrivateKey

Raises
cryptography.exceptions.UnsupportedAlgorithm – If ML-KEM-768 is not supported by the backend cryptography is using.

classmethod from_seed_bytes(*data*)

Load an ML-KEM-768 private key from seed bytes.

Parameters
data (*bytes-like*) – 64 byte seed.

Returns
MLKEM768PrivateKey

Raises

- **ValueError** – If the seed is not 64 bytes.
- *cryptography.exceptions.UnsupportedAlgorithm* – If ML-KEM-768 is not supported by the backend cryptography is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import mlkem
>>> private_key = mlkem.MLKEM768PrivateKey.generate()
>>> seed = private_key.private_bytes_raw()
>>> same_key = mlkem.MLKEM768PrivateKey.from_seed_bytes(seed)
```

public_key()

Returns
MLKEM768PublicKey

decapsulate(*ciphertext*)

Decapsulate a ciphertext using ML-KEM-768, returning the shared secret.

Parameters
ciphertext (*bytes-like*) – The ciphertext to decapsulate (1088 bytes).

Returns bytes
The shared secret (32 bytes).

Raises
ValueError – If the ciphertext is not the correct length.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

This method only returns the serialization of the seed form of the private key, never the expanded one.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

This method only returns the seed form of the private key (64 bytes).

Return bytes
Raw key (64-byte seed).

class cryptography.hazmat.primitives.asymmetric.mlkem.**MLKEM768PublicKey**

Added in version 47.0.0.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 1184 byte public key.

Returns
MLKEM768PublicKey

Raises

- **ValueError** – If the public key is not 1184 bytes.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If ML-KEM-768 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import mlkem
>>> private_key = mlkem.MLKEM768PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = mlkem.MLKEM768PublicKey.from_public_bytes(public_bytes)
```

encapsulate()

Generate a shared secret and encapsulate it for this public key.

Returns
A `(shared_secret, ciphertext)` tuple where both values are `bytes`. The shared secret is 32 bytes and the ciphertext is 1088 bytes.

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
1184-byte raw public key.

class cryptography.hazmat.primitives.asymmetric.mlkem.**MLKEM1024PrivateKey**

Added in version 47.0.0.

classmethod generate()

Generate an ML-KEM-1024 private key.

Returns
MLKEM1024PrivateKey

Raises
cryptography.exceptions.UnsupportedAlgorithm – If ML-KEM-1024 is not supported by the backend `cryptography` is using.

classmethod from_seed_bytes(*data*)

Load an ML-KEM-1024 private key from seed bytes.

Parameters
data (*bytes-like*) – 64 byte seed.

Returns
MLKEM1024PrivateKey

Raises

- **ValueError** – If the seed is not 64 bytes.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If ML-KEM-1024 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import mlkem
>>> private_key = mlkem.MLKEM1024PrivateKey.generate()
>>> seed = private_key.private_bytes_raw()
>>> same_key = mlkem.MLKEM1024PrivateKey.from_seed_bytes(seed)
```

public_key()

Returns
MLKEM1024PublicKey

decapsulate(*ciphertext*)

Decapsulate a ciphertext using ML-KEM-1024, returning the shared secret.

Parameters
ciphertext (*bytes-like*) – The ciphertext to decapsulate (1568 bytes).

Returns bytes
The shared secret (32 bytes).

Raises
ValueError – If the ciphertext is not the correct length.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

This method only returns the serialization of the seed form of the private key, never the expanded one.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

This method only returns the seed form of the private key (64 bytes).

Return bytes
Raw key (64-byte seed).

class cryptography.hazmat.primitives.asymmetric.mlkem.**MLKEM1024PublicKey**

Added in version 47.0.0.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 1568 byte public key.

Returns
MLKEM1024PublicKey

Raises

- **ValueError** – If the public key is not 1568 bytes.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If ML-KEM-1024 is not supported by the backend `cryptography` is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import mlkem
>>> private_key = mlkem.MLKEM1024PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = mlkem.MLKEM1024PublicKey.from_public_bytes(public_bytes)
```

encapsulate()

Generate a shared secret and encapsulate it for this public key.

Returns
A `(shared_secret, ciphertext)` tuple where both values are `bytes`. The shared secret is 32 bytes and the ciphertext is 1568 bytes.

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
1568-byte raw public key.

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.

Ed25519 signing

Ed25519 is an elliptic curve signing algorithm using EdDSA and Curve25519. If you do not have legacy interoperability concerns then you should strongly consider using this signature algorithm.

Signing & Verification

```
>>> from cryptography.hazmat.primitives.asymmetric.ed25519 import Ed25519PrivateKey
>>> private_key = Ed25519PrivateKey.generate()
>>> signature = private_key.sign(b"my authenticated message")
>>> public_key = private_key.public_key()
>>> # Raises InvalidSignature if verification fails
>>> public_key.verify(signature, b"my authenticated message")
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.ed25519.**Ed25519PrivateKey**

Added in version 2.6.

classmethod generate()

Generate an Ed25519 private key.

Returns
Ed25519PrivateKey

classmethod from_private_bytes(*data*)

Parameters
data (*bytes-like*) – 32 byte private key.

Returns
Ed25519PrivateKey

Raises

- **ValueError** – This is raised if the private key is not 32 bytes long.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If Ed25519 is not supported by the OpenSSL version cryptography is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import ed25519
>>> private_key = ed25519.Ed25519PrivateKey.generate()
>>> private_bytes = private_key.private_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PrivateFormat.Raw,
...     encryption_algorithm=serialization.NoEncryption()
... )
>>> loaded_private_key = ed25519.Ed25519PrivateKey.from_private_bytes(private_
↪bytes)
```

public_key()

Returns
Ed25519PublicKey

sign(*data*)

Parameters
data (*bytes-like*) – The data to sign.

Returns bytes
The 64 byte signature.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8*, *OpenSSH* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *PKCS8* or *OpenSSH*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

Return bytes
Raw key.

class cryptography.hazmat.primitives.asymmetric.ed25519.**Ed25519PublicKey**

Added in version 2.6.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 32 byte public key.

Returns
Ed25519PublicKey

Raises

- **ValueError** – This is raised if the public key is not 32 bytes long.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If Ed25519 is not supported by the OpenSSL version `cryptography` is using.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import ed25519
>>> private_key = ed25519.Ed25519PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = ed25519.Ed25519PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, *OpenSSH*, or *Raw*) and format (*SubjectPublicKeyInfo*, *OpenSSH* , or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw*. If `encoding` is *OpenSSH* then `format` must be *OpenSSH*. In all other cases `format` must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
Raw key.

verify(*signature*, *data*)

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The data to verify.

Returns
None

Raises
cryptography.exceptions.InvalidSignature – Raised when the signature cannot be verified.

> 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.

X25519 key exchange

X25519 is an elliptic curve Diffie-Hellman key exchange using Curve25519. It allows two parties to jointly agree on a shared secret using an insecure channel.

Exchange Algorithm

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric.x25519 import X25519PrivateKey
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate a private key for use in the exchange.
```

(continues on next page)

(continued from previous page)

```
>>> private_key = X25519PrivateKey.generate()
>>> # In a real handshake the peer_public_key will be received from the
>>> # other party. For this example we'll generate another private key and
>>> # get a public key from that. Note that in a DH handshake both peers
>>> # must agree on a common set of parameters.
>>> peer_public_key = X25519PrivateKey.generate().public_key()
>>> shared_key = private_key.exchange(peer_public_key)
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
>>> # For the next handshake we MUST generate another private key.
>>> private_key_2 = X25519PrivateKey.generate()
>>> peer_public_key_2 = X25519PrivateKey.generate().public_key()
>>> shared_key_2 = private_key_2.exchange(peer_public_key_2)
>>> derived_key_2 = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key_2)
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.x25519.**X25519PrivateKey**

Added in version 2.0.

classmethod generate()

Generate an X25519 private key.

Returns
X25519PrivateKey

classmethod from_private_bytes(*data*)

Added in version 2.5.

A class method for loading an X25519 key encoded as *Raw*.

Parameters
data (*bytes*) – 32 byte private key.

Returns
X25519PrivateKey

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import x25519
>>> private_key = x25519.X25519PrivateKey.generate()
>>> private_bytes = private_key.private_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PrivateFormat.Raw,
...     encryption_algorithm=serialization.NoEncryption()
```

(continues on next page)

(continued from previous page)

```
... )
>>> loaded_private_key = x25519.X25519PrivateKey.from_private_bytes(private_
↪bytes)
```

public_key()

Returns
X25519PublicKey

exchange(*peer_public_key*)

Parameters
peer_public_key (X25519PublicKey) – The public key for the peer.

Returns bytes
A shared key.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Added in version 2.5.

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the encoding is *Raw* then format must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

Return bytes
Raw key.

class cryptography.hazmat.primitives.asymmetric.x25519.**X25519PublicKey**

Added in version 2.0.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 32 byte public key.

Returns
X25519PublicKey

```
>>> from cryptography.hazmat.primitives.asymmetric import x25519
>>> private_key = x25519.X25519PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
```

(continues on next page)

(continued from previous page)

```
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = x25519.X25519PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
Raw key.

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.

Ed448 signing

Ed448 is an elliptic curve signing algorithm using EdDSA.

Signing & Verification

```
>>> from cryptography.hazmat.primitives.asymmetric.ed448 import Ed448PrivateKey
>>> private_key = Ed448PrivateKey.generate()
>>> signature = private_key.sign(b"my authenticated message")
>>> public_key = private_key.public_key()
>>> # Raises InvalidSignature if verification fails
>>> public_key.verify(signature, b"my authenticated message")
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.ed448.**Ed448PrivateKey**

Added in version 2.6.

classmethod generate()

Generate an Ed448 private key.

Returns
Ed448PrivateKey

classmethod from_private_bytes(*data*)

Parameters
data (*bytes-like*) – 57 byte private key.

Returns
Ed448PrivateKey

public_key()

Returns
Ed448PublicKey

sign(*data*)

Parameters
data (*bytes-like*) – The data to sign.

Returns bytes
The 114 byte signature.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the encoding is *Raw* then format must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

Return bytes
Raw key.

class cryptography.hazmat.primitives.asymmetric.ed448.**Ed448PublicKey**

Added in version 2.6.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 57 byte public key.

Returns
Ed448PublicKey

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
Raw key.

verify(*signature*, *data*)

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The data to verify.

Returns
None

Raises
cryptography.exceptions.InvalidSignature – Raised when the signature cannot be verified.

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.

X448 key exchange

X448 is an elliptic curve Diffie-Hellman key exchange using Curve448. It allows two parties to jointly agree on a shared secret using an insecure channel.

Exchange Algorithm

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric.x448 import X448PrivateKey
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate a private key for use in the exchange.
>>> private_key = X448PrivateKey.generate()
```

(continues on next page)

(continued from previous page)

```
>>> # In a real handshake the peer_public_key will be received from the
>>> # other party. For this example we'll generate another private key and
>>> # get a public key from that. Note that in a DH handshake both peers
>>> # must agree on a common set of parameters.
>>> peer_public_key = X448PrivateKey.generate().public_key()
>>> shared_key = private_key.exchange(peer_public_key)
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
>>> # For the next handshake we MUST generate another private key.
>>> private_key_2 = X448PrivateKey.generate()
>>> peer_public_key_2 = X448PrivateKey.generate().public_key()
>>> shared_key_2 = private_key_2.exchange(peer_public_key_2)
>>> derived_key_2 = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key_2)
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.x448.**X448PrivateKey**

Added in version 2.5.

classmethod generate()

Generate an X448 private key.

Returns
X448PrivateKey

classmethod from_private_bytes(*data*)

Parameters
data (*bytes-like*) – 56 byte private key.

Returns
X448PrivateKey

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import x448
>>> private_key = x448.X448PrivateKey.generate()
>>> private_bytes = private_key.private_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PrivateFormat.Raw,
...     encryption_algorithm=serialization.NoEncryption()
... )
>>> loaded_private_key = x448.X448PrivateKey.from_private_bytes(private_bytes)
```

public_key()

Returns
X448PublicKey

exchange(*peer_public_key*)

Parameters
peer_public_key (X448PublicKey) – The public key for the peer.

Returns bytes
A shared key.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*PKCS8* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *PKCS8*.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

private_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *private_bytes()* with *Raw* encoding, *Raw* format, and *NoEncryption*.

Return bytes
Raw key.

class cryptography.hazmat.primitives.asymmetric.x448.**X448PublicKey**

Added in version 2.5.

classmethod from_public_bytes(*data*)

Parameters
data (*bytes*) – 56 byte public key.

Returns
X448PublicKey

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import x448
>>> private_key = x448.X448PrivateKey.generate()
>>> public_key = private_key.public_key()
>>> public_bytes = public_key.public_bytes(
...     encoding=serialization.Encoding.Raw,
...     format=serialization.PublicFormat.Raw
... )
>>> loaded_public_key = x448.X448PublicKey.from_public_bytes(public_bytes)
```

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM*, *DER*, or *Raw*) and format (*SubjectPublicKeyInfo* or *Raw*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum. If the `encoding` is *Raw* then `format` must be *Raw* , otherwise it must be *SubjectPublicKeyInfo*.

Returns bytes
The public key bytes.

public_bytes_raw()

Added in version 40.

Allows serialization of the key to raw bytes. This method is a convenience shortcut for calling *public_bytes()* with *Raw* encoding and *Raw* format.

Return bytes
Raw key.

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.

Elliptic curve cryptography

`cryptography.hazmat.primitives.asymmetric.ec.`**generate_private_key**(*curve*)

Added in version 0.5.

Generate a new private key on `curve`.

Parameters
curve – An instance of *EllipticCurve*.

Returns
A new instance of *EllipticCurvePrivateKey*.

`cryptography.hazmat.primitives.asymmetric.ec.`**derive_private_key**(*private_value*, *curve*)

Added in version 1.6.

Derive a private key from `private_value` on `curve`.

Parameters

- **private_value** (*int*) – The secret scalar value.
- **curve** – An instance of *EllipticCurve*.

Returns
A new instance of *EllipticCurvePrivateKey*.

Elliptic Curve Signature Algorithms

class `cryptography.hazmat.primitives.asymmetric.ec.`**ECDSA**(*algorithm*)

Added in version 0.5.

The ECDSA signature algorithm first standardized in NIST publication FIPS 186-3, and later in FIPS 186-4.

Note that while elliptic curve keys can be used for both signing and key exchange, this is bad cryptographic practice. Instead, users should generate separate signing and ECDH keys.

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **deterministic_signing** (*bool*) – A boolean flag defaulting to `False` that specifies whether the signing procedure should be deterministic or not, as defined in **RFC 6979**. This only impacts the signing process, verification is not affected (the verification process is the same for both deterministic and non-deterministic signed messages).

  Added in version 43.0.0.

Raises

cryptography.exceptions.UnsupportedAlgorithm – If `deterministic_signing` is set to `True` and the version of OpenSSL does not support ECDSA with deterministic signing.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> private_key = ec.generate_private_key(
...     ec.SECP384R1()
... )
>>> data = b"this is some data I'd like to sign"
>>> signature = private_key.sign(
...     data,
...     ec.ECDSA(hashes.SHA256())
... )
```

The `signature` is a `bytes` object, whose contents are DER encoded as described in **RFC 3279**. This can be decoded using *decode_dss_signature()*.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
...     ec.ECDSA(utils.Prehashed(chosen_hash))
... )
```

Verification requires the public key, the DER-encoded signature itself, the signed data, and knowledge of the hashing algorithm that was used when producing the signature:

```
>>> public_key = private_key.public_key()
>>> public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```

As above, the `signature` is a `bytes` object whose contents are DER encoded as described in **RFC 3279**. It can be created from a raw `(r,s)` pair by using *encode_dss_signature()*.

If the signature is not valid, an *InvalidSignature* exception will be raised.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> public_key.verify(
...     sig,
...     digest,
...     ec.ECDSA(utils.Prehashed(chosen_hash))
... )
```

Note

Although in this case the public key was derived from the private one, in a typical setting you will not possess the private key. The *Key loading* section explains how to load the public key from other sources.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePrivateNumbers**(*private_value*, *public_numbers*)

Added in version 0.5.

The collection of integers that make up an EC private key.

public_numbers

Type
EllipticCurvePublicNumbers

The *EllipticCurvePublicNumbers* which makes up the EC public key associated with this EC private key.

private_value

Type
int

The private value.

private_key()

Convert a collection of numbers into a private key suitable for doing actual cryptographic operations.

Returns
A new instance of *EllipticCurvePrivateKey*.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePublicNumbers**(*x*, *y*, *curve*)

Warning

The point represented by this object is not validated in any way until *EllipticCurvePublicNumbers.public_key()* is called and may not represent a valid point on the curve. You should not attempt to perform any computations using the values from this class until you have either validated it yourself or called public_key() successfully.

Added in version 0.5.

The collection of integers that make up an EC public key.

curve

Type
EllipticCurve

The elliptic curve for this key.

x

Type
int

The affine x component of the public point used for verifying.

y

Type
int

The affine y component of the public point used for verifying.

public_key()

Convert a collection of numbers into a public key suitable for doing actual cryptographic operations.

Raises
ValueError – Raised if the point is invalid for the curve.

Returns
A new instance of *EllipticCurvePublicKey*.

Elliptic Curve Key Exchange algorithm

class cryptography.hazmat.primitives.asymmetric.ec.**ECDH**

Added in version 1.1.

The Elliptic Curve Diffie-Hellman Key Exchange algorithm standardized in NIST publication 800-56A.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Note that while elliptic curve keys can be used for both signing and key exchange, this is bad cryptographic practice. Instead, users should generate separate signing and ECDH keys.

Warning

This example does not give forward secrecy and is only provided as a demonstration of the basic Diffie-Hellman construction. For real world applications always use the ephemeral form described after this example.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate a private key for use in the exchange.
>>> server_private_key = ec.generate_private_key(
...     ec.SECP384R1()
... )
>>> # In a real handshake the peer is a remote client. For this
```

(continues on next page)

(continued from previous page)

```
>>> # example we'll generate another local private key though.
>>> peer_private_key = ec.generate_private_key(
...     ec.SECP384R1()
... )
>>> shared_key = server_private_key.exchange(
...     ec.ECDH(), peer_private_key.public_key())
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
>>> # And now we can demonstrate that the handshake performed in the
>>> # opposite direction gives the same final value
>>> same_shared_key = peer_private_key.exchange(
...     ec.ECDH(), server_private_key.public_key())
>>> # Perform key derivation.
>>> same_derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(same_shared_key)
>>> derived_key == same_derived_key
True
```

ECDHE (or EECDH), the ephemeral form of this exchange, is **strongly preferred** over simple ECDH and provides forward secrecy when used. You must generate a new private key using *generate_private_key()* for each *exchange()* when performing an ECDHE key exchange. An example of the ephemeral form:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate a private key for use in the exchange.
>>> private_key = ec.generate_private_key(
...     ec.SECP384R1()
... )
>>> # In a real handshake the peer_public_key will be received from the
>>> # other party. For this example we'll generate another private key
>>> # and get a public key from that.
>>> peer_public_key = ec.generate_private_key(
...     ec.SECP384R1()
... ).public_key()
>>> shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
```

(continues on next page)

(continued from previous page)

```
>>> # For the next handshake we MUST generate another private key.
>>> private_key_2 = ec.generate_private_key(
...     ec.SECP384R1()
... )
>>> peer_public_key_2 = ec.generate_private_key(
...     ec.SECP384R1()
... ).public_key()
>>> shared_key_2 = private_key_2.exchange(ec.ECDH(), peer_public_key_2)
>>> derived_key_2 = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key_2)
```

Elliptic Curves

Elliptic curves provide equivalent security at much smaller key sizes than other asymmetric cryptography systems such as RSA or DSA. For many operations elliptic curves are also significantly faster; elliptic curve diffie-hellman is faster than diffie-hellman.

> **Note**
>
> Curves with a size of less than 224 bits should not be used. You should strongly consider using curves of at least 224 *bits*.

Generally the NIST prime field ("P") curves are significantly faster than the other types suggested by NIST at both signing and verifying with ECDSA.

Prime fields also minimize the number of security concerns for elliptic-curve cryptography. However, there is some concern that both the prime field and binary field ("B") NIST curves may have been weakened during their generation.

Currently *cryptography* only supports NIST curves, none of which are considered "safe" by the SafeCurves project run by Daniel J. Bernstein and Tanja Lange.

All named curves are instances of `EllipticCurve`.

class `cryptography.hazmat.primitives.asymmetric.ec.`**`SECP256R1`**

Added in version 0.5.

SECG curve `secp256r1`. Also called NIST P-256.

class `cryptography.hazmat.primitives.asymmetric.ec.`**`SECP384R1`**

Added in version 0.5.

SECG curve `secp384r1`. Also called NIST P-384.

class `cryptography.hazmat.primitives.asymmetric.ec.`**`SECP521R1`**

Added in version 0.5.

SECG curve `secp521r1`. Also called NIST P-521.

class `cryptography.hazmat.primitives.asymmetric.ec.`**`SECP224R1`**

Added in version 0.5.

SECG curve `secp224r1`. Also called NIST P-224.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP192R1**

Added in version 0.5.

SECG curve `secp192r1`. Also called NIST P-192.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP256K1**

Added in version 0.9.

SECG curve `secp256k1`.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP256R1**

Added in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP384R1**

Added in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP512R1**

Added in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

Key Interfaces

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurve**

Added in version 0.5.

A named elliptic curve.

name

Type
str

The name of the curve. Usually the name used for the ASN.1 OID such as `secp256k1`.

key_size

Type
int

Size (in *bits*) of a secret scalar for the curve (as generated by *`generate_private_key()`*).

group_order

Added in version 45.

Type
int

The order of the curve's group.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurveSignatureAlgorithm**

Added in version 0.5.

Changed in version 1.6: *`Prehashed`* can now be used as an `algorithm`.

A signature algorithm for use with elliptic curve keys.

algorithm

Type
HashAlgorithm or *Prehashed*

The digest algorithm to be used with the signature scheme.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePrivateKey**

Added in version 0.5.

An elliptic curve private key for use with an algorithm such as ECDSA.

exchange(*algorithm*, *peer_public_key*)

Added in version 1.1.

Performs a key exchange operation using the provided algorithm with the peer's public key.

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Parameters

- **algorithm** – The key exchange algorithm, currently only *ECDH* is supported.
- **peer_public_key** (*EllipticCurvePublicKey*) – The public key for the peer.

Returns bytes
A shared key.

public_key()

Returns
EllipticCurvePublicKey

The EllipticCurvePublicKey object for this private key.

sign(*data*, *signature_algorithm*)

Added in version 1.5.

Sign one block of data which can be verified later by others using the public key.

Parameters

- **data** (*bytes-like*) – The message string to sign.
- **signature_algorithm** – An instance of *EllipticCurveSignatureAlgorithm*, such as *ECDSA*.

Return bytes
The signature as a `bytes` object, whose contents are DER encoded as described in **RFC 3279**. This can be decoded using *decode_dss_signature()*, which returns the decoded tuple `(r, s)`.

curve

Type
EllipticCurve

The EllipticCurve that this key is on.

key_size

Added in version 1.9.

Type
int

Size (in *bits*) of a secret scalar for the curve (as generated by `generate_private_key()`).

private_numbers()

Create a `EllipticCurvePrivateNumbers` object.

Returns
An `EllipticCurvePrivateNumbers` instance.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (`PEM` or `DER`), format (`TraditionalOpenSSL`, `OpenSSH` or `PKCS8`) and encryption algorithm (such as `BestAvailableEncryption` or `NoEncryption`) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the `Encoding` enum.
- **format** – A value from the `PrivateFormat` enum.
- **encryption_algorithm** – An instance of an object conforming to the `KeySerializationEncryption` interface.

Return bytes
Serialized key.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePublicKey**

Added in version 0.5.

An elliptic curve public key.

curve

Type
`EllipticCurve`

The elliptic curve for this key.

public_numbers()

Create a `EllipticCurvePublicNumbers` object.

Returns
An `EllipticCurvePublicNumbers` instance.

public_bytes(*encoding*, *format*)

Allows serialization of the key data to bytes. When encoding the public key the encodings (`PEM`, `DER`) and format (`SubjectPublicKeyInfo`) are chosen to define the exact serialization. When encoding the point the encoding `X962` should be used with the formats (`UncompressedPoint` or `CompressedPoint`).

Parameters

- **encoding** – A value from the `Encoding` enum.
- **format** – A value from the `PublicFormat` enum.

Return bytes
Serialized data.

verify(*signature*, *data*, *signature_algorithm*)

Added in version 1.5.

Verify one block of data was signed by the private key associated with this public key.

Parameters

- **signature** (*bytes-like*) – The DER-encoded signature to verify. A raw signature may be DER-encoded by splitting it into the `r` and `s` components and passing them into `encode_dss_signature()`.
- **data** (*bytes-like*) – The message string that was signed.
- **signature_algorithm** – An instance of `EllipticCurveSignatureAlgorithm`.

Returns
None

Raises
`cryptography.exceptions.InvalidSignature` – If the signature does not validate.

key_size

Added in version 1.9.

Type
int

Size (in *bits*) of a secret scalar for the curve (as generated by `generate_private_key()`).

classmethod from_encoded_point(*curve*, *data*)

Added in version 2.5.

Decodes a byte string as described in SEC 1 v2.0 section 2.3.3 and returns an `EllipticCurvePublicKey`. This class method supports compressed points.

Parameters

- **curve** – An `EllipticCurve` instance.
- **data** (*bytes*) – The serialized point byte string.

Returns
An `EllipticCurvePublicKey` instance.

Raises

- `ValueError` – Raised when an invalid point is supplied.
- `TypeError` – Raised when curve is not an `EllipticCurve`.

Serialization

This sample demonstrates how to generate a private key and serialize it.

```
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.asymmetric import ec

>>> private_key = ec.generate_private_key(ec.SECP384R1())

>>> serialized_private = private_key.private_bytes(
...     encoding=serialization.Encoding.PEM,
...     format=serialization.PrivateFormat.PKCS8,
...     encryption_algorithm=serialization.BestAvailableEncryption(b'testpassword')
... )
>>> serialized_private.splitlines()[0]
b'-----BEGIN ENCRYPTED PRIVATE KEY-----'
```

You can also serialize the key without a password, by relying on *NoEncryption*.

The public key is serialized as follows:

```
>>> public_key = private_key.public_key()
>>> serialized_public = public_key.public_bytes(
...     encoding=serialization.Encoding.PEM,
...     format=serialization.PublicFormat.SubjectPublicKeyInfo
... )
>>> serialized_public.splitlines()[0]
b'-----BEGIN PUBLIC KEY-----'
```

This is the part that you would normally share with the rest of the world.

Key loading

This extends the sample in the previous section, assuming that the variables `serialized_private` and `serialized_public` contain the respective keys in PEM format.

```
>>> loaded_public_key = serialization.load_pem_public_key(
...     serialized_public,
... )

>>> loaded_private_key = serialization.load_pem_private_key(
...     serialized_private,
...     # or password=None, if in plain text
...     password=b'testpassword',
... )
```

Elliptic Curve Object Identifiers

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurveOID**

Added in version 2.4.

SECP192R1

Corresponds to the dotted string `"1.2.840.10045.3.1.1"`.

SECP224R1

Corresponds to the dotted string `"1.3.132.0.33"`.

SECP256K1

Corresponds to the dotted string `"1.3.132.0.10"`.

SECP256R1

Corresponds to the dotted string `"1.2.840.10045.3.1.7"`.

SECP384R1

Corresponds to the dotted string `"1.3.132.0.34"`.

SECP521R1

Corresponds to the dotted string `"1.3.132.0.35"`.

BRAINPOOLP256R1

Added in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.7"`.

BRAINPOOLP384R1

Added in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.11"`.

BRAINPOOLP512R1

Added in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.13"`.

`cryptography.hazmat.primitives.asymmetric.ec.`**`get_curve_for_oid`**(*oid*)

Added in version 2.6.

A function that takes an `ObjectIdentifier` and returns the associated elliptic curve class.

Parameters

oid – An instance of `ObjectIdentifier`.

Returns

The matching elliptic curve class. The returned class conforms to the `EllipticCurve` interface.

Raises

`LookupError` – Raised if no elliptic curve is found that matches the provided object identifier.

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.

RSA

RSA is a public-key algorithm for encrypting and signing messages.

Generation

Unlike symmetric cryptography, where the key is typically just a random series of bytes, RSA keys have a complex internal structure with specific mathematical properties.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`generate_private_key`**(*public_exponent*, *key_size*)

Added in version 0.5.

Changed in version 3.0: Tightened restrictions on `public_exponent`.

Generates a new RSA private key. `key_size` describes how many *bits* long the key should be. Larger keys provide more security; currently `1024` and below are considered breakable while `2048` or `4096` are reasonable default key sizes for new keys. The `public_exponent` indicates what one mathematical property of the key generation will be. Unless you have a specific reason to do otherwise, you should always use 65537.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
```

Parameters

- **public_exponent** (*int*) – The public exponent of the new key. Either 65537 or 3 (for legacy purposes). Almost everyone should use 65537.

- **key_size** (*int*) – The length of the modulus in *bits*. For keys generated in 2015 it is strongly recommended to be at least 2048 (See page 41). It must not be less than 512.

Returns
An instance of *RSAPrivateKey*.

Key loading

If you already have an on-disk key in the PEM format (which are recognizable by the distinctive `-----BEGIN {format}-----` and `-----END {format}-----` markers), you can load it:

```
>>> from cryptography.hazmat.primitives import serialization

>>> with open("path/to/key.pem", "rb") as key_file:
...     private_key = serialization.load_pem_private_key(
...         key_file.read(),
...         password=None,
...     )
```

Serialized keys may optionally be encrypted on disk using a password. In this example we loaded an unencrypted key, and therefore we did not provide a password. If the key is encrypted we can pass a `bytes` object as the `password` argument.

There is also support for *loading public keys in the SSH format*.

Key serialization

If you have a private key that you've loaded you can use *private_bytes()* to serialize the key.

```
>>> from cryptography.hazmat.primitives import serialization
>>> pem = private_key.private_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PrivateFormat.PKCS8,
...    encryption_algorithm=serialization.BestAvailableEncryption(b'mypassword')
... )
>>> pem.splitlines()[0]
b'-----BEGIN ENCRYPTED PRIVATE KEY-----'
```

It is also possible to serialize without encryption using *NoEncryption*.

```
>>> pem = private_key.private_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PrivateFormat.TraditionalOpenSSL,
...    encryption_algorithm=serialization.NoEncryption()
... )
>>> pem.splitlines()[0]
b'-----BEGIN RSA PRIVATE KEY-----'
```

For public keys you can use *public_bytes()* to serialize the key.

```
>>> from cryptography.hazmat.primitives import serialization
>>> public_key = private_key.public_key()
>>> pem = public_key.public_bytes(
...    encoding=serialization.Encoding.PEM,
...    format=serialization.PublicFormat.SubjectPublicKeyInfo
```

(continues on next page)

(continued from previous page)

```
... )
>>> pem.splitlines()[0]
b'-----BEGIN PUBLIC KEY-----'
```

Signing

A private key can be used to sign a message. This allows anyone with the public key to verify that the message was created by someone who possesses the corresponding private key. RSA signatures require a specific hash function, and padding to be used. Here is an example of signing `message` using RSA, with a secure hash function and padding:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> message = b"A message I want to sign"
>>> signature = private_key.sign(
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

Valid paddings for signatures are *PSS* and *PKCS1v15*. `PSS` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(chosen_hash)
... )
```

Verification

The previous section describes what to do if you have a private key and want to sign something. If you have a public key, a message, a signature, and the signing algorithm that was used you can check that the private key associated with a given public key was used to sign that specific message. You can obtain a public key to use in verification using *load_pem_public_key()*, *load_der_public_key()*, *public_key()* , or *public_key()*.

```
>>> public_key = private_key.public_key()
>>> public_key.verify(
...     signature,
```

(continues on next page)

(continued from previous page)

```
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

If the signature does not match, `verify()` will raise an *InvalidSignature* exception.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> public_key.verify(
...     sig,
...     digest,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(chosen_hash)
... )
```

Encryption

RSA encryption is interesting because encryption is performed using the **public** key, meaning anyone can encrypt data. The data is then decrypted using the **private** key.

Like signatures, RSA supports encryption with several different padding options. Here's an example using a secure padding and hash function:

```
>>> message = b"encrypted data"
>>> ciphertext = public_key.encrypt(
...     message,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
```

Valid paddings for encryption are *OAEP* and *PKCS1v15*. `OAEP` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

Decryption

Once you have an encrypted message, it can be decrypted using the private key:

```
>>> plaintext = private_key.decrypt(
...     ciphertext,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
>>> plaintext == message
True
```

Padding

class cryptography.hazmat.primitives.asymmetric.padding.**AsymmetricPadding**

Added in version 0.2.

name

class cryptography.hazmat.primitives.asymmetric.padding.**PSS**(*mgf*, *salt_length*)

Added in version 0.3.

Changed in version 0.4: Added `salt_length` parameter.

PSS (Probabilistic Signature Scheme) is a signature scheme defined in **RFC 3447**. It is more complex than PKCS1 but possesses a security proof. This is the recommended padding algorithm for RSA signatures. It cannot be used with RSA encryption.

Parameters

- **mgf** – A mask generation function object. At this time the only supported MGF is *MGF1*.
- **salt_length** (*int*) – The length of the salt. It is recommended that this be set to `PSS.DIGEST_LENGTH` or `PSS.MAX_LENGTH`.

MAX_LENGTH

Pass this attribute to `salt_length` to get the maximum salt length available.

DIGEST_LENGTH

Added in version 37.0.0.

Pass this attribute to `salt_length` to set the salt length to the byte length of the digest passed when calling `sign`. Note that this is **not** the length of the digest passed to `MGF1`.

AUTO

Added in version 37.0.0.

Pass this attribute to `salt_length` to automatically determine the salt length when verifying. Raises `ValueError` if used when signing.

mgf

Type
MGF

Added in version 42.0.0.

The padding's mask generation function (MGF).

class `cryptography.hazmat.primitives.asymmetric.padding.`**`OAEP`**(*mgf*, *algorithm*, *label*)

Added in version 0.4.

OAEP (Optimal Asymmetric Encryption Padding) is a padding scheme defined in **RFC 3447**. It provides probabilistic encryption and is proven secure against several attack types. This is the recommended padding algorithm for RSA encryption. It cannot be used with RSA signing.

Parameters

- **mgf** – A mask generation function object. At this time the only supported MGF is *MGF1*.
- **algorithm** – An instance of *HashAlgorithm*.
- **label** (*bytes*) – A label to apply. This is a rarely used field and should typically be set to `None` or `b""`, which are equivalent.

algorithm

Type
HashAlgorithm

Added in version 42.0.0.

The padding's hash algorithm.

mgf

Type
MGF

Added in version 42.0.0.

The padding's mask generation function (MGF).

class `cryptography.hazmat.primitives.asymmetric.padding.`**`PKCS1v15`**

Added in version 0.3.

PKCS1 v1.5 (also known as simply PKCS1) is a simple padding scheme developed for use with RSA keys. It is defined in **RFC 3447**. This padding can be used for signing and encryption.

It is not recommended that `PKCS1v15` be used for new applications, *OAEP* should be preferred for encryption and *PSS* should be preferred for signatures.

> **Warning**
>
> When used with OpenSSL 3.2 or older, our implementation of PKCS1 v1.5 decryption is not constant time. See *Known security limitations* for details.

`cryptography.hazmat.primitives.asymmetric.padding.`**`calculate_max_pss_salt_length`**(*key*, *hash_algorithm*)

Added in version 1.5.

Parameters

- **key** – An RSA public or private key.
- **hash_algorithm** – A *cryptography.hazmat.primitives.hashes.HashAlgorithm*.

Returns int
The computed salt length.

Computes the length of the salt that *PSS* will use if *PSS.MAX_LENGTH* is used.

Mask generation functions

class cryptography.hazmat.primitives.asymmetric.padding.**MGF**

Added in version 37.0.0.

class cryptography.hazmat.primitives.asymmetric.padding.**MGF1**(*algorithm*)

Added in version 0.3.

Changed in version 0.6: Removed the deprecated `salt_length` parameter.

MGF1 (Mask Generation Function 1) is used as the mask generation function in *PSS* and *OAEP* padding. It takes a hash algorithm.

Parameters
algorithm – An instance of *HashAlgorithm*.

Numbers

These classes hold the constituent components of an RSA key. They are useful only when more traditional *Key Serialization* is unavailable.

class cryptography.hazmat.primitives.asymmetric.rsa.**RSAPublicNumbers**(*e*, *n*)

Added in version 0.5.

The collection of integers that make up an RSA public key.

n

Type
int

The public modulus.

e

Type
int

The public exponent.

public_key()

Returns
A new instance of *RSAPublicKey*.

class cryptography.hazmat.primitives.asymmetric.rsa.**RSAPrivateNumbers**(*p*, *q*, *d*, *dmp1*, *dmq1*, *iqmp*, *public_numbers*)

Added in version 0.5.

The collection of integers that make up an RSA private key.

> **Warning**
>
> With the exception of the integers contained in the *RSAPublicNumbers* all attributes of this class must be kept secret. Revealing them will compromise the security of any cryptographic operations performed with a key loaded from them.

public_numbers

Type
RSAPublicNumbers

The *RSAPublicNumbers* which makes up the RSA public key associated with this RSA private key.

p

Type
int

p, one of the two primes composing n.

q

Type
int

q, one of the two primes composing n.

d

Type
int

The private exponent.

dmp1

Type
int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: d mod (p-1)

dmq1

Type
int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: d mod (q-1)

iqmp

Type
int

A Chinese remainder theorem coefficient used to speed up RSA operations. Calculated as: q^{-1} mod p

private_key(*, *unsafe_skip_rsa_key_validation=False*)

Parameters
unsafe_skip_rsa_key_validation (*bool*) – Added in version 39.0.0.

A keyword-only argument that defaults to False. If True RSA private keys will not be validated. This significantly speeds up loading the keys, but is *unsafe* unless you are certain the key is valid. User supplied keys should never be loaded with this parameter set to True. If you do load an invalid key this way and attempt to use it OpenSSL may hang, crash, or otherwise misbehave.

Returns
An instance of *RSAPrivateKey*.

Handling partial RSA private keys

If you are trying to load RSA private keys yourself you may find that not all parameters required by `RSAPrivateNumbers` are available. In particular the Chinese Remainder Theorem (CRT) values `dmp1`, `dmq1`, `iqmp` may be missing or present in a different form. For example, OpenPGP does not include the `iqmp`, `dmp1` or `dmq1` parameters.

The following functions are provided for users who want to work with keys like this without having to do the math themselves.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`rsa_crt_iqmp`**(*p*, *q*)

Added in version 0.4.

Computes the `iqmp` (also known as `qInv`) parameter from the RSA primes `p` and `q`.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`rsa_crt_dmp1`**(*private_exponent*, *p*)

Added in version 0.4.

Computes the `dmp1` parameter from the RSA private exponent (`d`) and prime `p`.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`rsa_crt_dmq1`**(*private_exponent*, *q*)

Added in version 0.4.

Computes the `dmq1` parameter from the RSA private exponent (`d`) and prime `q`.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`rsa_recover_private_exponent`**(*e*, *p*, *q*)

Added in version 43.0.0.

Computes the RSA private_exponent (`d`) given the public exponent (`e`) and the RSA primes `p` and `q`.

> **Note**
>
> This implementation uses the Carmichael totient function to return the smallest working value of `d`. Older RSA implementations, including the original RSA paper, often used the Euler totient function, which results in larger but equally functional private exponents. The private exponents resulting from the Carmichael totient function, as returned here, are slightly more computationally efficient to use, and some modern standards require them.

`cryptography.hazmat.primitives.asymmetric.rsa.`**`rsa_recover_prime_factors`**(*n*, *e*, *d*)

Added in version 0.8.

Computes the prime factors `(p, q)` given the modulus, public exponent, and private exponent.

> **Note**
>
> When recovering prime factors this algorithm will always return `p` and `q` such that `p > q`. Note: before 1.5, this function always returned `p` and `q` such that `p < q`. It was changed because libraries commonly require `p > q`.

Returns

A tuple `(p, q)`

Key interfaces

class `cryptography.hazmat.primitives.asymmetric.rsa.`**`RSAPrivateKey`**

Added in version 0.2.

An RSA private key.

`decrypt`(*ciphertext*, *padding*)

Added in version 0.4.

> **Warning**
>
> Our implementation of PKCS1 v1.5 decryption is not constant time. See *Known security limitations* for details.

Decrypt data that was encrypted with the public key.

Parameters

- **`ciphertext`** (*`bytes`*) – The ciphertext to decrypt.
- **`padding`** – An instance of *`AsymmetricPadding`*.

Return bytes
Decrypted data.

`public_key`()

Returns
`RSAPublicKey`

An RSA public key object corresponding to the values of the private key.

`key_size`

Type
int

The bit length of the modulus.

`sign`(*data*, *padding*, *algorithm*)

Added in version 1.4.

Changed in version 1.6: *`Prehashed`* can now be used as an `algorithm`.

Sign one block of data which can be verified later by others using the public key.

Parameters

- **`data`** (*bytes-like*) – The message string to sign.
- **`padding`** – An instance of *`AsymmetricPadding`*.
- **`algorithm`** – An instance of *`HashAlgorithm`* or *`Prehashed`* if the `data` you want to sign has already been hashed.

Return bytes
Signature.

`private_numbers`()

Create a *`RSAPrivateNumbers`* object.

Returns
An *RSAPrivateNumbers* instance.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*), format (*TraditionalOpenSSL*, *OpenSSH* or *PKCS8*) and encryption algorithm (such as *BestAvailableEncryption* or *NoEncryption*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

class cryptography.hazmat.primitives.asymmetric.rsa.**RSAPublicKey**

Added in version 0.2.

An RSA public key.

encrypt(*plaintext*, *padding*)

Added in version 0.4.

Encrypt data with the public key.

Parameters

- **plaintext** (*bytes*) – The plaintext to encrypt.
- **padding** – An instance of *AsymmetricPadding*.

Return bytes
Encrypted data.

Raises
ValueError – The data could not be encrypted. One possible cause is if `data` is too large; RSA keys can only encrypt data that is smaller than the key size.

key_size

Type
int

The bit length of the modulus.

public_numbers()

Create a *RSAPublicNumbers* object.

Returns
An *RSAPublicNumbers* instance.

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*) and format (*SubjectPublicKeyInfo* or *PKCS1*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum.

Return bytes
Serialized key.

verify(*signature*, *data*, *padding*, *algorithm*)

Added in version 1.4.

Changed in version 1.6: `Prehashed` can now be used as an `algorithm`.

Verify one block of data was signed by the private key associated with this public key.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The message string that was signed.
- **padding** – An instance of `AsymmetricPadding`.
- **algorithm** – An instance of `HashAlgorithm` or `Prehashed` if the `data` you want to verify has already been hashed.

Returns
None

Raises
`cryptography.exceptions.InvalidSignature` – If the signature does not validate.

recover_data_from_signature(*signature*, *padding*, *algorithm*)

Added in version 3.3.

Recovers the signed data from the signature. The data typically contains the digest of the original message string. The `padding` and `algorithm` parameters must match the ones used when the signature was created for the recovery to succeed.

The `algorithm` parameter can also be set to `NoDigestInfo` to recover all the data present in the signature, without regard to its format or the hash algorithm used for its creation. (Note that setting `algorithm` to `None` is deprecated and have the same semantic as setting ``NoDigestInfo`.)

For `PKCS1v15` padding, this method returns the data after removing the padding layer. For standard signatures the data contains the full `DigestInfo` structure. For non-standard signatures, any data can be returned, including zero-length data.

Normally you should use the `verify()` function to validate the signature. But for some non-standard signature formats you may need to explicitly recover and validate the signed data. The following are some examples:

- Some old Thawte and Verisign timestamp certificates without `DigestInfo`.
- Signed MD5/SHA1 hashes in TLS 1.1 or earlier (**RFC 4346**, section 4.7).
- IKE version 1 signatures without `DigestInfo` (**RFC 2409**, section 5.1).

Parameters

- **signature** (*bytes*) – The signature.
- **padding** – An instance of `AsymmetricPadding`. Recovery is only supported with some of the padding types. (Currently only with `PKCS1v15`).
- **algorithm** – An instance of `HashAlgorithm`. Can be `None` to return the all the data present in the signature.

Return bytes
The signed data.

Raises

- *cryptography.exceptions.InvalidSignature* – If the signature is invalid.
- *cryptography.exceptions.UnsupportedAlgorithm* – If signature data recovery is not supported with the provided `padding` type.

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.

Diffie-Hellman key exchange

Deprecated since version 50.0.0: Diffie-Hellman over finite fields (FFDH) is deprecated and support will be removed in a future release. Users should migrate to a more modern key exchange algorithm.

Note

For security and performance reasons we suggest using *ECDH* instead of DH where possible.

Diffie-Hellman key exchange (D–H) is a method that allows two parties to jointly agree on a shared secret using an insecure channel.

Exchange Algorithm

For most applications the `shared_key` should be passed to a key derivation function. This allows mixing of additional information into the key, derivation of multiple keys, and destroys any structure that may be present.

Warning

This example does not give forward secrecy and is only provided as a demonstration of the basic Diffie-Hellman construction. For real world applications always use the ephemeral form described after this example.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate some parameters. These can be reused.
>>> parameters = dh.generate_parameters(generator=2, key_size=2048)
>>> # Generate a private key for use in the exchange.
>>> server_private_key = parameters.generate_private_key()
>>> # In a real handshake the peer is a remote client. For this
>>> # example we'll generate another local private key though. Note that in
>>> # a DH handshake both peers must agree on a common set of parameters.
>>> peer_private_key = parameters.generate_private_key()
>>> shared_key = server_private_key.exchange(peer_private_key.public_key())
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
```

(continues on next page)

(continued from previous page)

```
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
>>> # And now we can demonstrate that the handshake performed in the
>>> # opposite direction gives the same final value
>>> same_shared_key = peer_private_key.exchange(
...     server_private_key.public_key()
... )
>>> same_derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(same_shared_key)
>>> derived_key == same_derived_key
```

DHE (or EDH), the ephemeral form of this exchange, is **strongly preferred** over simple DH and provides forward secrecy when used. You must generate a new private key using *generate_private_key()* for each *exchange()* when performing an DHE key exchange. An example of the ephemeral form:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> # Generate some parameters. These can be reused.
>>> parameters = dh.generate_parameters(generator=2, key_size=2048)
>>> # Generate a private key for use in the exchange.
>>> private_key = parameters.generate_private_key()
>>> # In a real handshake the peer_public_key will be received from the
>>> # other party. For this example we'll generate another private key and
>>> # get a public key from that. Note that in a DH handshake both peers
>>> # must agree on a common set of parameters.
>>> peer_public_key = parameters.generate_private_key().public_key()
>>> shared_key = private_key.exchange(peer_public_key)
>>> # Perform key derivation.
>>> derived_key = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key)
>>> # For the next handshake we MUST generate another private key, but
>>> # we can reuse the parameters.
>>> private_key_2 = parameters.generate_private_key()
>>> peer_public_key_2 = parameters.generate_private_key().public_key()
>>> shared_key_2 = private_key_2.exchange(peer_public_key_2)
>>> derived_key_2 = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
... ).derive(shared_key_2)
```

To assemble a *DHParameters* and a *DHPublicKey* from primitive integers, you must first create the

DHParameterNumbers and *DHPublicNumbers* objects. For example, if **p**, **g**, and **y** are `int` objects received from a peer:

```
pn = dh.DHParameterNumbers(p, g)
parameters = pn.parameters()
peer_public_numbers = dh.DHPublicNumbers(y, pn)
peer_public_key = peer_public_numbers.public_key()
```

Group parameters

`cryptography.hazmat.primitives.asymmetric.dh.`**`generate_parameters`**(*generator*, *key_size*)

Added in version 1.7.

Generate a new DH parameter group.

Parameters

- **generator** – The `int` to use as a generator. Must be 2 or 5.
- **key_size** – The bit length of the prime modulus to generate.

Returns
DH parameters as a new instance of *DHParameters*.

Raises
`ValueError` – If `key_size` is not at least 512.

class `cryptography.hazmat.primitives.asymmetric.dh.`**`DHParameters`**

Added in version 1.7.

`generate_private_key`()

Generate a DH private key. This method can be used to generate many new private keys from a single set of parameters.

Returns
An instance of *DHPrivateKey*.

`parameter_numbers`()

Return the numbers that make up this set of parameters.

Returns
A *DHParameterNumbers*.

`parameter_bytes`(*encoding*, *format*)

Added in version 2.0.

Allows serialization of the parameters to bytes. Encoding (*PEM* or *DER*) and format (*PKCS3*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *ParameterFormat* enum. At the moment only `PKCS3` is supported.

Return bytes
Serialized parameters.

Key interfaces

class cryptography.hazmat.primitives.asymmetric.dh.**DHPrivateKey**

Added in version 1.7.

key_size

The bit length of the prime modulus.

public_key()

Return the public key associated with this private key.

Returns
A *DHPublicKey*.

parameters()

Return the parameters associated with this private key.

Returns
A *DHParameters*.

exchange(*peer_public_key*)

Added in version 1.7.

Parameters
peer_public_key (DHPublicKey) – The public key for the peer.

Return bytes
The agreed key. The bytes are ordered in 'big' endian.

private_numbers()

Return the numbers that make up this private key.

Returns
A *DHPrivateNumbers*.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Added in version 1.8.

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*), format (*PKCS8*) and encryption algorithm (such as *BestAvailableEncryption* or *NoEncryption*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

class cryptography.hazmat.primitives.asymmetric.dh.**DHPublicKey**

Added in version 1.7.

key_size

The bit length of the prime modulus.

parameters()

Return the parameters associated with this private key.

Returns
A *DHParameters*.

public_numbers()

Return the numbers that make up this public key.

Returns
A *DHPublicNumbers*.

public_bytes(*encoding*, *format*)

Added in version 1.8.

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*) and format (*SubjectPublicKeyInfo*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum.

Return bytes
Serialized key.

Numbers

class cryptography.hazmat.primitives.asymmetric.dh.**DHParameterNumbers**(*p*, *g*, *q=None*)

Added in version 0.8.

The collection of integers that define a Diffie-Hellman group.

p

Type
int

The prime modulus value.

g

Type
int

The generator value. Must be 2 or greater.

q

Added in version 1.8.

Type
int

p subgroup order value.

parameters()

Added in version 1.7.

Returns
A new instance of *DHParameters*.

Raises
`ValueError` – If the parameters are invalid.

class `cryptography.hazmat.primitives.asymmetric.dh.`**`DHPrivateNumbers`**(*x*, *public_numbers*)

Added in version 0.8.

The collection of integers that make up a Diffie-Hellman private key.

`public_numbers`

Type
DHPublicNumbers

The *DHPublicNumbers* which makes up the DH public key associated with this DH private key.

`x`

Type
int

The private value.

`private_key`()

Added in version 1.7.

Returns
A new instance of *DHPrivateKey*.

class `cryptography.hazmat.primitives.asymmetric.dh.`**`DHPublicNumbers`**(*y*, *parameter_numbers*)

Added in version 0.8.

The collection of integers that make up a Diffie-Hellman public key.

`parameter_numbers`

Type
DHParameterNumbers

The parameters for this DH group.

`y`

Type
int

The public value.

`public_key`()

Added in version 1.7.

Returns
A new instance of *DHPublicKey*.

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.

DSA

> **Note**
>
> DSA is a **legacy algorithm** and should generally be avoided in favor of choices like *EdDSA using curve25519* or *ECDSA*.

DSA is a public-key algorithm for signing messages.

Generation

`cryptography.hazmat.primitives.asymmetric.dsa.`**`generate_private_key`**(*key_size*)

Added in version 0.5.

Changed in version 3.0: Added support for 4096-bit keys for some legacy applications that continue to use DSA despite the wider cryptographic community's ongoing protestations.

Generate a DSA private key from the given key size. This function will generate a new set of parameters and key in one step.

Parameters

key_size (*int*) – The length of the modulus in *bits*. It should be either 1024, 2048, 3072, or 4096. For keys generated in 2015 this should be at least 2048 (See page 41).

Returns

An instance of *DSAPrivateKey*.

`cryptography.hazmat.primitives.asymmetric.dsa.`**`generate_parameters`**(*key_size*)

Added in version 0.5.

Changed in version 3.0: Added support for 4096-bit keys for some legacy applications that continue to use DSA despite the wider cryptographic community's ongoing protestations.

Generate DSA parameters.

Parameters

key_size (*int*) – The length of *p*. It should be either 1024, 2048, 3072, or 4096. For keys generated in 2015 this should be at least 2048 (See page 41).

Returns

An instance of *DSAParameters*.

Signing

Using a *DSAPrivateKey* instance.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import dsa
>>> private_key = dsa.generate_private_key(
...     key_size=1024,
... )
>>> data = b"this is some data I'd like to sign"
>>> signature = private_key.sign(
...     data,
...     hashes.SHA256()
... )
```

The `signature` is a `bytes` object, whose contents is DER encoded as described in **RFC 3279**. This can be decoded using *decode_dss_signature()*.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
...     utils.Prehashed(chosen_hash)
... )
```

Verification

Verification is performed using a *DSAPublicKey* instance. You can get a public key object with *load_pem_public_key()*, *load_der_public_key()*, *public_key()* , or *public_key()*.

```
>>> public_key = private_key.public_key()
>>> public_key.verify(
...     signature,
...     data,
...     hashes.SHA256()
... )
```

`verify()` takes the signature in the same format as is returned by `sign()`.

`verify()` will raise an *InvalidSignature* exception if the signature isn't valid.

If your data is too large to be passed in a single call, you can hash it separately and pass that value using *Prehashed*.

```
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash)
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> public_key.verify(
...     sig,
...     digest,
...     utils.Prehashed(chosen_hash)
... )
```

Numbers

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAParameterNumbers**(*p*, *q*, *g*)

Added in version 0.5.

The collection of integers that make up a set of DSA parameters.

p

Type
int

The public modulus.

q

Type
int

The sub-group order.

g

Type
int

The generator.

parameters()

Returns
A new instance of *DSAParameters*.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPublicNumbers**(*y*, *parameter_numbers*)

Added in version 0.5.

The collection of integers that make up a DSA public key.

y

Type
int

The public value `y`.

parameter_numbers

Type
DSAParameterNumbers

The *DSAParameterNumbers* associated with the public key.

public_key()

Returns
A new instance of *DSAPublicKey*.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPrivateNumbers**(*x*, *public_numbers*)

Added in version 0.5.

The collection of integers that make up a DSA private key.

> **Warning**
>
> Revealing the value of `x` will compromise the security of any cryptographic operations performed.

x

Type
int

The private value `x`.

public_numbers

Type
DSAPublicNumbers

The *DSAPublicNumbers* associated with the private key.

private_key()

Returns
A new instance of *DSAPrivateKey*.

Key interfaces

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAParameters**

Added in version 0.3.

DSA parameters.

generate_private_key()

Added in version 0.5.

Generate a DSA private key. This method can be used to generate many new private keys from a single set of parameters.

Returns
An instance of *DSAPrivateKey*.

parameter_numbers()

Create a *DSAParameterNumbers* object.

Returns
A *DSAParameterNumbers* instance.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPrivateKey**

Added in version 0.3.

A DSA private key.

public_key()

Returns
DSAPublicKey

An DSA public key object corresponding to the values of the private key.

parameters()

Returns
DSAParameters

The DSAParameters object associated with this private key.

key_size

Type
int

The bit length of *p*.

sign(*data*, *algorithm*)

Added in version 1.5.

Changed in version 1.6: *Prehashed* can now be used as an `algorithm`.

Sign one block of data which can be verified later by others using the public key.

Parameters

- **data** (*bytes-like*) – The message string to sign.
- **algorithm** – An instance of *HashAlgorithm* or *Prehashed* if the `data` you want to sign has already been hashed.

Return bytes
Signature.

private_numbers()

Create a *DSAPrivateNumbers* object.

Returns
A *DSAPrivateNumbers* instance.

private_bytes(*encoding*, *format*, *encryption_algorithm*)

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*), format (*TraditionalOpenSSL*, or *PKCS8*) and encryption algorithm (such as *BestAvailableEncryption* or *NoEncryption*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PrivateFormat* enum.
- **encryption_algorithm** – An instance of an object conforming to the *KeySerializationEncryption* interface.

Return bytes
Serialized key.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPublicKey**

Added in version 0.3.

A DSA public key.

key_size

Type
int

The bit length of *p*.

parameters()

Returns
DSAParameters

The DSAParameters object associated with this public key.

public_numbers()

Create a *DSAPublicNumbers* object.

Returns
A *DSAPublicNumbers* instance.

public_bytes(*encoding*, *format*)

Allows serialization of the key to bytes. Encoding (*PEM* or *DER*) and format (*SubjectPublicKeyInfo*) are chosen to define the exact serialization.

Parameters

- **encoding** – A value from the *Encoding* enum.
- **format** – A value from the *PublicFormat* enum.

Return bytes
Serialized key.

verify(*signature*, *data*, *algorithm*)

Added in version 1.5.

Changed in version 1.6: *Prehashed* can now be used as an `algorithm`.

Verify one block of data was signed by the private key associated with this public key.

Parameters

- **signature** (*bytes-like*) – The signature to verify.
- **data** (*bytes-like*) – The message string that was signed.
- **algorithm** – An instance of *HashAlgorithm* or *Prehashed* if the `data` you want to sign has already been hashed.

Returns
None

Raises
cryptography.exceptions.InvalidSignature – If the signature does not validate.

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.

Key Serialization

There are several common schemes for serializing asymmetric private and public keys to bytes. They generally support encryption of private keys and additional key metadata.

Many serialization formats support multiple different types of asymmetric keys and will return an instance of the appropriate type. You should check that the returned key matches the type your application expects when using these methods.

```
>>> from cryptography.hazmat.primitives.asymmetric import dsa, rsa
>>> from cryptography.hazmat.primitives.serialization import load_pem_private_
↪key
>>> key = load_pem_private_key(pem_data, password=None)
>>> if isinstance(key, rsa.RSAPrivateKey):
...     signature = sign_with_rsa_key(key, message)
... elif isinstance(key, dsa.DSAPrivateKey):
...     signature = sign_with_dsa_key(key, message)
... else:
...     raise TypeError
```

Key dumping

The `serialization` module contains functions for loading keys from `bytes`. To dump a `key` object to `bytes`, you must call the appropriate method on the key object. Documentation for these methods is found in the *rsa*, *dsa*, and *ec* module documentation.

PEM

PEM is an encapsulation format, meaning keys in it can actually be any of several different key types. However these are all self-identifying, so you don't need to worry about this detail. PEM keys are recognizable because they all begin with `-----BEGIN {format}-----` and end with `-----END {format}-----`.

> **Note**
>
> A PEM block which starts with `-----BEGIN CERTIFICATE-----` is not a public or private key, it's an *X.509 Certificate*. You can load it using *load_pem_x509_certificate()* and extract the public key with *Certificate.public_key*.

`cryptography.hazmat.primitives.serialization.`**`load_pem_private_key`**(*data*, *password*, **, unsafe_skip_rsa_key_validation=False*)

Added in version 0.6.

> **Note**
>
> SSH private keys are a different format and must be loaded with *load_ssh_private_key()*.

Deserialize a private key from PEM encoded data to one of the supported asymmetric private key types.

Parameters

- **`data`** (*bytes-like*) – The PEM encoded key data.
- **`password`** (*bytes-like*) – The password to use to decrypt the data. Should be `None` if the private key is not encrypted.
- **`unsafe_skip_rsa_key_validation`** (*bool*) – Added in version 39.0.0.

  A keyword-only argument that defaults to `False`. If `True` RSA private keys will not be validated. This significantly speeds up loading the keys, but is *unsafe* unless you are certain the key is valid. User supplied keys should never be loaded with this parameter set to `True`. If you do load an invalid key this way and attempt to use it OpenSSL may hang, crash, or otherwise misbehave.

Returns

One of *Ed25519PrivateKey*, *X25519PrivateKey*, *Ed448PrivateKey*, *X448PrivateKey*, *RSAPrivateKey*, *DSAPrivateKey*, *DHPrivateKey*, or *EllipticCurvePrivateKey* depending on the contents of `data`.

Raises

- **`ValueError`** – If the PEM data could not be decrypted or if its structure could not be decoded successfully.
- **`TypeError`** – If a `password` was given and the private key was not encrypted. Or if the key was encrypted but no password was supplied.

- *cryptography.exceptions.UnsupportedAlgorithm* – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

`cryptography.hazmat.primitives.serialization.`**`load_pem_public_key`**(*data*)

Added in version 0.6.

Deserialize a public key from PEM encoded data to one of the supported asymmetric public key types. The PEM encoded data is typically a `subjectPublicKeyInfo` payload as specified in **RFC 5280**.

```
>>> from cryptography.hazmat.primitives.serialization import load_pem_public_key
>>> key = load_pem_public_key(public_pem_data)
>>> isinstance(key, rsa.RSAPublicKey)
True
```

Parameters

data (*bytes*) – The PEM encoded key data.

Returns

One of *Ed25519PublicKey*, *X25519PublicKey*, *Ed448PublicKey*, *X448PublicKey*, *RSAPublicKey*, *DSAPublicKey*, *DHPublicKey*, or *EllipticCurvePublicKey* depending on the contents of `data`.

Raises

- **ValueError** – If the PEM data's structure could not be decoded successfully.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

`cryptography.hazmat.primitives.serialization.`**`load_pem_parameters`**(*data*)

`.. versionadded:: 2.0`

Deprecated since version 50.0.0: Diffie-Hellman over finite fields (FFDH) is deprecated and support will be removed in a future release.

Deserialize parameters from PEM encoded data to one of the supported asymmetric parameters types.

```
>>> from cryptography.hazmat.primitives.serialization import load_pem_parameters
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> parameters = load_pem_parameters(parameters_pem_data)
>>> isinstance(parameters, dh.DHParameters)
True
```

Parameters

data (*bytes*) – The PEM encoded parameters data.

Returns

Currently only *DHParameters* supported.

Raises

- **ValueError** – If the PEM data's structure could not be decoded successfully.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

DER

DER is an ASN.1 encoding type. There are no encapsulation boundaries and the data is binary. DER keys may be in a variety of formats, but as long as you know whether it is a public or private key the loading functions will handle the rest.

`cryptography.hazmat.primitives.serialization.load_der_private_key(data, password, *, unsafe_skip_rsa_key_validation=False)`

Added in version 0.8.

Deserialize a private key from DER encoded data to one of the supported asymmetric private key types.

Parameters

- **data** (*bytes-like*) – The DER encoded key data.
- **password** (*bytes-like*) – The password to use to decrypt the data. Should be `None` if the private key is not encrypted.
- **unsafe_skip_rsa_key_validation** (*bool*) – Added in version 39.0.0.

  A keyword-only argument that defaults to `False`. If `True` RSA private keys will not be validated. This significantly speeds up loading the keys, but is *unsafe* unless you are certain the key is valid. User supplied keys should never be loaded with this parameter set to `True`. If you do load an invalid key this way and attempt to use it OpenSSL may hang, crash, or otherwise misbehave.

Returns

One of *Ed25519PrivateKey*, *X25519PrivateKey*, *Ed448PrivateKey*, *X448PrivateKey*, *RSAPrivateKey*, *DSAPrivateKey*, *DHPrivateKey*, or *EllipticCurvePrivateKey* depending on the contents of `data`.

Raises

- **ValueError** – If the DER data could not be decrypted or if its structure could not be decoded successfully.
- **TypeError** – If a `password` was given and the private key was not encrypted. Or if the key was encrypted but no password was supplied.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.hazmat.primitives.serialization import load_der_private_key
>>> key = load_der_private_key(der_data, password=None)
>>> isinstance(key, rsa.RSAPrivateKey)
True
```

`cryptography.hazmat.primitives.serialization.load_der_public_key(data)`

Added in version 0.8.

Deserialize a public key from DER encoded data to one of the supported asymmetric public key types. The DER encoded data is typically a `subjectPublicKeyInfo` payload as specified in **RFC 5280**.

Parameters

data (*bytes*) – The DER encoded key data.

Returns

One of *Ed25519PublicKey*, *X25519PublicKey*, *Ed448PublicKey*, *X448PublicKey*,

RSAPublicKey, *DSAPublicKey*, *DHPublicKey*, or *EllipticCurvePublicKey* depending on the contents of `data`.

Raises

- **ValueError** – If the DER data's structure could not be decoded successfully.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.hazmat.primitives.serialization import load_der_public_key
>>> key = load_der_public_key(public_der_data)
>>> isinstance(key, rsa.RSAPublicKey)
True
```

`cryptography.hazmat.primitives.serialization.`**load_der_parameters**(*data*)

Added in version 2.0.

Deprecated since version 50.0.0: Diffie-Hellman over finite fields (FFDH) is deprecated and support will be removed in a future release.

Deserialize parameters from DER encoded data to one of the supported asymmetric parameters types.

Parameters

data (*bytes*) – The DER encoded parameters data.

Returns

Currently only *DHParameters* supported.

Raises

- **ValueError** – If the DER data's structure could not be decoded successfully.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the serialized key type is not supported by the OpenSSL version `cryptography` is using.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.primitives.serialization import load_der_parameters
>>> parameters = load_der_parameters(parameters_der_data)
>>> isinstance(parameters, dh.DHParameters)
True
```

OpenSSH Public Key

The format used by OpenSSH to store public keys, as specified in **RFC 4253**.

An example RSA key in OpenSSH format (line breaks added for formatting purposes):

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDDu/XRP1kyK6Cgt36gts9XAk
FiiuJLW6RU0j3KKVZSs1I7Z3UmU9/9aVh/rZV43WQG8jaR6kkcP4stOR0DEtll
PDA7ZRBnrfiHpSQYQ874AZaAoIjgkv7DBfsE6gcDQLub0PFjWyrYQUJhtOLQEK
vY/G0vt2iRL3juawWmCFdTK3W3XvwAdgGk71i6lHt+deOPNEPN2H58E4odrZ2f
sxn/adpDqfb2sM0kPwQs0aWvrrKGvUaustkivQE4XWiSFnB0oJB/lKK/CKVKuy
///ImSCGHQRvhwariN2tvZ6CBNSLh3iQgeB0AkyJlng7MXB2qYq/Ci2FUOryCX
2MzHvnbv testkey@localhost
```

DSA keys look almost identical but begin with `ssh-dss` rather than `ssh-rsa`. ECDSA keys have a slightly different format, they begin with `ecdsa-sha2-{curve}`.

`cryptography.hazmat.primitives.serialization.SSHPublicKeyTypes`

Added in version 40.0.0.

Type alias: A union of public key types accepted for SSH: *RSAPublicKey*, *DSAPublicKey*, *EllipticCurvePublicKey* , or *Ed25519PublicKey*.

`cryptography.hazmat.primitives.serialization.load_ssh_public_key(data)`

Added in version 0.7.

> **Note**
>
> SSH DSA key support is deprecated and will be removed in a future release.

Deserialize a public key from OpenSSH (**RFC 4253** and PROTOCOL.certkeys) encoded data to an instance of the public key type.

Parameters

data (*bytes-like*) – The OpenSSH encoded key data.

Returns

One of *SSHPublicKeyTypes* depending on the contents of `data`.

Raises

- **ValueError** – If the OpenSSH data could not be properly decoded or if the key is not in the proper format.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the serialized key is of a type that is not supported.

`cryptography.hazmat.primitives.serialization.ssh_key_fingerprint(key, hash_algorithm)`

Added in version 45.0.0.

Computes the fingerprint of an SSH public key. The fingerprint is the raw bytes of the hash, depending on your use you may need to encode the data as base64 or hex.

Parameters

- **key** (One of *SSHPublicKeyTypes*) – The public key to compute the fingerprint for.
- **hash_algorithm** – The hash algorithm to use, either `MD5()`, `SHA1()`, or `SHA256()`.

Returns

The key fingerprint.

Return type

bytes

```
>>> from cryptography.hazmat.primitives.serialization import load_ssh_public_key,␣
↪ssh_key_fingerprint
>>> key_data = b"ssh-ed25519␣
↪AAAAC3NzaC1lZDI1NTE5AAAAIAhVNvf1vigXfagQXKjdKN5zEF12KWVMVdDrU3sVLhgd user@example.
↪com"
>>> public_key = load_ssh_public_key(key_data)
>>> md5_fingerprint = ssh_key_fingerprint(public_key, hashes.MD5())
>>> md5_fingerprint
b'\x95\xf6\xc0\xe3so\xaen\xcc\x98\xbb\xf4\xd8BJ\x15'
>>> sha256_fingerprint = ssh_key_fingerprint(public_key, hashes.SHA256())
>>> sha256_fingerprint
```

(continues on next page)

(continued from previous page)

```
b'R\x0f*!\x99f9\x9a\xcd\x98[\xe8-&\xbah\xa6x\x96\x87\xb3\xf9\xe0\x9b\xb1,\xcc\xbdt\
↪xd4\xc3\xb7'
```

OpenSSH Private Key

The format used by OpenSSH to store private keys, as approximately specified in PROTOCOL.key.

An example ECDSA key in OpenSSH format:

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAaAAAABNlY2RzYS
1zaGEyLW5pc3RwMjU2AAAACG5pc3RwMjU2AAAAQQRI0fWnI1CxX7qYqp0ih6bxjhGmUrZK
/Axf8vhM8Db3oH7CFR+JdL715lUdu4XCWvQZKVf60/h3kBFhuxQC23XjAAAAqKPzVaOj81
WjAAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBEjR9acjULFfupiq
nSKHpvGOEaZStkr8DF/y+EzwNvegfsIVH4l0vvXmVR27hcJa9BkpV/rT+HeQEWG7FALbde
MAAAAga/VGV2asRlL3kXXao0aochQ59nXHA2xEGeAoQd952r0AAAAJbWFya29AdmZmAQID
BAUGBw==
-----END OPENSSH PRIVATE KEY-----
```

`cryptography.hazmat.primitives.serialization.`**`SSHPrivateKeyTypes`**

Added in version 40.0.0.

Type alias: A union of private key types accepted for SSH: *RSAPrivateKey*, *DSAPrivateKey*, *EllipticCurvePrivateKey* or *Ed25519PrivateKey*.

`cryptography.hazmat.primitives.serialization.`**`load_ssh_private_key`**(*data*, *password*, *, *unsafe_skip_rsa_key_validation=False*)

Added in version 3.0.

> **Note**
>
> SSH DSA key support is deprecated and will be removed in a future release.

Deserialize a private key from OpenSSH encoded data to an instance of the private key type.

Parameters

- **data** (*bytes-like*) – The PEM encoded OpenSSH private key data.
- **password** (*bytes*) – Password bytes to use to decrypt password-protected key. Or `None` if not needed.
- **unsafe_skip_rsa_key_validation** (*bool*) – Added in version 45.0.0.

  A keyword-only argument that defaults to `False`. If `True` RSA private keys will not be validated. This significantly speeds up loading the keys, but is *unsafe* unless you are certain the key is valid. User supplied keys should never be loaded with this parameter set to `True`. If you do load an invalid key this way and attempt to use it OpenSSL may hang, crash, or otherwise misbehave.

Returns

One of *SSHPrivateKeyTypes* depending on the contents of `data`.

Raises

- **ValueError** – If the OpenSSH data could not be properly decoded, if the key is not in the proper format or the incorrect password was provided.

- *cryptography.exceptions.UnsupportedAlgorithm* – If the serialized key is of a type that is not supported.

OpenSSH Certificate

The format used by OpenSSH for certificates, as specified in PROTOCOL.certkeys.

`cryptography.hazmat.primitives.serialization.`**`SSHCertPublicKeyTypes`**

Added in version 40.0.0.

Type alias: A union of public key types supported for SSH certificates: *RSAPublicKey*, *EllipticCurvePublicKey* or *Ed25519PublicKey*

`cryptography.hazmat.primitives.serialization.`**`SSHCertPrivateKeyTypes`**

Added in version 40.0.0.

Type alias: A union of private key types supported for SSH certificates: *RSAPrivateKey*, *EllipticCurvePrivateKey* or *Ed25519PrivateKey*

`cryptography.hazmat.primitives.serialization.`**`load_ssh_public_identity`**`(`*data*`)`

Added in version 40.0.0.

> **Note**
>
> This function does not support parsing certificates with DSA public keys or signatures from DSA certificate authorities. DSA is a deprecated algorithm and should not be used.

Deserialize an OpenSSH encoded identity to an instance of *SSHCertificate* or the appropriate public key type. Parsing a certificate does not verify anything. It is up to the caller to perform any necessary verification.

Parameters

data (*bytes*) – The OpenSSH encoded data.

Returns

SSHCertificate or one of *SSHCertPublicKeyTypes*.

Raises

- **ValueError** – If the OpenSSH data could not be properly decoded.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the data contains a public key type that is not supported.

class `cryptography.hazmat.primitives.serialization.`**`SSHCertificate`**

Added in version 40.0.0.

nonce

Type

bytes

The nonce field is a CA-provided random value of arbitrary length (but typically 16 or 32 bytes) included to make attacks that depend on inducing collisions in the signature hash infeasible.

public_key`()`

The public key contained in the certificate, one of *SSHCertPublicKeyTypes*.

serial

Type
int

Serial is an optional certificate serial number set by the CA to provide an abbreviated way to refer to certificates from that CA. If a CA does not wish to number its certificates, it must set this field to zero.

type

Type
SSHCertificateType

Type specifies whether this certificate is for identification of a user or a host.

key_id

Type
bytes

This is a free-form text field that is filled in by the CA at the time of signing; the intention is that the contents of this field are used to identify the identity principal in log messages.

valid_principals

Type
list[bytes]

"valid principals" is a list containing one or more principals as byte strings. These principals list the names for which this certificate is valid; hostnames for host certificates and usernames for user certificates. As a special case, an empty list means the certificate is valid for any principal of the specified type.

valid_after

Type
int

An integer representing the Unix timestamp (in UTC) after which the certificate is valid. **This time is inclusive.**

valid_before

Type
int

An integer representing the Unix timestamp (in UTC) before which the certificate is valid. **This time is not inclusive.**

critical_options

Type
dict[bytes, bytes]

Critical options is a dict of zero or more options that are critical for the certificate to be considered valid. If any of these options are not supported by the implementation, the certificate must be rejected.

extensions

Type
dict[bytes, bytes]

Extensions is a dict of zero or more options that are non-critical for the certificate to be considered valid. If any of these options are not supported by the implementation, the implementation may safely ignore them.

signature_key()

The public key used to sign the certificate, one of *SSHCertPublicKeyTypes*.

verify_cert_signature()

> **Warning**
>
> This method does not validate anything about whether the signing key is trusted! Callers are responsible for validating trust in the signer.

Validates that the signature on the certificate was created by the private key associated with the certificate's signature key and that the certificate has not been changed since signing.

Returns
None

Raises
InvalidSignature if the signature is invalid.

public_bytes()

Returns
The serialized certificate in OpenSSH format.

Return type
bytes

class cryptography.hazmat.primitives.serialization.**SSHCertificateType**

Added in version 40.0.0.

An enumeration of the types of SSH certificates.

USER

The cert is intended for identification of a user. Corresponds to the value 1.

HOST

The cert is intended for identification of a host. Corresponds to the value 2.

SSH Certificate Builder

class cryptography.hazmat.primitives.serialization.**SSHCertificateBuilder**

Added in version 40.0.0.

> **Note**
>
> This builder does not support generating certificates with DSA public keys or creating signatures with DSA certificate authorities. DSA is a deprecated algorithm and should not be used.

```
>>> import datetime
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives.serialization import (
...     SSHCertificateType, SSHCertificateBuilder
... )
>>> signing_key = ec.generate_private_key(ec.SECP256R1())
```

(continues on next page)

(continued from previous page)

```
>>> public_key = ec.generate_private_key(ec.SECP256R1()).public_key()
>>> valid_after = datetime.datetime(
...     2023, 1, 1, 1, tzinfo=datetime.timezone.utc
... ).timestamp()
>>> valid_before = datetime.datetime(
...     2023, 7, 1, 1, tzinfo=datetime.timezone.utc
... ).timestamp()
>>> key_id = b"a_key_id"
>>> valid_principals = [b"eve", b"alice"]
>>> builder = (
...     SSHCertificateBuilder()
...     .public_key(public_key)
...     .type(SSHCertificateType.USER)
...     .valid_before(valid_before)
...     .valid_after(valid_after)
...     .key_id(b"a_key_id")
...     .valid_principals(valid_principals)
...     .add_extension(b"no-touch-required", b"")
... )
>>> builder.sign(signing_key).public_bytes()
b'...'
```

public_key(*public_key*)

Parameters

public_key (*SSHCertPublicKeyTypes*) – The public key to be included in the certificate. This value is required.

serial(*serial*)

Parameters

serial (*int*) – The serial number to be included in the certificate. This is not a required value and will be set to zero if not provided. Value must be between 0 and 2:sup:*64* - 1, inclusive.

type(*type*)

Parameters

type (*SSHCertificateType*) – The type of the certificate. There are two options, user or host.

key_id(*key_id*)

Parameters

key_id (*bytes*) – The key ID to be included in the certificate. This is not a required value.

valid_principals(*valid_principals*)

Parameters

valid_principals (*list[bytes]*) – A list of principals that the certificate is valid for. This is a required value unless *valid_for_all_principals()* has been called.

valid_for_all_principals()

Marks the certificate as valid for all principals. This cannot be set if principals have been added via *valid_principals()*.

valid_after(*valid_after*)

Parameters
valid_after (*int*) – The Unix timestamp (in UTC) that marks the activation time for the certificate. This is a required value.

valid_before(*valid_before*)

Parameters
valid_before (*int*) – The Unix timestamp (in UTC) that marks the expiration time for the certificate. This is a required value.

add_critical_option(*name*, *value*)

Parameters

- **name** (*bytes*) – The name of the critical option to add. No duplicates are allowed.
- **value** (*bytes*) – The value of the critical option to add. This is commonly an empty byte string.

add_extension(*name*, *value*)

Parameters

- **name** (*bytes*) – The name of the extension to add. No duplicates are allowed.
- **value** (*bytes*) – The value of the extension to add.

sign(*private_key*)

Parameters
private_key (*SSHCertPrivateKeyTypes*) – The private key that will be used to sign the certificate.

Returns
The signed certificate.

Return type
SSHCertificate

PKCS12

PKCS12 is a binary format described in **RFC 7292**. It can contain certificates, keys, and more. PKCS12 files commonly have a `pfx` or `p12` file suffix.

> **Note**
>
> `cryptography` only supports a single private key and associated certificates when parsing PKCS12 files at this time.

`cryptography.hazmat.primitives.serialization.pkcs12.`**PKCS12PrivateKeyTypes**

Added in version 40.0.0.

Type alias: A union of private key types supported for PKCS12 serialization: *RSAPrivateKey* , *EllipticCurvePrivateKey* , *Ed25519PrivateKey* , *Ed448PrivateKey* or *DSAPrivateKey*.

`cryptography.hazmat.primitives.serialization.pkcs12.`**`load_key_and_certificates`**(*data*, *password*)

Added in version 2.5.

Deserialize a PKCS12 blob.

Parameters

- **data** (*bytes-like*) – The binary data.
- **password** (*bytes-like*) – The password to use to decrypt the data. `None` if the PKCS12 is not encrypted.

Returns

A tuple of `(private_key, certificate, additional_certificates)`. `private_key` is a private key type or `None`, `certificate` is either the `Certificate` whose public key matches the private key in the PKCS 12 object or `None`, and `additional_certificates` is a list of all other `Certificate` instances in the PKCS12 object.

`cryptography.hazmat.primitives.serialization.pkcs12.`**`load_pkcs12`**(*data*, *password*)

Added in version 36.0.0.

Deserialize a PKCS12 blob, and return a `PKCS12KeyAndCertificates` instance.

Parameters

- **data** (*bytes-like*) – The binary data.
- **password** (*bytes-like*) – The password to use to decrypt the data. `None` if the PKCS12 is not encrypted.

Returns

A `PKCS12KeyAndCertificates` instance.

`cryptography.hazmat.primitives.serialization.pkcs12.`**`serialize_key_and_certificates`**(*name*, *key*, *cert*, *cas*, *encryption_algorithm*)

Added in version 3.0.

> **Note**
>
> With OpenSSL 3.0.0+ the defaults for encryption when serializing PKCS12 have changed and some versions of Windows and macOS will not be able to read the new format. Maximum compatibility can be achieved by using `SHA1` for MAC algorithm and `PBESv1SHA1And3KeyTripleDESCBC` for encryption algorithm as seen in the example below. However, users should avoid this unless required for compatibility.

> **Warning**
>
> PKCS12 encryption is typically not secure and should not be used as a security mechanism. Wrap a PKCS12 blob in a more secure envelope if you need to store or send it safely.

Serialize a PKCS12 blob.

> **Note**
>
> Due to a bug in Firefox it's not possible to load unencrypted PKCS12 blobs in Firefox.

Parameters

- **name** (*bytes*) – The friendly name to use for the supplied certificate and key.
- **key** (*PKCS12PrivateKeyTypes*) – The private key to include in the structure.
- **cert** (*Certificate* or None) – The certificate associated with the private key.
- **cas** (None, or list of *Certificate* or *PKCS12Certificate*) – An optional set of certificates to also include in the structure. If a *PKCS12Certificate* is given, its friendly name will be serialized.
- **encryption_algorithm** – The encryption algorithm that should be used for the key and certificate. An instance of an object conforming to the *KeySerializationEncryption* interface. PKCS12 encryption is typically **very weak** and should not be used as a security boundary.

Return bytes

Serialized PKCS12.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives.serialization import␣
↪BestAvailableEncryption, load_pem_private_key, pkcs12
>>> cert = x509.load_pem_x509_certificate(ca_cert)
>>> key = load_pem_private_key(ca_key, None)
>>> p12 = pkcs12.serialize_key_and_certificates(
...     b"friendlyname", key, cert, None, BestAvailableEncryption(b"password")
... )
```

This example uses an `encryption_builder()` to create a PKCS12 with more compatible, but substantially less secure, encryption.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.serialization import PrivateFormat, load_
↪pem_private_key, pkcs12
>>> encryption = (
...     PrivateFormat.PKCS12.encryption_builder().
...     kdf_rounds(50000).
...     key_cert_algorithm(pkcs12.PBES.PBESv1SHA1And3KeyTripleDESCBC).
...     hmac_hash(hashes.SHA1()).build(b"my password")
... )
>>> cert = x509.load_pem_x509_certificate(ca_cert)
>>> key = load_pem_private_key(ca_key, None)
>>> p12 = pkcs12.serialize_key_and_certificates(
...     b"friendlyname", key, cert, None, encryption
... )
```

`cryptography.hazmat.primitives.serialization.pkcs12.`**`serialize_java_truststore`**(*pkcs12_certs*, *encryption_algorithm*)

Added in version 45.0.0.

> **Warning**
>
> PKCS12 encryption is typically not secure and should not be used as a security mechanism. Wrap a PKCS12 blob in a more secure envelope if you need to store or send it safely.

Serialize a PKCS12 blob containing provided certificates. Java expects an internal flag to denote truststore usage, which this function adds.

Parameters

- **certs** (A list of *PKCS12Certificate* instances.) – A set of certificates to also include in the structure.
- **encryption_algorithm** – The encryption algorithm that should be used for the key and certificate. An instance of an object conforming to the *KeySerializationEncryption* interface. PKCS12 encryption is typically **very weak** and should not be used as a security boundary.

Return bytes

Serialized PKCS12.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives.serialization import␣
↪BestAvailableEncryption, pkcs12
>>> cert = x509.load_pem_x509_certificate(ca_cert)
>>> p12 = pkcs12.serialize_java_truststore(
...     [pkcs12.PKCS12Certificate(cert, b"friendlyname")], BestAvailableEncryption(b
↪"password")
... )
```

class cryptography.hazmat.primitives.serialization.pkcs12.**PKCS12Certificate**(*cert*, *friendly_name=None*)

Added in version 36.0.0.

Represents additional data provided for a certificate in a PKCS12 file.

Parameters

- **cert** (*Certificate*) – The certificate to associate with the additional data.
- **friendly_name** (*bytes or None*) – An optional friendly name for the certificate.

certificate

A *Certificate* instance.

friendly_name

Type

bytes or None

An optional byte string containing the friendly name of the certificate.

class cryptography.hazmat.primitives.serialization.pkcs12.**PKCS12KeyAndCertificates**

Added in version 36.0.0.

A simplified representation of a PKCS12 file.

key

An optional private key belonging to *cert* (see *PKCS12PrivateKeyTypes*).

cert

An optional *PKCS12Certificate* instance belonging to the private key *key*.

additional_certs

A list of *PKCS12Certificate* instances.

class cryptography.hazmat.primitives.serialization.pkcs12.**PBES**

Added in version 38.0.0.

An enumeration of password-based encryption schemes used in PKCS12. These values are used with *KeySerializationEncryptionBuilder*.

PBESv1SHA1And3KeyTripleDESCBC

PBESv1 using SHA1 as the KDF PRF and 3-key triple DES-CBC as the cipher.

PBESv2SHA256AndAES256CBC

PBESv2 using SHA256 as the KDF PRF and AES256-CBC as the cipher. This is only supported on OpenSSL 3.0.0 or newer.

PKCS7

PKCS7 is a format described in **RFC 2315**, among other specifications. It can contain certificates, CRLs, and much more. PKCS7 files commonly have a `p7b`, `p7m`, or `p7s` file suffix but other suffixes are also seen in the wild.

cryptography.hazmat.primitives.serialization.pkcs7.**PKCS7HashTypes**

Added in version 40.0.0.

Type alias: A union of hash types supported for PKCS7 serialization: *SHA1*, *SHA224*, *SHA256*, *SHA384*, or *SHA512*.

cryptography.hazmat.primitives.serialization.pkcs7.**PKCS7PrivateKeyTypes**

Added in version 40.0.0.

Type alias: A union of private key types supported for PKCS7 serialization: *RSAPrivateKey* or *EllipticCurvePrivateKey*

cryptography.hazmat.primitives.serialization.pkcs7.**load_pem_pkcs7_certificates**(*data*)

Added in version 3.1.

Deserialize a PEM encoded PKCS7 blob to a list of certificates. PKCS7 can contain many other types of data, including CRLs, but this function will ignore everything except certificates.

Parameters

data (*bytes*) – The data.

Returns

A list of *Certificate*.

Raises

- **ValueError** – If the PKCS7 data could not be loaded.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the PKCS7 data is of a type that is not supported.

cryptography.hazmat.primitives.serialization.pkcs7.**load_der_pkcs7_certificates**(*data*)

Added in version 3.1.

Deserialize a DER encoded PKCS7 blob to a list of certificates. PKCS7 can contain many other types of data, including CRLs, but this function will ignore everything except certificates.

Parameters
data (*bytes*) – The data.

Returns
A list of *Certificate*.

Raises

- **ValueError** – If the PKCS7 data could not be loaded.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the PKCS7 data is of a type that is not supported.

cryptography.hazmat.primitives.serialization.pkcs7.**serialize_certificates**(*certs*, *encoding*)

Added in version 37.0.0.

Serialize a list of certificates to a PKCS7 structure.

Parameters

- **certs** – A list of *Certificate*.
- **encoding** – *PEM* or *DER*.

Returns bytes
The serialized PKCS7 data.

class cryptography.hazmat.primitives.serialization.pkcs7.**PKCS7SignatureBuilder**

The PKCS7 signature builder can create both basic PKCS7 signed messages as well as S/MIME messages, which are commonly used in email. S/MIME has multiple versions, most of this implementation follows S/MIME 2.0 (RFC 2311). Only a small subset of **RFC 2632**, also known as S/MIME version 3 is supported.

Added in version 3.2.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import hashes, serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> cert = x509.load_pem_x509_certificate(ca_cert)
>>> key = serialization.load_pem_private_key(ca_key, None)
>>> options = [pkcs7.PKCS7Options.DetachedSignature]
>>> pkcs7.PKCS7SignatureBuilder().set_data(
...     b"data to sign"
... ).add_signer(
...     cert, key, hashes.SHA256()
... ).sign(
...     serialization.Encoding.SMIME, options
... )
b'...'
```

set_data(*data*)

Parameters
data (*bytes-like*) – The data to be hashed and signed.

add_signer(*certificate*, *private_key*, *hash_algorithm*, *, *rsa_padding=None*)

Parameters

- **certificate** – The *Certificate*.
- **private_key** – The *RSAPrivateKey* or *EllipticCurvePrivateKey* associated with the certificate provided (matches *PKCS7PrivateKeyTypes*).

- **hash_algorithm** – The *HashAlgorithm* that will be used to generate the signature. This must be one of the types in *PKCS7HashTypes*.
- **rsa_padding** – Added in version 42.0.0.

  This is a keyword-only argument. If `private_key` is an `RSAPrivateKey` then this can be set to either *PKCS1v15* or *PSS* to sign with those respective paddings. If this is `None` then RSA keys will default to `PKCS1v15` padding. All other key types **must** not pass a value other than `None`.

add_certificate(*certificate*)

Add an additional certificate (typically used to help build a verification chain) to the PKCS7 structure. This method may be called multiple times to add as many certificates as desired.

Parameters
certificate – The *Certificate* to add.

sign(*encoding*, *options*)

Parameters

- **encoding** – *PEM*, *DER*, or *SMIME*.
- **options** – A list of *PKCS7Options*.

Returns bytes
The signed PKCS7 message.

class cryptography.hazmat.primitives.serialization.pkcs7.**PKCS7EnvelopeBuilder**

The PKCS7 envelope builder can create encrypted S/MIME messages, which are commonly used in email. S/MIME has multiple versions, most of this implementation follows S/MIME 2.0 (RFC 2311). Only a small subset of **RFC 2632**, also known as S/MIME version 3 is supported.

Added in version 43.0.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> from cryptography.hazmat.primitives.ciphers import algorithms
>>> cert = x509.load_pem_x509_certificate(ca_cert_rsa)
>>> options = [pkcs7.PKCS7Options.Text]
>>> pkcs7.PKCS7EnvelopeBuilder().set_data(
...     b"data to encrypt"
... ).set_content_encryption_algorithm(
...     algorithms.AES128
... ).add_recipient(
...     cert
... ).encrypt(
...     serialization.Encoding.SMIME, options
... )
b'...'
```

set_data(*data*)

Parameters
data (*bytes-like*) – The data to be encrypted.

set_content_encryption_algorithm(*content_encryption_algorithm*)

Parameters

content_encryption_algorithm (*AES128* or *AES256*) – the content encryption algorithm to use. Only AES is supported, with a key size of 128 or 256 bits.

add_recipient(*certificate*)

Add a recipient for the message. Recipients will be able to use their private keys to decrypt the message. This method may be called multiple times to add as many recipients as desired.

Parameters

certificate – A *Certificate* for an intended recipient of the encrypted message. Only certificates with public RSA keys are currently supported.

encrypt(*encoding*, *options*)

The message is encrypted using AES-128-CBC. The encryption key used is included in the envelope, encrypted using the recipient's public RSA key. If multiple recipients are specified, the key is encrypted once with each recipient's public key, and all encrypted keys are included in the envelope (one per recipient).

Parameters

- **encoding** – *PEM*, *DER*, or *SMIME*.
- **options** – A list of *PKCS7Options*. For this operation only *Text* and *Binary* are supported, and cannot be used at the same time.

Returns bytes

The enveloped PKCS7 message.

cryptography.hazmat.primitives.serialization.pkcs7.**pkcs7_decrypt_der**(*data*, *certificate*, *private_key*, *options*)

Added in version 44.0.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> cert = x509.load_pem_x509_certificate(ca_cert_rsa)
>>> key = serialization.load_pem_private_key(ca_key_rsa, None)
>>> options = [pkcs7.PKCS7Options.Text]
>>> enveloped = pkcs7.PKCS7EnvelopeBuilder().set_data(
...     b"data to encrypt"
... ).add_recipient(
...     cert
... ).encrypt(
...     serialization.Encoding.DER, options
... )
>>> pkcs7.pkcs7_decrypt_der(enveloped, cert, key, options)
b'data to encrypt'
```

Deserialize and decrypt a DER-encoded PKCS7 message. PKCS7 (or S/MIME) has multiple versions, most of this implementation follows S/MIME 2.0 (RFC 2311). A small subset of **RFC 2632**, also known as S/MIME version 3 is supported.

Warning

`EnvelopedData` does not authenticate its content. An application that decrypts attacker-supplied messages and reveals whether decryption succeeded – through an error, a status code, or timing – gives the attacker

a CBC padding oracle sufficient to recover the plaintext. This is a property of PKCS#7 and cannot be fixed here, so avoid decrypting untrusted `EnvelopedData`.

Parameters

- **data** (*bytes*) – The data, encoded in DER format.
- **certificate** – A *Certificate* for an intended recipient of the encrypted message. Only certificates with public RSA keys are currently supported.
- **private_key** – The *RSAPrivateKey* associated with the certificate provided. Only private RSA keys are supported.
- **options** – A list of *PKCS7Options*. For this operation only *Text* is supported. Passing an empty list will allow decryption of binary data.

Returns bytes

The decrypted message.

Raises

- **ValueError** – If the recipient certificate does not match any of the encrypted keys in the PKCS7 data.
- *cryptography.exceptions.UnsupportedAlgorithm* – If any of the PKCS7 keys are encrypted with another algorithm than RSA with PKCS1 v1.5 padding.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the content is encrypted with another algorithm than AES (with key sizes 128 and 256), with CBC mode.
- **ValueError** – If the PKCS7 data does not contain encrypted content.
- **ValueError** – If the PKCS7 data is not of the enveloped data type.

`cryptography.hazmat.primitives.serialization.pkcs7.`**`pkcs7_decrypt_pem`**(*data*, *certificate*, *private_key*, *options*)

Added in version 44.0.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> cert = x509.load_pem_x509_certificate(ca_cert_rsa)
>>> key = serialization.load_pem_private_key(ca_key_rsa, None)
>>> options = [pkcs7.PKCS7Options.Text]
>>> enveloped = pkcs7.PKCS7EnvelopeBuilder().set_data(
...     b"data to encrypt"
... ).add_recipient(
...     cert
... ).encrypt(
...     serialization.Encoding.PEM, options
... )
>>> pkcs7.pkcs7_decrypt_pem(enveloped, cert, key, options)
b'data to encrypt'
```

Deserialize and decrypt a PEM-encoded PKCS7 message. PKCS7 (or S/MIME) has multiple versions, most of this implementation follows S/MIME 2.0 (RFC 2311). A small subset of **RFC 2632**, also known as S/MIME version 3 is supported.

> **Warning**
>
> The padding oracle described in *pkcs7_decrypt_der()* applies equally here.

Parameters

- **data** (*bytes*) – The data, encoded in PEM format.
- **certificate** – A *Certificate* for an intended recipient of the encrypted message. Only certificates with public RSA keys are currently supported.
- **private_key** – The *RSAPrivateKey* associated with the certificate provided. Only private RSA keys are supported.
- **options** – A list of *PKCS7Options*. For this operation only *Text* is supported.

Returns bytes

The decrypted message.

Raises

- **ValueError** – If the PEM data does not have the PKCS7 tag.
- **ValueError** – If the recipient certificate does not match any of the encrypted keys in the PKCS7 data.
- *cryptography.exceptions.UnsupportedAlgorithm* – If any of the PKCS7 keys are encrypted with another algorithm than RSA with PKCS1 v1.5 padding.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the content is encrypted with another algorithm than AES (with key sizes 128 and 256), with CBC mode.
- **ValueError** – If the PKCS7 data does not contain encrypted content.
- **ValueError** – If the PKCS7 data is not of the enveloped data type.

cryptography.hazmat.primitives.serialization.pkcs7.**pkcs7_decrypt_smime**(*data*, *certificate*, *private_key*, *options*)

Added in version 44.0.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.serialization import pkcs7
>>> cert = x509.load_pem_x509_certificate(ca_cert_rsa)
>>> key = serialization.load_pem_private_key(ca_key_rsa, None)
>>> options = [pkcs7.PKCS7Options.Text]
>>> enveloped = pkcs7.PKCS7EnvelopeBuilder().set_data(
...     b"data to encrypt"
... ).add_recipient(
...     cert
... ).encrypt(
...     serialization.Encoding.SMIME, options
... )
>>> pkcs7.pkcs7_decrypt_smime(enveloped, cert, key, options)
b'data to encrypt'
```

Deserialize and decrypt a S/MIME-encoded PKCS7 message. PKCS7 (or S/MIME) has multiple versions, most of this implementation follows S/MIME 2.0 (RFC 2311). A small subset of **RFC 2632**, also known as S/MIME version 3 is supported.

> **Warning**
>
> The padding oracle described in *pkcs7_decrypt_der()* applies equally here.

Parameters

- **data** (*bytes*) – The data. It should be in S/MIME format, meaning MIME with content type `application/pkcs7-mime` or `application/x-pkcs7-mime`.
- **certificate** – A *Certificate* for an intended recipient of the encrypted message. Only certificates with public RSA keys are currently supported.
- **private_key** – The *RSAPrivateKey* associated with the certificate provided. Only private RSA keys are supported.
- **options** – A list of *PKCS7Options*. For this operation only *Text* is supported.

Returns bytes

The decrypted message.

Raises

- **ValueError** – If the S/MIME data is not one of the correct content types.
- **ValueError** – If the recipient certificate does not match any of the encrypted keys in the PKCS7 data.
- *cryptography.exceptions.UnsupportedAlgorithm* – If any of the PKCS7 keys are encrypted with another algorithm than RSA with PKCS1 v1.5 padding.
- *cryptography.exceptions.UnsupportedAlgorithm* – If the content is encrypted with another algorithm than AES (with key sizes 128 and 256), with CBC mode.
- **ValueError** – If the PKCS7 data does not contain encrypted content.
- **ValueError** – If the PKCS7 data is not of the enveloped data type.

class `cryptography.hazmat.primitives.serialization.pkcs7.`**PKCS7Options**

Added in version 3.2.

An enumeration of options for PKCS7 signature, envelope creation, and decryption.

Text

For signing, the text option adds `text/plain` headers to an S/MIME message when serializing to *SMIME*. This option is disallowed with `DER` serialization. For envelope creation, it adds `text/plain` headers to the encrypted content, regardless of the specified encoding. For envelope decryption, it parses the decrypted content headers (if any), checks if the content type is 'text/plain', then removes all headers (keeping only the payload) of this decrypted content. If there is no header, or the content type is not "text/plain", it raises an error.

Binary

Signature and envelope creation normally converts line endings (LF to CRLF). When passing this option, the data will not be converted.

DetachedSignature

Don't embed the signed data within the ASN.1. When signing with *SMIME* this also results in the data being added as clear text before the PEM encoded structure.

NoCapabilities

PKCS7 structures contain a `MIMECapabilities` section inside the `authenticatedAttributes`. Passing this as an option removes `MIMECapabilities`.

NoAttributes

PKCS7 structures contain an `authenticatedAttributes` section. Passing this as an option removes that section. Note that if you pass `NoAttributes` you can't pass `NoCapabilities` since `NoAttributes` removes `MIMECapabilities` and more.

NoCerts

Don't include the signer's certificate in the PKCS7 structure. This can reduce the size of the signature but requires that the recipient can obtain the signer's certificate by other means (for example from a previously signed message).

Serialization Formats

class `cryptography.hazmat.primitives.serialization.`**PrivateFormat**

Added in version 0.8.

An enumeration for private key formats. Used with the `private_bytes` method available on *RSAPrivateKey* , *EllipticCurvePrivateKey* , *DHPrivateKey* and *DSAPrivateKey*.

TraditionalOpenSSL

Frequently known as PKCS#1 format. Still a widely used format, but generally considered legacy.

A PEM encoded RSA key will look like:

```
-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----
```

PKCS8

A more modern format for serializing keys which allows for better encryption. Choose this unless you have explicit legacy compatibility requirements.

A PEM encoded key will look like:

```
-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----
```

Raw

Added in version 2.5.

A raw format used by *Ed25519 signing*, *Ed448 signing*, *X25519 key exchange*, and *X448 key exchange*. It is a binary format and is invalid for other key types.

OpenSSH

Added in version 3.0.

Custom private key format for OpenSSH, internals are based on SSH protocol and not ASN1. Requires *PEM* encoding.

A PEM encoded OpenSSH key will look like:

```
-----BEGIN OPENSSH PRIVATE KEY-----
...
-----END OPENSSH PRIVATE KEY-----
```

PKCS12

Added in version 38.0.0.

The PKCS#12 format is a binary format used to store private keys and certificates. This attribute is used in conjunction with `encryption_builder()` to allow control of the encryption algorithm and parameters.

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.serialization import PrivateFormat,␣
↪pkcs12
>>> encryption = (
...     PrivateFormat.PKCS12.encryption_builder().
...     kdf_rounds(50000).
...     key_cert_algorithm(pkcs12.PBES.PBESv2SHA256AndAES256CBC).
...     hmac_hash(hashes.SHA256()).build(b"my password")
... )
>>> p12 = pkcs12.serialize_key_and_certificates(
...     b"friendlyname", key, None, None, encryption
... )
```

encryption_builder()

Added in version 38.0.0.

Returns a builder for configuring how values are encrypted with this format. You must call this method on an element of the enumeration. For example, `PrivateFormat.OpenSSH.encryption_builder()`.

For most use cases, *BestAvailableEncryption* is preferred.

Returns

A new instance of *KeySerializationEncryptionBuilder*

```
>>> from cryptography.hazmat.primitives import serialization
>>> encryption = (
...     serialization.PrivateFormat.OpenSSH.encryption_builder().kdf_rounds(30).
↪build(b"my password")
... )
>>> key.private_bytes(
...     encoding=serialization.Encoding.PEM,
...     format=serialization.PrivateFormat.OpenSSH,
...     encryption_algorithm=encryption
... )
b'-----BEGIN OPENSSH PRIVATE KEY-----\n...\n-----END OPENSSH PRIVATE KEY-----\n'
```

class cryptography.hazmat.primitives.serialization.**PublicFormat**

Added in version 0.8.

An enumeration for public key formats. Used with the `public_bytes` method available on *RSAPublicKey* , *EllipticCurvePublicKey* , *DHPublicKey* , and *DSAPublicKey*.

SubjectPublicKeyInfo

This is the typical public key format. It consists of an algorithm identifier and the public key as a bit string. Choose this unless you have specific needs.

A PEM encoded key will look like:

```
-----BEGIN PUBLIC KEY-----
...
-----END PUBLIC KEY-----
```

PKCS1

Just the public key elements (without the algorithm identifier). This format is RSA only, but is used by some older systems.

A PEM encoded key will look like:

```
-----BEGIN RSA PUBLIC KEY-----
...
-----END RSA PUBLIC KEY-----
```

OpenSSH

Added in version 1.4.

The public key format used by OpenSSH (e.g. as found in `~/.ssh/id_rsa.pub` or `~/.ssh/authorized_keys`).

Raw

Added in version 2.5.

A raw format used by *Ed25519 signing*, *Ed448 signing*, *X25519 key exchange*, and *X448 key exchange*. It is a binary format and is invalid for other key types.

CompressedPoint

Added in version 2.5.

A compressed elliptic curve public key as defined in ANSI X9.62 section 4.3.6 (as well as SEC 1 v2.0).

UncompressedPoint

Added in version 2.5.

An uncompressed elliptic curve public key as defined in ANSI X9.62 section 4.3.6 (as well as SEC 1 v2.0).

class `cryptography.hazmat.primitives.serialization.`**ParameterFormat**

Added in version 2.0.

An enumeration for parameters formats. Used with the `parameter_bytes` method available on *`DHParameters`*.

PKCS3

ASN1 DH parameters sequence as defined in PKCS3.

Serialization Encodings

class `cryptography.hazmat.primitives.serialization.`**Encoding**

An enumeration for encoding types. Used with the `private_bytes` method available on *`RSAPrivateKey`*, *`EllipticCurvePrivateKey`*, *`DHPrivateKey`*, *`DSAPrivateKey`*, and *`X448PrivateKey`* as well as `public_bytes` on *`RSAPublicKey`*, *`DHPublicKey`*, *`EllipticCurvePublicKey`*, and *`X448PublicKey`*.

PEM

Added in version 0.8.

For PEM format. This is a base64 format with delimiters.

DER

Added in version 0.9.

For DER format. This is a binary format.

OpenSSH

Added in version 1.4.

The format used by OpenSSH public keys. This is a text format.

Raw

Added in version 2.5.

A raw format used by *Ed25519 signing*, *Ed448 signing*, *X25519 key exchange*, and *X448 key exchange*. It is a binary format and is invalid for other key types.

X962

Added in version 2.5.

The format used by elliptic curve point encodings. This is a binary format.

SMIME

Added in version 3.2.

An output format used for PKCS7. This is a text format.

Serialization Encryption Types

class cryptography.hazmat.primitives.serialization.**KeySerializationEncryption**

Objects with this interface are usable as encryption types with methods like `private_bytes` available on `RSAPrivateKey` , `EllipticCurvePrivateKey` , `DHPrivateKey` and `DSAPrivateKey`. All other classes in this section represent the available choices for encryption and have this interface.

class cryptography.hazmat.primitives.serialization.**BestAvailableEncryption**(*password*)

Encrypt using the best available encryption for a given key. This is a curated encryption choice and the algorithm may change over time. The encryption algorithm may vary based on which version of OpenSSL the library is compiled against.

Parameters

password (*bytes*) – The password to use for encryption.

class cryptography.hazmat.primitives.serialization.**NoEncryption**

Do not encrypt.

class cryptography.hazmat.primitives.serialization.**KeySerializationEncryptionBuilder**

Added in version 38.0.0.

A builder that can be used to configure how data is encrypted. To create one, call `PrivateFormat.encryption_builder()`. Different serialization types will support different options on this builder.

kdf_rounds(*rounds*)

Set the number of rounds the Key Derivation Function should use. The meaning of the number of rounds varies on the KDF being used.

Parameters

rounds (*int*) – Number of rounds.

key_cert_algorithm(*algorithm*)

Set the encryption algorithm to use when encrypting the key and certificate in a PKCS12 structure.

Parameters

algorithm – A value from the *PBES* enumeration.

hmac_hash(*algorithm*)

Set the hash algorithm to use within the MAC for a PKCS12 structure.

Parameters
algorithm – An instance of a *HashAlgorithm*

build(*password*)

Turns the builder into an instance of *KeySerializationEncryption* with a given password.

Parameters
password (*bytes*) – The password.

Returns
A *KeySerializationEncryption* encryption object that can be passed to methods like `private_bytes` or *serialize_key_and_certificates()*.

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.

Asymmetric Utilities

cryptography.hazmat.primitives.asymmetric.utils.**decode_dss_signature**(*signature*)

Takes in signatures generated by the DSA/ECDSA signers and returns a tuple `(r, s)`. These signatures are ASN.1 encoded `Dss-Sig-Value` sequences (as defined in **RFC 3279**)

Parameters
signature (*bytes*) – The signature to decode.

Returns
The decoded tuple `(r, s)`.

Raises
ValueError – Raised if the signature is malformed.

cryptography.hazmat.primitives.asymmetric.utils.**encode_dss_signature**(*r*, *s*)

Creates an ASN.1 encoded `Dss-Sig-Value` (as defined in **RFC 3279**) from raw `r` and `s` values.

Parameters

- **r** (*int*) – The raw signature value `r`.
- **s** (*int*) – The raw signature value `s`.

Return bytes
The encoded signature.

class cryptography.hazmat.primitives.asymmetric.utils.**NoDigestInfo**

Added in version 47.0.0.

Use a non-standard RSA signature formats where the PKCS #1-padded data is without DigestInfo.

`NoDigestInfo` can be passed as the `algorithm` in the RSA *sign()*, *verify()* and *recover_data_from_signature()* methods.

class cryptography.hazmat.primitives.asymmetric.utils.**Prehashed**(*algorithm*)

Added in version 1.6.

`Prehashed` can be passed as the `algorithm` in the RSA *sign()* and *verify()* as well as DSA *sign()* and *verify()* methods.

For elliptic curves it can be passed as the `algorithm` in *ECDSA* and then used with *sign()* and *verify()* .

Parameters

algorithm – An instance of *HashAlgorithm*.

```
>>> import hashlib
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import (
...    padding, rsa, utils
... )
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> prehashed_msg = hashlib.sha256(b"A message I want to sign").digest()
>>> signature = private_key.sign(
...     prehashed_msg,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(hashes.SHA256())
... )
>>> public_key = private_key.public_key()
>>> public_key.verify(
...     signature,
...     prehashed_msg,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(hashes.SHA256())
... )
```

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.

Cloud KMS and HSM Asymmetric Keys

`cryptography` provides a set of abstract base classes for asymmetric keys that can be used to integrate with cloud key management services, HSMs, and other ways of managing keys that are not in-memory. A minimal example with a hypothetical cloud key management service for an RSA key is provided below, but this works for all asymmetric types. You must provide all methods of the base class, but many methods can be stubs with no implementation if you only need a subset of functionality.

```
>>> import typing
>>> from cryptography.hazmat.primitives.asymmetric import rsa, utils
>>> from cryptography.hazmat.primitives import hashes, serialization
```

(continues on next page)

(continued from previous page)

```
>>> from cryptography.hazmat.primitives.asymmetric.padding import AsymmetricPadding,␣
↪PKCS1v15
>>>
>>> class CloudRSAPrivateKey(rsa.RSAPrivateKey):
...     def __init__(self, creds, key_id):
...         self._creds = creds
...         self._cloud_client = SomeCloudClient(creds)
...         self._key_id = key_id
...
...     def sign(
...         self,
...         data: bytes,
...         padding: AsymmetricPadding,
...         algorithm: typing.Union[utils.Prehashed, hashes.HashAlgorithm],
...     ) -> bytes:
...         """
...         Signs data using the cloud KMS. You'll need to define a mapping
...         between the way your cloud provider represents padding and algorithms
...         and the way cryptography represents them.
...         """
...
...         # Hash the data if necessary
...         if not isinstance(algorithm, utils.Prehashed):
...             h = hashes.Hash(algorithm)
...             h.update(data)
...             digest = h.finalize()
...             hash_alg = algorithm
...         else:
...             digest = data
...             hash_alg = algorithm._algorithm
...         # Map cryptography padding/algorithm to KMS signing algorithm
...         kms_algorithm = self._map_to_kms_algorithm(padding, hash_alg)
...
...         # Call KMS API to sign the digest
...         response = self._cloud_client.sign(
...             key_id=self._key_id,
...             algorithm=kms_algorithm,
...             message=digest,
...         )
...
...         return response.signature
...
...     def decrypt(self, ciphertext: bytes, padding: AsymmetricPadding) -> bytes:
...         raise NotImplementedError()
...
...     def _map_to_kms_algorithm(
...         self,
...         padding: AsymmetricPadding,
...         algorithm: hashes.HashAlgorithm
...     ) -> bytes:
...         """
...         Maps the cryptography padding and algorithm to the corresponding KMS signing␣
```

(continues on next page)

(continued from previous page)

```
↪algorithm.
...         This is specific to your implementation.
...         """
...         if padding == PKCS1v15() and algorithm == hashes.SHA256():
...             return b"RSA_PKCS1_V1_5_SHA_256"
...         else:
...             raise NotImplementedError()
...
...     @property
...     def key_size(self) -> int:
...         return self._cloud_client.key_size(self._key_id)
...
...     def public_key(self) -> rsa.RSAPublicKey:
...         raise NotImplementedError()
...
...     def private_numbers(self) -> rsa.RSAPrivateNumbers:
...         """
...         This method typically can't be implemented for cloud KMS keys
...         as the private key material is not accessible.
...         """
...         raise NotImplementedError()
...
...     def private_bytes(
...         self,
...         encoding: serialization.Encoding,
...         format: serialization.PrivateFormat,
...         encryption_algorithm: serialization.KeySerializationEncryption,
...     ) -> bytes:
...         """
...         This method typically can't be implemented for cloud KMS keys
...         as the private key material is not accessible.
...         """
...         raise NotImplementedError()
...
...     def __copy__(self) -> "CloudRSAPrivateKey":
...         return self
...
...     def __deepcopy__(self, memo: dict) -> "CloudRSAPrivateKey":
...         return self
...
>>> cloud_private_key = CloudRSAPrivateKey("creds", "key_id")
>>> sig = cloud_private_key.sign(b"message", PKCS1v15(), hashes.SHA256())
>>> isinstance(sig, bytes)
True
```

This key can then be used with other parts of `cryptography`, such as the X.509 APIs. In the example below we assume that we are using our cloud private key to sign a leaf certificate (not self-signed).

```
>>> from cryptography import x509
>>> from cryptography.x509.oid import NameOID
>>> import datetime
>>> one_day = datetime.timedelta(1, 0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> leaf_private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
>>> leaf_public_key = leaf_private_key.public_key()
>>> builder = x509.CertificateBuilder()
>>> builder = builder.subject_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'cryptography.io'),
... ]))
>>> builder = builder.issuer_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, 'My Cloud CA'),
... ]))
>>> builder = builder.not_valid_before(datetime.datetime.today() - one_day)
>>> builder = builder.not_valid_after(datetime.datetime.today() + (one_day * 30))
>>> builder = builder.serial_number(x509.random_serial_number())
>>> builder = builder.public_key(leaf_public_key)
>>> builder = builder.add_extension(
...     x509.SubjectAlternativeName(
...         [x509.DNSName('cryptography.io')]
...     ),
...     critical=False
... )
>>> builder = builder.add_extension(
...     x509.BasicConstraints(ca=False, path_length=None), critical=True,
... )
>>> certificate = builder.sign(
...     private_key=cloud_private_key, algorithm=hashes.SHA256(),
... )
>>> isinstance(certificate, x509.Certificate)
True
```

Common types

Asymmetric key types do not inherit from a common base class. The following union type aliases can be used instead to reference a multitude of key types.

`cryptography.hazmat.primitives.asymmetric.types.`**`PublicKeyTypes`**

Added in version 40.0.0.

Type alias: A union of all public key types supported: *DHPublicKey*, *RSAPublicKey*, *DSAPublicKey*, *EllipticCurvePublicKey*, *Ed25519PublicKey*, *Ed448PublicKey*, *X25519PublicKey*, *X448PublicKey*.

`cryptography.hazmat.primitives.asymmetric.types.`**`PrivateKeyTypes`**

Added in version 40.0.0.

Type alias: A union of all private key types supported: *DHPrivateKey*, *RSAPrivateKey*, *DSAPrivateKey*, *EllipticCurvePrivateKey*, *Ed25519PrivateKey*, *Ed448PrivateKey*, *X25519PrivateKey*, *X448PrivateKey*.

`cryptography.hazmat.primitives.asymmetric.types.`**`CertificatePublicKeyTypes`**

Added in version 40.0.0.

Type alias: A union of all public key types supported for X.509 certificates: *RSAPublicKey*, *DSAPublicKey*, *EllipticCurvePublicKey*, *Ed25519PublicKey*, *Ed448PublicKey*, *X25519PublicKey*, *X448PublicKey*.

cryptography.hazmat.primitives.asymmetric.types.**CertificateIssuerPublicKeyTypes**

Added in version 40.0.0.

Type alias: A union of all public key types that can sign other X.509 certificates as an issuer. x448/x25519 can be a public key, but cannot be used in signing, so they are not allowed in these contexts.

Allowed: *RSAPublicKey*, *DSAPublicKey*, *EllipticCurvePublicKey*, *Ed25519PublicKey*, *Ed448PublicKey*.

cryptography.hazmat.primitives.asymmetric.types.**CertificateIssuerPrivateKeyTypes**

Added in version 40.0.0.

Type alias: A union of all private key types that can sign other X.509 certificates as an issuer. x448/x25519 can be a public key, but cannot be used in signing, so they are not allowed in these contexts.

Allowed: *RSAPrivateKey*, *DSAPrivateKey*, *EllipticCurvePrivateKey*, *Ed25519PrivateKey*, *Ed448PrivateKey*.

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.

2.4.3 HPKE (Hybrid Public Key Encryption)

HPKE is a standard for public key encryption that combines a Key Encapsulation Mechanism (KEM), a Key Derivation Function (KDF), and an Authenticated Encryption with Associated Data (AEAD) scheme. It is defined in **RFC 9180**.

HPKE provides authenticated encryption: the recipient can be certain that the message was encrypted by someone who knows the recipient's public key, but the sender is anonymous. Each call to *Suite.encrypt()* generates a fresh ephemeral key pair, so encrypting the same plaintext twice will produce different ciphertext.

The `info` parameter should be used to bind the encryption to a specific context (e.g., "MyApp-v1-UserMessages"). Per **RFC 9180 Section 8.1**, applications using single-shot APIs should use the `info` parameter for specifying auxiliary authenticated information.

```
from cryptography.hazmat.primitives.hpke import Suite, KEM, KDF, AEAD
from cryptography.hazmat.primitives.asymmetric import x25519

suite = Suite(KEM.X25519, KDF.HKDF_SHA256, AEAD.AES_128_GCM)

# Generate recipient key pair
private_key = x25519.X25519PrivateKey.generate()
public_key = private_key.public_key()

# Encrypt
ciphertext = suite.encrypt(b"secret message", public_key, info=b"app info")

# Decrypt
plaintext = suite.decrypt(ciphertext, private_key, info=b"app info")
```

class cryptography.hazmat.primitives.hpke.**Suite**(*kem*, *kdf*, *aead*)

An HPKE cipher suite combining a KEM, KDF, and AEAD.

Parameters

- **kem** (*KEM*) – The key encapsulation mechanism.

- **kdf** (*KDF*) – The key derivation function.
- **aead** (*AEAD*) – The authenticated encryption algorithm.

encrypt(*plaintext*, *public_key*, *info=b''*)

Encrypt a message using HPKE.

Parameters

- **plaintext** (*bytes*) – The message to encrypt.
- **public_key** – The recipient's public key. The type must match the *KEM* chosen for the suite: *X25519PublicKey* for *KEM.X25519*; *EllipticCurvePublicKey* on the corresponding curve for *KEM.P256*, *KEM.P384*, and *KEM.P521*; *MLKEM768PublicKey* for *KEM.MLKEM768*; *MLKEM1024PublicKey* for *KEM.MLKEM1024*; *MLKEM768X25519PublicKey* for *KEM.MLKEM768_X25519*; and *MLKEM1024P384PublicKey* for *KEM.MLKEM1024_P384*.
- **info** (*bytes*) – Application-specific context string for binding the encryption to a specific application or protocol.

Returns

The encapsulated key concatenated with ciphertext (enc || ct).

Return type

bytes

decrypt(*ciphertext*, *private_key*, *info=b''*)

Decrypt a message using HPKE.

Parameters

- **ciphertext** (*bytes*) – The enc || ct value from *encrypt()*.
- **private_key** – The recipient's private key. The type must match the *KEM* chosen for the suite: *X25519PrivateKey* for *KEM.X25519*; *EllipticCurvePrivateKey* on the corresponding curve for *KEM.P256*, *KEM.P384*, and *KEM.P521*; *MLKEM768PrivateKey* for *KEM.MLKEM768*; *MLKEM1024PrivateKey* for *KEM.MLKEM1024*; *MLKEM768X25519PrivateKey* for *KEM.MLKEM768_X25519*; and *MLKEM1024P384PrivateKey* for *KEM.MLKEM1024_P384*.
- **info** (*bytes*) – Application-specific context string (must match the value used during encryption).

Returns

The decrypted plaintext.

Return type

bytes

Raises

cryptography.exceptions.InvalidTag – If decryption fails.

class cryptography.hazmat.primitives.hpke.**KEM**

An enumeration of key encapsulation mechanisms.

enc_length()

Added in version 49.0.0.

Returns

The length in bytes of the encapsulated key (enc) produced by this KEM. The enc is the prefix of the value returned by *Suite.encrypt()*, so this can be used to split the result into enc and the AEAD ciphertext:

```
ciphertext = suite.encrypt(plaintext, public_key)
enc_len = KEM.X25519.enc_length()
enc, ct = ciphertext[:enc_len], ciphertext[enc_len:]
```

Return type
int

X25519
DHKEM(X25519, HKDF-SHA256)

P256
DHKEM(P-256, HKDF-SHA256)

P384
DHKEM(P-384, HKDF-SHA384)

P521
DHKEM(P-521, HKDF-SHA512)

MLKEM768
ML-KEM-768. Post-quantum secure. Only available on backends that support ML-KEM.

MLKEM1024
ML-KEM-1024. Post-quantum secure. Only available on backends that support ML-KEM.

MLKEM768_X25519
A hybrid KEM combining ML-KEM-768 with X25519 (also known as X-Wing). Post-quantum secure. Only available on backends that support ML-KEM. Public and private keys are *MLKEM768X25519PublicKey* and *MLKEM768X25519PrivateKey*.

MLKEM1024_P384
A hybrid KEM combining ML-KEM-1024 with P-384. Post-quantum secure. Only available on backends that support ML-KEM. Public and private keys are *MLKEM1024P384PublicKey* and *MLKEM1024P384PrivateKey*.

class cryptography.hazmat.primitives.hpke.**MLKEM768X25519PrivateKey**(*mlkem_key*, *x25519_key*)

Added in version 47.0.0.

A hybrid ML-KEM-768 / X25519 private key for use with *KEM.MLKEM768_X25519*. Combines an *MLKEM768PrivateKey* and an *X25519PrivateKey* into a single recipient key.

Parameters

- **mlkem_key** (*MLKEM768PrivateKey*) – The ML-KEM-768 private key component.
- **x25519_key** (*X25519PrivateKey*) – The X25519 private key component.

public_key()

Returns
MLKEM768X25519PublicKey

class cryptography.hazmat.primitives.hpke.**MLKEM768X25519PublicKey**(*mlkem_key*, *x25519_key*)

Added in version 47.0.0.

A hybrid ML-KEM-768 / X25519 public key for use with *KEM.MLKEM768_X25519*. Combines an *MLKEM768PublicKey* and an *X25519PublicKey* into a single recipient key.

Parameters

- **mlkem_key** (*MLKEM768PublicKey*) – The ML-KEM-768 public key component.
- **x25519_key** (*X25519PublicKey*) – The X25519 public key component.

class cryptography.hazmat.primitives.hpke.**MLKEM1024P384PrivateKey**(*mlkem_key*, *p384_key*)

Added in version 47.0.0.

A hybrid ML-KEM-1024 / P-384 private key for use with *KEM.MLKEM1024_P384*. Combines an *MLKEM1024PrivateKey* and an *EllipticCurvePrivateKey* on the SECP384R1 curve into a single recipient key.

Parameters

- **mlkem_key** (*MLKEM1024PrivateKey*) – The ML-KEM-1024 private key component.
- **p384_key** (*EllipticCurvePrivateKey*) – The P-384 private key component.

public_key()

Returns
MLKEM1024P384PublicKey

class cryptography.hazmat.primitives.hpke.**MLKEM1024P384PublicKey**(*mlkem_key*, *p384_key*)

Added in version 47.0.0.

A hybrid ML-KEM-1024 / P-384 public key for use with *KEM.MLKEM1024_P384*. Combines an *MLKEM1024PublicKey* and an *EllipticCurvePublicKey* on the SECP384R1 curve into a single recipient key.

Parameters

- **mlkem_key** (*MLKEM1024PublicKey*) – The ML-KEM-1024 public key component.
- **p384_key** (*EllipticCurvePublicKey*) – The P-384 public key component.

class cryptography.hazmat.primitives.hpke.**KDF**

An enumeration of key derivation functions.

HKDF_SHA256

HKDF-SHA256

HKDF_SHA384

HKDF-SHA384

HKDF_SHA512

HKDF-SHA512

SHAKE128

SHAKE-128

SHAKE256

SHAKE-256

class cryptography.hazmat.primitives.hpke.**AEAD**

An enumeration of authenticated encryption algorithms.

AES_128_GCM

AES-128-GCM

AES_256_GCM
AES-256-GCM

CHACHA20_POLY1305
ChaCha20Poly1305

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.

2.4.4 Constant time functions

This module contains functions for operating with secret data in a way that does not leak information about that data through how long it takes to perform the operation. These functions should be used whenever operating on secret data along with data that is user supplied.

An example would be comparing a HMAC signature received from a client to the one generated by the server code for authentication purposes.

For more information about this sort of issue, Wikipedia has a section about Constant-time challenges.

`cryptography.hazmat.primitives.constant_time.bytes_eq(a, b)`

Compares `a` and `b` with one another. If `a` and `b` have different lengths, this returns `False` immediately. Otherwise it compares them in a way that takes the same amount of time, regardless of how many characters are the same between the two.

```
>>> from cryptography.hazmat.primitives import constant_time
>>> constant_time.bytes_eq(b"foo", b"foo")
True
>>> constant_time.bytes_eq(b"foo", b"bar")
False
```

Parameters

- **a** (*bytes*) – The left-hand side.
- **b** (*bytes*) – The right-hand side.

Returns bool
`True` if `a` has the same bytes as `b`, otherwise `False`.

Raises
`TypeError` – This exception is raised if `a` or `b` is not `bytes`.

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.

2.4.5 Key derivation functions

Key derivation functions derive bytes suitable for cryptographic operations from passwords or other data sources using a pseudo-random function (PRF). Different KDFs are suitable for different tasks such as:

- Cryptographic key derivation

  Deriving a key suitable for use as input to an encryption algorithm. Typically this means taking a password and running it through an algorithm such as *PBKDF2HMAC* or *HKDF*. This process is typically known as key stretching.

- Password storage

  When storing passwords you want to use an algorithm that is computationally intensive. Legitimate users will only need to compute it once (for example, taking the user's password, running it through the KDF, then comparing it to the stored value), while attackers will need to do it billions of times. Ideal password storage KDFs will be demanding on both computational and memory resources.

Variable cost algorithms

Argon2 Family

The Argon2 family of key derivation functions are designed for password storage and is described in **RFC 9106**. It consists of three variants that differ only how they access an internal memory buffer, which leads to different trade-offs in resistance to hardware attacks.

Each of the classes constructors and parameters are the same; only details of Argon2id are defined before, for brevity.

class `cryptography.hazmat.primitives.kdf.argon2.`**`Argon2d`**(*, *salt*, *length*, *iterations*, *lanes*, *memory_cost*, *ad=None*, *secret=None*)

Added in version 47.0.0.

This variant of the Argon2 family maximizes resistance to time-memory-trade-off attacks, but introduces possible side-channels

class `cryptography.hazmat.primitives.kdf.argon2.`**`Argon2i`**(*, *salt*, *length*, *iterations*, *lanes*, *memory_cost*, *ad=None*, *secret=None*)

Added in version 47.0.0.

This variant of the Argon2 family resists side-channel attacks, but is vulnerable to time-memory-trade-off attacks

class `cryptography.hazmat.primitives.kdf.argon2.`**`Argon2id`**(*, *salt*, *length*, *iterations*, *lanes*, *memory_cost*, *ad=None*, *secret=None*)

Added in version 44.0.0.

Argon2id is a blend of the previous two variants. Argon2id should be used by most users, as recommended in **RFC 9106**.

This class conforms to the *KeyDerivationFunction* interface.

```
>>> import os
>>> from cryptography.hazmat.primitives.kdf.argon2 import Argon2id
>>> salt = os.urandom(16)
```

(continues on next page)

(continued from previous page)

```
>>> # derive
>>> kdf = Argon2id(
...     salt=salt,
...     length=32,
...     iterations=1,
...     lanes=4,
...     memory_cost=64 * 1024,
...     ad=None,
...     secret=None,
... )
>>> key = kdf.derive(b"my great password")
>>> # verify
>>> kdf = Argon2id(
...     salt=salt,
...     length=32,
...     iterations=1,
...     lanes=4,
...     memory_cost=64 * 1024,
...     ad=None,
...     secret=None,
... )
>>> kdf.verify(b"my great password", key)
```

All arguments to the constructor are keyword-only.

Parameters

- **salt** (*bytes*) – A salt should be unique (and randomly generated) per password and is recommended to be 16 bytes or longer
- **length** (*int*) – The desired length of the derived key in bytes.
- **iterations** (*int*) – Also known as passes, this is used to tune the running time independently of the memory size.
- **lanes** (*int*) – The number of lanes (parallel threads) to use. Also known as parallelism.
- **memory_cost** (*int*) – The amount of memory to use in kibibytes. 1 kibibyte (KiB) is 1024 bytes. This must be at minimum `8 * lanes`.
- **ad** (*bytes*) – Optional associated data.
- **secret** (*bytes*) – Optional secret data; used for keyed hashing.

RFC 9106 has recommendations for parameter choice.

Raises

cryptography.exceptions.UnsupportedAlgorithm – If Argon2id is not supported by the OpenSSL version `cryptography` is using.

derive(*key_material*)

Parameters

key_material (*bytes-like*) – The input key material.

Return bytes

the derived key.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- **MemoryError** – This exception is raised if there is insufficient memory to derive the key with the chosen `memory_cost`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This generates and returns a new key from the supplied password.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int

The number of bytes written to the buffer.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- **ValueError** – This exception is raised if the buffer is too small for the derived key.
- **MemoryError** – This exception is raised if there is insufficient memory to derive the key with the chosen `memory_cost`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This generates a new key from the supplied password and writes it directly into the provided buffer.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match. This can be used for checking whether the password a user provides matches the stored derived key.

derive_phc_encoded(*key_material*)

Added in version 45.0.0.

Parameters

key_material (*bytes-like*) – The input key material.

Return str
A PHC-formatted string containing the parameters, salt, and derived key.

Raises
cryptography.exceptions.AlreadyFinalized – This is raised when any method is called more than once.

This method generates and returns a new key from the supplied password, formatting the result as a string according to the Password Hashing Competition (PHC) format. The returned string includes the algorithm, all parameters, the salt, and the derived key in a standardized format: `$argon2id$v=19$m=<memory_cost>,t=<iterations>,p=<lanes>$<salt>$<key>`

This format is suitable for password storage and is compatible with other Argon2id implementations that support the PHC format.

classmethod verify_phc_encoded(*key_material*, *phc_encoded*, *secret=None*)

Added in version 45.0.0.

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive_phc_encoded()*.
- **phc_encoded** (*str*) – A PHC-formatted string as returned by *derive_phc_encoded()*.
- **secret** (*bytes*) – Optional secret data; used for keyed hashing.

Raises
cryptography.exceptions.InvalidKey – This is raised when the derived key does not match the key in the encoded string or when the format of the encoded string is invalid.

This class method verifies whether the supplied `key_material` matches the key contained in the PHC-formatted string. It extracts the parameters from the string, recomputes the key with those parameters, and compares the result to the key in the string.

This is useful for validating a password against a stored PHC-formatted hash string.

```
>>> import os
>>> from cryptography.hazmat.primitives.kdf.argon2 import Argon2id
>>> salt = os.urandom(16)
>>> # Create an Argon2id instance and derive a PHC-formatted string
>>> kdf = Argon2id(
...     salt=salt,
...     length=32,
...     iterations=1,
...     lanes=4,
...     memory_cost=64 * 1024,
... )
>>> encoded = kdf.derive_phc_encoded(b"my great password")
>>> # later, verify the password
>>> Argon2id.verify_phc_encoded(b"my great password", encoded)
```

PBKDF2

class cryptography.hazmat.primitives.kdf.pbkdf2.**PBKDF2HMAC**(*algorithm*, *length*, *salt*, *iterations*)

Added in version 0.2.

PBKDF2 (Password Based Key Derivation Function 2) is typically used for deriving a cryptographic key from a password. It may also be used for key storage, but an alternate key storage KDF such as *Scrypt* is generally

considered a better solution.

This class conforms to the *KeyDerivationFunction* interface.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
>>> # Salts should be randomly generated
>>> salt = os.urandom(16)
>>> # derive
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=1_200_000,
... )
>>> key = kdf.derive(b"my great password")
>>> # verify
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=1_200_000,
... )
>>> kdf.verify(b"my great password", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key in bytes. Maximum is (2^{32} - 1) * `algorithm.digest_size`.
- **salt** (*bytes*) – A salt. Secure values[1] are 128-bits (16 bytes) or longer and randomly generated.
- **iterations** (*int*) – The number of iterations to perform of the hash function. This can be used to control the length of time the operation takes. Higher numbers help mitigate brute force attacks against derived keys. A more detailed description can be consulted for additional information.

Raises

`TypeError` – This exception is raised if `salt` is not `bytes`.

derive(*key_material*)

Parameters

key_material (*bytes-like*) – The input key material. For PBKDF2 this should be a password.

Return bytes

the derived key.

Raises

- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

[1] See NIST SP 800-132.

- **TypeError** – This exception is raised if `key_material` is not `bytes`.

This generates and returns a new key from the supplied password.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material. For PBKDF2 this should be a password.
- **buffer** – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int

The number of bytes written to the buffer.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- **ValueError** – This exception is raised if the buffer is too small for the derived key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This generates a new key from the supplied password and writes it directly into the provided buffer.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match. This can be used for checking whether the password a user provides matches the stored derived key.

Scrypt

class cryptography.hazmat.primitives.kdf.scrypt.**Scrypt**(*salt*, *length*, *n*, *r*, *p*)

Added in version 1.6.

Scrypt is a KDF designed for password storage by Colin Percival to be resistant against hardware-assisted attackers by having a tunable memory cost. It is described in **RFC 7914**.

This class conforms to the *KeyDerivationFunction* interface.

```
>>> import os
>>> from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
>>> salt = os.urandom(16)
>>> # derive
>>> kdf = Scrypt(
...     salt=salt,
...     length=32,
...     n=2**14,
...     r=8,
...     p=1,
... )
>>> key = kdf.derive(b"my great password")
>>> # verify
>>> kdf = Scrypt(
...     salt=salt,
...     length=32,
...     n=2**14,
...     r=8,
...     p=1,
... )
>>> kdf.verify(b"my great password", key)
```

Parameters

- **salt** (*bytes*) – A salt.
- **length** (*int*) – The desired length of the derived key in bytes.
- **n** (*int*) – CPU/Memory cost parameter. It must be larger than 1 and be a power of 2.
- **r** (*int*) – Block size parameter.
- **p** (*int*) – Parallelization parameter.

The computational and memory cost of Scrypt can be adjusted by manipulating the 3 parameters: `n`, `r`, and `p`. In general, the memory cost of Scrypt is affected by the values of both `n` and `r`, while `n` also determines the number of iterations performed. `p` increases the computational cost without affecting memory usage. A more in-depth explanation of the 3 parameters can be found here.

RFC 7914 recommends values of `r=8` and `p=1` while scaling `n` to a number appropriate for your system. The scrypt paper suggests a minimum value of `n=2**14` for interactive logins (t < 100ms), or `n=2**20` for more sensitive files (t < 5s).

Raises

- *cryptography.exceptions.UnsupportedAlgorithm* – If Scrypt is not supported by the OpenSSL version `cryptography` is using.
- **TypeError** – This exception is raised if `salt` is not `bytes`.
- **ValueError** – This exception is raised if `n` is less than 2, if `n` is not a power of 2, if `r` is less than 1 or if `p` is less than 1.

derive(*key_material*)

Parameters

key_material (*bytes-like*) – The input key material.

Return bytes
the derived key.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.
- *`cryptography.exceptions.AlreadyFinalized`* – This is raised when *`derive()`*, *`derive_into()`*, or *`verify()`* is called more than once.

This generates and returns a new key from the supplied password.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** (*bytes-like*) – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int
the number of bytes written to the buffer.

Raises

- `ValueError` – This exception is raised if the buffer length does not match the specified `length`.
- `TypeError` – This exception is raised if `key_material` or `buffer` is not `bytes`.
- *`cryptography.exceptions.AlreadyFinalized`* – This is raised when *`derive()`*, *`derive_into()`*, or *`verify()`* is called more than once.

This generates a new key from the supplied password and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *`derive()`*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *`derive()`*.

Raises

- *`cryptography.exceptions.InvalidKey`* – This is raised when the derived key does not match the expected key.
- *`cryptography.exceptions.AlreadyFinalized`* – This is raised when *`derive()`*, *`derive_into()`*, or *`verify()`* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match. This can be used for checking whether the password a user provides matches the stored derived key.

Fixed cost algorithms

ConcatKDF

class cryptography.hazmat.primitives.kdf.concatkdf.**ConcatKDFHash**(*algorithm*, *length*, *otherinfo*)

Added in version 1.0.

ConcatKDFHash (Concatenation Key Derivation Function) is defined by the NIST Special Publication NIST SP 800-56Ar3 document, to be used to derive keys for use after a Key Exchange negotiation operation.

> **Warning**
>
> ConcatKDFHash should not be used for password storage.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.concatkdf import ConcatKDFHash
>>> otherinfo = b"concatkdf-example"
>>> ckdf = ConcatKDFHash(
...     algorithm=hashes.SHA256(),
...     length=32,
...     otherinfo=otherinfo,
... )
>>> key = ckdf.derive(b"input key")
>>> ckdf = ConcatKDFHash(
...     algorithm=hashes.SHA256(),
...     length=32,
...     otherinfo=otherinfo,
... )
>>> ckdf.verify(b"input key", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key in bytes. Maximum is `hashlen * (2^32 -1)`.
- **otherinfo** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.

Raises

`TypeError` – This exception is raised if `otherinfo` is not `bytes`.

derive(*key_material*)

Parameters

key_material (*bytes-like*) – The input key material.

Return bytes

The derived key.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.

- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** (*bytes-like*) – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int

the number of bytes written to the buffer.

Raises

- **ValueError** – This exception is raised if the buffer length does not match the specified `length`.
- **TypeError** – This exception is raised if `key_material` or `buffer` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- *cryptography.exceptions.InvalidKey* – This is raised when the derived key does not match the expected key.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

class cryptography.hazmat.primitives.kdf.concatkdf.**ConcatKDFHMAC**(*algorithm*, *length*, *salt*, *otherinfo*)

Added in version 1.0.

Similar to ConcatKFDHash but uses an HMAC function instead.

> **Warning**
>
> ConcatKDFHMAC should not be used for password storage.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.concatkdf import ConcatKDFHMAC
>>> salt = os.urandom(16)
>>> otherinfo = b"concatkdf-example"
>>> ckdf = ConcatKDFHMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     otherinfo=otherinfo,
... )
>>> key = ckdf.derive(b"input key")
>>> ckdf = ConcatKDFHMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     otherinfo=otherinfo,
... )
>>> ckdf.verify(b"input key", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key in bytes. Maximum is `hashlen * (2^32 -1)`.
- **salt** (*bytes*) – A salt. Randomizes the KDF's output. Optional, but highly recommended. Ideally as many bits of entropy as the security level of the hash: often that means cryptographically random and as long as the hash output. Does not have to be secret, but may cause stronger security guarantees if secret; If `None` is explicitly passed a default salt of `algorithm.block_size` null bytes will be used.
- **otherinfo** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.

Raises

`TypeError` – This exception is raised if `salt` or `otherinfo` is not `bytes`.

derive(*key_material*)

Parameters

key_material (*bytes*) – The input key material.

Return bytes

The derived key.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** (*bytes-like*) – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int

the number of bytes written to the buffer.

Raises

- **ValueError** – This exception is raised if the buffer length does not match the specified `length`.
- **TypeError** – This exception is raised if `key_material` or `buffer` is not `bytes`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

HKDF

class cryptography.hazmat.primitives.kdf.hkdf.**HKDF**(*algorithm*, *length*, *salt*, *info*)

Added in version 0.2.

HKDF (HMAC-based Extract-and-Expand Key Derivation Function) is suitable for deriving keys of a fixed size used for other cryptographic operations.

> **Warning**
>
> HKDF should not be used for password storage.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDF
>>> salt = os.urandom(16)
>>> info = b"hkdf-example"
>>> hkdf = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     info=info,
... )
>>> key = hkdf.derive(b"input key")
>>> hkdf = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     info=info,
... )
>>> hkdf.verify(b"input key", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key in bytes. Maximum is `255 * (algorithm.digest_size // 8)`.
- **salt** (*bytes*) – A salt. Randomizes the KDF's output. Optional, but highly recommended. Ideally as many bits of entropy as the security level of the hash: often that means cryptographically random and as long as the hash output. Worse (shorter, less entropy) salt values can still meaningfully contribute to security. May be reused. Does not have to be secret, but may cause stronger security guarantees if secret; see **RFC 5869** and the HKDF paper for more details. If `None` is explicitly passed a default salt of `algorithm.digest_size // 8` null bytes will be used. See understanding HKDF for additional detail about the salt and info parameters.
- **info** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.

Raises

`TypeError` – This exception is raised if `salt` or `info` is not `bytes`.

static extract(*algorithm*, *salt*, *key_material*)

Added in version 47.0.0.

> **Note**
>
> Extract is a component of the complete HKDF algorithm. Unless needed for implementing an existing protocol, users should ignore this method and use call *derive()*.

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **salt** (*bytes*) – A salt. Randomizes the KDF's output. Optional, but highly recommended. Ideally as many bits of entropy as the security level of the hash: often that means cryptographically random and as long as the hash output. Worse (shorter, less entropy) salt values can still meaningfully contribute to security. May be reused. Does not have to be secret, but may cause stronger security guarantees if secret; see **RFC 5869** and the HKDF paper for more details. If `None` is explicitly passed a default salt of `algorithm.digest_size // 8` null bytes will be used. See understanding HKDF for additional detail about the salt and info parameters.
- **key_material** (*bytes-like*) – The input key material.

Return bytes
The extracted value.

Raises
TypeError – This exception is raised if `key_material`, `salt`, or `algorithm` are the wrong type.

derive(*key_material*)

Parameters
key_material (*bytes-like*) – The input key material.

Return bytes
The derived key.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material by performing both the extract and expand operations.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int
The number of bytes written to the buffer.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- **ValueError** – This exception is raised if the buffer is too small for the derived key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material by performing both the extract and expand operations, writing the result into the provided buffer.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

class cryptography.hazmat.primitives.kdf.hkdf.**HKDFExpand**(*algorithm*, *length*, *info*)

Added in version 0.5.

HKDF consists of two stages, extract and expand. This class exposes an expand only version of HKDF that is suitable when the key material is already cryptographically strong.

> **Warning**
>
> HKDFExpand should only be used if the key material is cryptographically strong. You should use *HKDF* if you are unsure.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.hkdf import HKDFExpand
>>> info = b"hkdf-example"
>>> key_material = os.urandom(16)
>>> hkdf = HKDFExpand(
...     algorithm=hashes.SHA256(),
...     length=32,
...     info=info,
... )
>>> key = hkdf.derive(key_material)
>>> hkdf = HKDFExpand(
...     algorithm=hashes.SHA256(),
...     length=32,
...     info=info,
... )
>>> hkdf.verify(key_material, key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key in bytes. Maximum is `255 * (algorithm.digest_size // 8)`.

- **info** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.

Raises

`TypeError` – This exception is raised if `info` is not `bytes`.

`derive`(*key_material*)

Parameters

key_material (*bytes*) – The input key material.

Return bytes

The derived key.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material by only performing the expand operation.

`derive_into`(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes*) – The input key material.
- **buffer** – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int

The number of bytes written to the buffer.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.
- `ValueError` – This exception is raised if the buffer is too small for the derived key.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material by only performing the expand operation, writing the result into the provided buffer.

`verify`(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- *cryptography.exceptions.InvalidKey* – This is raised when the derived key does not match the expected key.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

- **TypeError** – This is raised if the provided `key_material` is a `unicode` object

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

KBKDF

class cryptography.hazmat.primitives.kdf.kbkdf.**KBKDFHMAC**(*algorithm*, *mode*, *length*, *rlen*, *llen*, *location*, *label*, *context*, *fixed*)

Added in version 1.4.

KBKDF (Key Based Key Derivation Function) is defined by the NIST SP 800-108 document, to be used to derive additional keys from a key that has been established through an automated key-establishment scheme.

> **Warning**
>
> KBKDFHMAC should not be used for password storage.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.kbkdf import (
...     CounterLocation, KBKDFHMAC, Mode
... )
>>> label = b"KBKDF HMAC Label"
>>> context = b"KBKDF HMAC Context"
>>> kdf = KBKDFHMAC(
...     algorithm=hashes.SHA256(),
...     mode=Mode.CounterMode,
...     length=32,
...     rlen=4,
...     llen=4,
...     location=CounterLocation.BeforeFixed,
...     label=label,
...     context=context,
...     fixed=None,
... )
>>> key = kdf.derive(b"input key")
>>> kdf = KBKDFHMAC(
...     algorithm=hashes.SHA256(),
...     mode=Mode.CounterMode,
...     length=32,
...     rlen=4,
...     llen=4,
...     location=CounterLocation.BeforeFixed,
...     label=label,
...     context=context,
...     fixed=None,
... )
>>> kdf.verify(b"input key", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.

- **mode** – The desired mode of the PRF. A value from the *Mode* enum.
- **length** (*int*) – The desired length of the derived key in bytes.
- **rlen** (*int*) – An integer that indicates the length of the binary representation of the counter in bytes.
- **llen** (*int*) – An integer that indicates the binary representation of the `length` in bytes.
- **location** – The desired location of the counter. A value from the *CounterLocation* enum.
- **label** (*bytes*) – Application specific label information. If `None` is explicitly passed an empty byte string will be used.
- **context** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.
- **fixed** (*bytes*) – Instead of specifying `label` and `context` you may supply your own fixed data. If `fixed` is specified, `label` and `context` is ignored.
- **break_location** (*int*) – A keyword-only argument. An integer that indicates the bytes offset where counter bytes are to be located. Required when `location` is *MiddleFixed*.

Raises

- **TypeError** – This exception is raised if `label` or `context` is not `bytes`. Also raised if `rlen`, `llen`, or `break_location` is not `int`.
- **ValueError** – This exception is raised if `rlen` or `llen` is greater than 4 or less than 1. This exception is also raised if you specify a `label` or `context` and `fixed`. This exception is also raised if you specify `break_location` and `location` is not *MiddleFixed*.

derive(*key_material*)

Parameters
key_material (*bytes-like*) – The input key material.

Return bytes
The derived key.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** (*bytes-like*) – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int
the number of bytes written to the buffer.

Raises

- **ValueError** – This exception is raised if the buffer length does not match the specified `length`.
- **TypeError** – This exception is raised if `key_material` or `buffer` is not `bytes`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

class cryptography.hazmat.primitives.kdf.kbkdf.**KBKDFCMAC**(*algorithm*, *mode*, *length*, *rlen*, *llen*, *location*, *label*, *context*, *fixed*)

Added in version 35.0.0.

KBKDF (Key Based Key Derivation Function) is defined by the NIST SP 800-108 document, to be used to derive additional keys from a key that has been established through an automated key-establishment scheme.

Warning

KBKDFCMAC should not be used for password storage.

```
>>> from cryptography.hazmat.primitives.ciphers import algorithms
>>> from cryptography.hazmat.primitives.kdf.kbkdf import (
...     CounterLocation, KBKDFCMAC, Mode
... )
>>> label = b"KBKDF CMAC Label"
>>> context = b"KBKDF CMAC Context"
>>> kdf = KBKDFCMAC(
...     algorithm=algorithms.AES,
...     mode=Mode.CounterMode,
...     length=32,
...     rlen=4,
...     llen=4,
...     location=CounterLocation.BeforeFixed,
...     label=label,
...     context=context,
```

(continues on next page)

(continued from previous page)

```
...     fixed=None,
... )
>>> key = kdf.derive(b"32 bytes long input key material")
>>> kdf = KBKDFCMAC(
...     algorithm=algorithms.AES,
...     mode=Mode.CounterMode,
...     length=32,
...     rlen=4,
...     llen=4,
...     location=CounterLocation.BeforeFixed,
...     label=label,
...     context=context,
...     fixed=None,
... )
>>> kdf.verify(b"32 bytes long input key material", key)
```

Parameters

- **algorithm** – A class implementing a block cipher algorithm being a subclass of *CipherAlgorithm* and *BlockCipherAlgorithm*.
- **mode** – The desired mode of the PRF. A value from the *Mode* enum.
- **length** (*int*) – The desired length of the derived key in bytes.
- **rlen** (*int*) – An integer that indicates the length of the binary representation of the counter in bytes.
- **llen** (*int*) – An integer that indicates the binary representation of the `length` in bytes.
- **location** – The desired location of the counter. A value from the *CounterLocation* enum.
- **label** (*bytes*) – Application specific label information. If `None` is explicitly passed an empty byte string will be used.
- **context** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.
- **fixed** (*bytes*) – Instead of specifying `label` and `context` you may supply your own fixed data. If `fixed` is specified, `label` and `context` is ignored.
- **break_location** (*int*) – A keyword-only argument. An integer that indicates the bytes offset where counter bytes are to be located. Required when `location` is *MiddleFixed*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if `algorithm` is not a subclass of *CipherAlgorithm* and *BlockCipherAlgorithm*.
- **TypeError** – This exception is raised if `label` or `context` is not `bytes`, `rlen`, `llen`, or `break_location` is not `int`, `mode` is not *Mode* or `location` is not *CounterLocation*.
- **ValueError** – This exception is raised if `rlen` or `llen` is greater than 4 or less than 1. This exception is also raised if you specify a `label` or `context` and `fixed`. This exception is also raised if you specify `break_location` and `location` is not *MiddleFixed*.

derive(*key_material*)

Parameters
key_material (*bytes-like*) – The input key material.

Return bytes
The derived key.

Raises

- **TypeError** – This exception is raised if `key_material` is not `bytes`.
- **ValueError** – This exception is raised if `key_material` is not a valid key for `algorithm` passed to *KBKDFCMAC* constructor.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int
the number of bytes written to the buffer.

Raises

- **ValueError** – This exception is raised if the buffer length does not match the specified `length`, or if `key_material` is not a valid key for `algorithm` passed to *KBKDFCMAC* constructor.
- **TypeError** – This exception is raised if `key_material` or `buffer` is not `bytes`.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- ***cryptography.exceptions.InvalidKey*** – This is raised when the derived key does not match the expected key.
- ***cryptography.exceptions.AlreadyFinalized*** – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

class cryptography.hazmat.primitives.kdf.kbkdf.**Mode**

An enumeration for the key based key derivative modes.

CounterMode

The output of the PRF is computed with a counter as the iteration variable.

class cryptography.hazmat.primitives.kdf.kbkdf.**CounterLocation**

An enumeration for the key based key derivative counter location.

BeforeFixed

The counter iteration variable will be concatenated before the fixed input data.

AfterFixed

The counter iteration variable will be concatenated after the fixed input data.

MiddleFixed

Added in version 38.0.0.

The counter iteration variable will be concatenated in the middle of the fixed input data.

X963KDF

class cryptography.hazmat.primitives.kdf.x963kdf.**X963KDF**(*algorithm*, *length*, *otherinfo*)

Added in version 1.1.

X963KDF (ANSI X9.63 Key Derivation Function) is defined by ANSI in the ANSI X9.63:2001 document, to be used to derive keys for use after a Key Exchange negotiation operation.

SECG in SEC 1 v2.0 recommends that *ConcatKDFHash* be used for new projects. This KDF should only be used for backwards compatibility with pre-existing protocols.

> **Warning**
>
> X963KDF should not be used for password storage.

```
>>> import os
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.x963kdf import X963KDF
>>> sharedinfo = b"ANSI X9.63 Example"
>>> xkdf = X963KDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     sharedinfo=sharedinfo,
... )
>>> key = xkdf.derive(b"input key")
>>> xkdf = X963KDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     sharedinfo=sharedinfo,
... )
>>> xkdf.verify(b"input key", key)
```

Parameters

- **algorithm** – An instance of *HashAlgorithm*.

- **length** (*int*) – The desired length of the derived key in bytes. Maximum is `hashlen * (2^32 -1)`.
- **sharedinfo** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.

Raises
`TypeError` – This exception is raised if `sharedinfo` is not `bytes`.

derive(*key_material*)

Parameters
key_material (*bytes-like*) – The input key material.

Return bytes
The derived key.

Raises

- `TypeError` – This exception is raised if `key_material` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material.
- **buffer** (*bytes-like*) – A writable buffer to write the derived key into. The buffer must be equal to the length supplied in the constructor.

Return int
the number of bytes written to the buffer.

Raises

- `ValueError` – This exception is raised if the buffer length does not match the specified `length`.
- `TypeError` – This exception is raised if `key_material` or `buffer` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

Derives a new key from the input key material and writes it into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- *cryptography.exceptions.InvalidKey* – This is raised when the derived key does not match the expected key.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match.

Interface

class cryptography.hazmat.primitives.kdf.**KeyDerivationFunction**

Added in version 0.2.

derive(*key_material*)

Parameters

key_material (*bytes*) – The input key material. Depending on what key derivation function you are using this could be either random bytes, or a user supplied password.

Returns

The new key.

Raises

cryptography.exceptions.AlreadyFinalized – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This generates and returns a new key from the supplied key material.

derive_into(*key_material*, *buffer*)

Added in version 47.0.0.

Parameters

- **key_material** (*bytes-like*) – The input key material. Depending on what key derivation function you are using this could be either random bytes, or a user supplied password.
- **buffer** – A writable buffer to write the derived key into.

Return int

the number of bytes written to the buffer.

Raises

- **ValueError** – This exception is raised if the buffer length does not match the expected key length.
- **TypeError** – This exception is raised if `key_material` or `buffer` is not `bytes`.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This generates a new key from the supplied key material and writes it directly into the provided buffer. This is useful when you want to avoid allocating new memory for the derived key.

verify(*key_material*, *expected_key*)

Parameters

- **key_material** (*bytes*) – The input key material. This is the same as `key_material` in *derive()*.
- **expected_key** (*bytes*) – The expected result of deriving a new key, this is the same as the return value of *derive()*.

Raises

- *cryptography.exceptions.InvalidKey* – This is raised when the derived key does not match the expected key.
- *cryptography.exceptions.AlreadyFinalized* – This is raised when *derive()*, *derive_into()*, or *verify()* is called more than once.

This checks whether deriving a new key from the supplied `key_material` generates the same key as the `expected_key`, and raises an exception if they do not match. This can be used for something like checking whether a user's password attempt matches the stored derived key.

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.

2.4.6 Key wrapping

Key wrapping is a cryptographic construct that uses symmetric encryption to encapsulate key material. Key wrapping algorithms are occasionally utilized to protect keys at rest or transmit them over insecure networks. Many of the protections offered by key wrapping are also offered by using authenticated *symmetric encryption*.

`cryptography.hazmat.primitives.keywrap.`**`aes_key_wrap`**(*wrapping_key*, *key_to_wrap*)

Added in version 1.1.

This function performs AES key wrap (without padding) as specified in **RFC 3394**.

Parameters

- **wrapping_key** (*bytes*) – The wrapping key.
- **key_to_wrap** (*bytes*) – The key to wrap.

Return bytes
The wrapped key as bytes.

`cryptography.hazmat.primitives.keywrap.`**`aes_key_unwrap`**(*wrapping_key*, *wrapped_key*)

Added in version 1.1.

This function performs AES key unwrap (without padding) as specified in **RFC 3394**.

Parameters

- **wrapping_key** (*bytes*) – The wrapping key.
- **wrapped_key** (*bytes*) – The wrapped key.

Return bytes
The unwrapped key as bytes.

Raises
cryptography.hazmat.primitives.keywrap.InvalidUnwrap – This is raised if the key is not successfully unwrapped.

`cryptography.hazmat.primitives.keywrap.`**`aes_key_wrap_with_padding`**(*wrapping_key*, *key_to_wrap*)

Added in version 2.2.

This function performs AES key wrap with padding as specified in **RFC 5649**.

Parameters

- **wrapping_key** (*bytes*) – The wrapping key.
- **key_to_wrap** (*bytes*) – The key to wrap.

Return bytes
The wrapped key as bytes.

`cryptography.hazmat.primitives.keywrap.aes_key_unwrap_with_padding(wrapping_key, wrapped_key)`

Added in version 2.2.

This function performs AES key unwrap with padding as specified in **RFC 5649**.

Parameters

- **wrapping_key** (*bytes*) – The wrapping key.
- **wrapped_key** (*bytes*) – The wrapped key.

Return bytes
The unwrapped key as bytes.

Raises
cryptography.hazmat.primitives.keywrap.InvalidUnwrap – This is raised if the key is not successfully unwrapped.

Exceptions

class `cryptography.hazmat.primitives.keywrap.InvalidUnwrap`

This is raised when a wrapped key fails to unwrap. It can be caused by a corrupted or invalid wrapped key or an invalid wrapping key.

> 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.

2.4.7 Message authentication codes

While cryptography supports multiple MAC algorithms, we strongly recommend that HMAC should be used unless you have a very specific need.

For more information on why HMAC is preferred, see Use cases for CMAC vs. HMAC?

> 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.

Cipher-based message authentication code (CMAC)

Cipher-based message authentication codes (or CMACs) are a tool for calculating message authentication codes using a block cipher coupled with a secret key. You can use an CMAC to verify both the integrity and authenticity of a message.

A subset of CMAC with the AES-128 algorithm is described in **RFC 4493**.

class cryptography.hazmat.primitives.cmac.**CMAC**(*algorithm*)

Added in version 0.4.

CMAC objects take a *BlockCipherAlgorithm* instance.

```
>>> from cryptography.hazmat.primitives import cmac
>>> from cryptography.hazmat.primitives.ciphers import algorithms
>>> key = b"\x00" * 16 # A real key should come from os.urandom(16)
>>> c = cmac.CMAC(algorithms.AES(key))
>>> c.update(b"message to authenticate")
>>> c.finalize()
b'CT\x1d\xc8\x0e\x15\xbe4e\xdb\xb6\x84\xca\xd9Xk'
```

If `algorithm` isn't a *BlockCipherAlgorithm* instance then `TypeError` will be raised.

To check that a given signature is correct use the *verify()* method. You will receive an exception if the signature is wrong:

```
>>> c = cmac.CMAC(algorithms.AES(key))
>>> c.update(b"message to authenticate")
>>> c.verify(b"an incorrect signature")
Traceback (most recent call last):
...
cryptography.exceptions.InvalidSignature: Signature did not match digest.
```

Parameters

algorithm – An instance of *BlockCipherAlgorithm*.

Raises

- **TypeError** – This is raised if the provided `algorithm` is not an instance of *BlockCipherAlgorithm*
- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `algorithm` is unsupported.

update(*data*)

Parameters

data (*bytes*) – The bytes to hash and authenticate.

Raises

- ***cryptography.exceptions.AlreadyFinalized*** – See *finalize()*
- **TypeError** – This exception is raised if `data` is not `bytes`.

copy()

Copy this *CMAC* instance, usually so that we may call *finalize()* to get an intermediate value while we continue to call *update()* on the original instance.

Returns

A new instance of *CMAC* that can be updated and finalized independently of the original instance.

Raises

cryptography.exceptions.AlreadyFinalized – See *finalize()*

verify(*signature*)

Finalize the current context and securely compare the MAC to `signature`.

Parameters

signature (*bytes*) – The bytes to compare the current CMAC against.

Raises

- *cryptography.exceptions.AlreadyFinalized* – See *finalize()*
- *cryptography.exceptions.InvalidSignature* – If signature does not match digest
- TypeError – This exception is raised if `signature` is not `bytes`.

finalize()

Finalize the current context and return the message authentication code as bytes.

After `finalize` has been called this object can no longer be used and *update()*, *copy()*, *verify()* and *finalize()* will raise an *AlreadyFinalized* exception.

Return bytes

The message authentication code as bytes.

Raises

cryptography.exceptions.AlreadyFinalized –

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.

Hash-based message authentication codes (HMAC)

Hash-based message authentication codes (or HMACs) are a tool for calculating message authentication codes using a cryptographic hash function coupled with a secret key. You can use an HMAC to verify both the integrity and authenticity of a message.

class cryptography.hazmat.primitives.hmac.**HMAC**(*key*, *algorithm*)

HMAC objects take a `key` and a *HashAlgorithm* instance. The `key` should be *randomly generated bytes* and is recommended to be equal in length to the `digest_size` of the hash function chosen. You must keep the `key` secret.

This is an implementation of **RFC 2104**.

```
>>> from cryptography.hazmat.primitives import hashes, hmac
>>> key = b'test key. Beware! A real key should use os.urandom or TRNG to generate'
>>> h = hmac.HMAC(key, hashes.SHA256())
>>> h.update(b"message to hash")
>>> signature = h.finalize()
>>> signature
b'k\xd9\xb29\xefS\xf8\xcf\xec\xed\xbf\x95\xe6\x97X\x18\x9e%\x11DU1\x9fq}\x9a\x9c\
↪xe0)y`='
```

If `algorithm` isn't a *HashAlgorithm* instance then `TypeError` will be raised.

To check that a given signature is correct use the *verify()* method. You will receive an exception if the signature is wrong:

```
>>> h = hmac.HMAC(key, hashes.SHA256())
>>> h.update(b"message to hash")
>>> h_copy = h.copy() # get a copy of `h' to be reused
>>> h.verify(signature)
>>>
>>> h_copy.verify(b"an incorrect signature")
Traceback (most recent call last):
...
cryptography.exceptions.InvalidSignature: Signature did not match digest.
```

Parameters

- **key** (*bytes-like*) – The secret key.
- **algorithm** – An *HashAlgorithm* instance such as those described in *Cryptographic Hashes*.

Raises

cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `algorithm` isn't supported.

update(*msg*)

Parameters

msg (*bytes-like*) – The bytes to hash and authenticate.

Raises

- ***cryptography.exceptions.AlreadyFinalized*** – See *finalize()*
- **TypeError** – This exception is raised if `msg` is not `bytes`.

copy()

Copy this *HMAC* instance, usually so that we may call *finalize()* to get an intermediate digest value while we continue to call *update()* on the original instance.

Returns

A new instance of *HMAC* that can be updated and finalized independently of the original instance.

Raises

cryptography.exceptions.AlreadyFinalized – See *finalize()*

verify(*signature*)

Finalize the current context and securely compare digest to `signature`.

Parameters

signature (*bytes*) – The bytes to compare the current digest against.

Raises

- ***cryptography.exceptions.AlreadyFinalized*** – See *finalize()*
- ***cryptography.exceptions.InvalidSignature*** – If signature does not match digest
- **TypeError** – This exception is raised if `signature` is not `bytes`.

finalize()

Finalize the current context and return the message digest as bytes.

After `finalize` has been called this object can no longer be used and *update()*, *copy()*, *verify()* and *finalize()* will raise an *AlreadyFinalized* exception.

Return bytes
The message digest as bytes.

Raises
cryptography.exceptions.AlreadyFinalized –

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.

Poly1305

Poly1305 is an authenticator that takes a 32-byte key and a message and produces a 16-byte tag. This tag is used to authenticate the message. Each key **must** only be used once. Using the same key to generate tags for multiple messages allows an attacker to forge tags. Poly1305 is described in **RFC 7539**.

class cryptography.hazmat.primitives.poly1305.**Poly1305**(*key*)

Added in version 2.7.

Warning

Using the same key to generate tags for multiple messages allows an attacker to forge tags. Always generate a new key per message you want to authenticate. If you are using this as a MAC for symmetric encryption please use *ChaCha20Poly1305* instead.

```
>>> from cryptography.hazmat.primitives import poly1305
>>> key = b"\x01" * 32 # A real key should come from os.urandom(32)
>>> p = poly1305.Poly1305(key)
>>> p.update(b"message to authenticate")
>>> p.finalize()
b'T\xae\xff3\xbdW\xef\xd5r\x01\xe2n=\xb7\xd2h'
```

To check that a given tag is correct use the *verify()* method. You will receive an exception if the tag is wrong:

```
>>> p = poly1305.Poly1305(key)
>>> p.update(b"message to authenticate")
>>> p.verify(b"an incorrect tag")
Traceback (most recent call last):
...
cryptography.exceptions.InvalidSignature: Value did not match computed tag.
```

Parameters
key (*bytes-like*) – The secret key.

Raises
cryptography.exceptions.UnsupportedAlgorithm – This is raised if the version of OpenSSL `cryptography` is compiled against does not support this algorithm.

update(*data*)

Parameters
data (*bytes-like*) – The bytes to hash and authenticate.

Raises

- *cryptography.exceptions.AlreadyFinalized* – See *finalize()*
- **TypeError** – This exception is raised if `data` is not `bytes`.

verify(*tag*)

Finalize the current context and securely compare the MAC to `tag`.

Parameters
tag (*bytes*) – The bytes to compare against.

Raises

- *cryptography.exceptions.AlreadyFinalized* – See *finalize()*
- *cryptography.exceptions.InvalidSignature* – If tag does not match.
- **TypeError** – This exception is raised if `tag` is not `bytes`.

finalize()

Finalize the current context and return the message authentication code as bytes.

After `finalize` has been called this object can no longer be used and *update()*, *verify()*, and *finalize()* will raise an *AlreadyFinalized* exception.

Return bytes
The message authentication code as bytes.

Raises
cryptography.exceptions.AlreadyFinalized –

classmethod generate_tag(*key*, *data*)

A single step alternative to do sign operations. Returns the message authentication code as `bytes` for the given `key` and `data`.

Parameters

- **key** (*bytes-like*) – Secret key as `bytes`.
- **data** (*bytes-like*) – The bytes to hash and authenticate.

Return bytes
The message authentication code as bytes.

Raises

- *cryptography.exceptions.UnsupportedAlgorithm* – This is raised if the version of OpenSSL `cryptography` is compiled against does not support this algorithm.
- **TypeError** – This exception is raised if `key` or `data` are not `bytes`.

```
>>> poly1305.Poly1305.generate_tag(key, b"message to authenticate")
b'T\xae\xff3\xbdW\xef\xd5r\x01\xe2n=\xb7\xd2h'
```

classmethod verify_tag(*key*, *data*, *tag*)

A single step alternative to do verify operations. Securely compares the MAC to `tag`, using the given `key` and `data`.

Parameters

- **key** (*bytes-like*) – Secret key as `bytes`.
- **data** (*bytes-like*) – The bytes to hash and authenticate.
- **tag** (*bytes*) – The bytes to compare against.

Raises

- *cryptography.exceptions.UnsupportedAlgorithm* – This is raised if the version of OpenSSL `cryptography` is compiled against does not support this algorithm.
- **TypeError** – This exception is raised if `key`, `data` or `tag` are not `bytes`.
- *cryptography.exceptions.InvalidSignature* – If tag does not match.

```
>>> poly1305.Poly1305.verify_tag(key, b"message to authenticate", b"an␣
↪incorrect tag")
Traceback (most recent call last):
...
cryptography.exceptions.InvalidSignature: Value did not match computed tag.
```

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.

2.4.8 Message digests (Hashing)

class `cryptography.hazmat.primitives.hashes.`**Hash**(*algorithm*)

A cryptographic hash function takes an arbitrary block of data and calculates a fixed-size bit string (a digest), such that different data results (with a high probability) in different digests.

This is an implementation of *HashContext* meant to be used with *HashAlgorithm* implementations to provide an incremental interface to calculating various message digests.

```
>>> from cryptography.hazmat.primitives import hashes
>>> digest = hashes.Hash(hashes.SHA256())
>>> digest.update(b"abc")
>>> digest.update(b"123")
>>> digest.finalize()
b'l\xa1=R\xcap\xc8\x83\xe0\xf0\xbb\x10\x1eBZ\x89\xe8bM\xe5\x1d\xb2\xd29%\x93\xafj\
↪x84\x11\x80\x90'
```

Keep in mind that attacks against cryptographic hashes only get stronger with time, and that often algorithms that were once thought to be strong, become broken. Because of this it's important to include a plan for upgrading the hash algorithm you use over time. For more information, see Lifetimes of cryptographic hash functions.

Parameters

algorithm – A *HashAlgorithm* instance such as those described *below*.

Raises

cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `algorithm` is unsupported.

update(*data*)

Parameters
data (*bytes*) – The bytes to be hashed.

Raises

- *cryptography.exceptions.AlreadyFinalized* – See *finalize()*.
- **TypeError** – This exception is raised if `data` is not `bytes`.

copy()

Copy this *Hash* instance, usually so that you may call *finalize()* to get an intermediate digest value while we continue to call *update()* on the original instance.

Returns
A new instance of *Hash* that can be updated and finalized independently of the original instance.

Raises
cryptography.exceptions.AlreadyFinalized – See *finalize()*.

finalize()

Finalize the current context and return the message digest as bytes.

After `finalize` has been called this object can no longer be used and *update()*, *copy()*, and *finalize()* will raise an *AlreadyFinalized* exception.

Return bytes
The message digest as bytes.

static hash(*algorithm*, *data*)

Added in version 47.0.0.

A one-shot method to compute the digest for the provided `data`.

```
>>> from cryptography.hazmat.primitives.hashes import Hash
>>> Hash.hash(hashes.SHA256(), b"my data")
b'\xb2\x16{\n\xa7\xefw\x94t\x0b\x05Z\xc7\xa8\x80\xa5)4\xaag\xef\x1c\xa6\x88z\
↪xd8\x1d\xcc\xef\xd5\xb9\xde'
```

Parameters

- **algorithm** – A *HashAlgorithm* instance.
- **data** – The bytes to be hashed.

Return bytes
The message digest as bytes.

Raises

- *cryptography.exceptions.UnsupportedAlgorithm* – This is raised if the provided `algorithm` is unsupported.
- **TypeError** – This exception is raised if `data` is not `bytes`.

class cryptography.hazmat.primitives.hashes.**XOFHash**(*algorithm*)

An extendable output function (XOF) is a cryptographic hash function that can produce an arbitrary amount of output for a given input. The output can be obtained by repeatedly calling *squeeze()* with the desired length.

```
>>> from cryptography.hazmat.primitives import hashes
>>> digest = hashes.XOFHash(hashes.SHAKE128.xof())
>>> digest.update(b"abc")
>>> digest.update(b"123")
>>> digest.squeeze(16)
b'\x18\xd6\xbd\xeb5u\x83[@\xfa%/\xdc\xca\x9f\x1b'
>>> digest.squeeze(16)
b'\xc2\xeb\x12\x05\xc3\xf9Bu\x88\xe0\xda\x80FvAV'
```

Parameters

algorithm – A *ExtendableOutputFunction* instance such as those described *below*. The `digest_size` passed is the maximum number of bytes that can be squeezed from the XOF when using this class.

Raises

cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `algorithm` is unsupported.

update(*data*)

Parameters

data (*bytes*) – The bytes to be hashed.

Raises

- ***cryptography.exceptions.AlreadyFinalized*** – If already squeezed.
- **TypeError** – This exception is raised if `data` is not `bytes`.

copy()

Copy this *XOFHash* instance, usually so that you may call *squeeze()* to get an intermediate digest value while we continue to call *update()* on the original instance.

Returns

A new instance of *XOFHash* that can be updated and squeezed independently of the original instance. If you copy an instance that has already been squeezed, the copy will also be in a squeezed state.

Raises

cryptography.exceptions.AlreadyFinalized – See *squeeze()*.

squeeze(*length*)

Parameters

length (*int*) – The number of bytes to squeeze.

After *squeeze()* has been called this object can no longer be updated and *update()*, will raise an *AlreadyFinalized* exception.

Return bytes

`length` bytes of output from the extendable output function (XOF).

Raises

ValueError – If the maximum number of bytes that can be squeezed has been exceeded.

SHA-2 family

class `cryptography.hazmat.primitives.hashes.`**`SHA224`**

SHA-224 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 224-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA256`**

SHA-256 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 256-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA384`**

SHA-384 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 384-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA512`**

SHA-512 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 512-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA512_224`**

Added in version 2.5.

SHA-512/224 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 224-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA512_256`**

Added in version 2.5.

SHA-512/256 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 256-bit message digest.

BLAKE2

BLAKE2 is a cryptographic hash function specified in **RFC 7693**. BLAKE2's design makes it immune to length-extension attacks, an advantage over the SHA-family of hashes.

> **Note**
>
> While the RFC specifies keying, personalization, and salting features, these are not supported at this time due to limitations in OpenSSL.

class `cryptography.hazmat.primitives.hashes.`**`BLAKE2b`**(*digest_size*)

BLAKE2b is optimized for 64-bit platforms and produces an 1 to 64-byte message digest.

Parameters

`digest_size` (*int*) – The desired size of the hash output in bytes. Only `64` is supported at this time.

Raises

`ValueError` – If the `digest_size` is invalid.

class `cryptography.hazmat.primitives.hashes.`**`BLAKE2s`**(*digest_size*)

BLAKE2s is optimized for 8 to 32-bit platforms and produces a 1 to 32-byte message digest.

Parameters

`digest_size` (*int*) – The desired size of the hash output in bytes. Only `32` is supported at this time.

Raises
`ValueError` – If the `digest_size` is invalid.

SHA-3 family

SHA-3 is the most recent NIST secure hash algorithm standard. Despite the larger number SHA-3 is not considered to be better than SHA-2. Instead, it uses a significantly different internal structure so that **if** an attack appears against SHA-2 it is unlikely to apply to SHA-3. SHA-3 is significantly slower than SHA-2 so at this time most users should choose SHA-2.

class `cryptography.hazmat.primitives.hashes.`**`SHA3_224`**

Added in version 2.5.

SHA3/224 is a cryptographic hash function from the SHA-3 family and is standardized by NIST. It produces a 224-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA3_256`**

Added in version 2.5.

SHA3/256 is a cryptographic hash function from the SHA-3 family and is standardized by NIST. It produces a 256-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA3_384`**

Added in version 2.5.

SHA3/384 is a cryptographic hash function from the SHA-3 family and is standardized by NIST. It produces a 384-bit message digest.

class `cryptography.hazmat.primitives.hashes.`**`SHA3_512`**

Added in version 2.5.

SHA3/512 is a cryptographic hash function from the SHA-3 family and is standardized by NIST. It produces a 512-bit message digest.

SHA-1

Warning

SHA-1 is a deprecated hash algorithm that has practical known collision attacks. You are strongly discouraged from using it. Existing applications should strongly consider moving away.

class `cryptography.hazmat.primitives.hashes.`**`SHA1`**

SHA-1 is a cryptographic hash function standardized by NIST. It produces an 160-bit message digest. Cryptanalysis of SHA-1 has demonstrated that it is vulnerable to practical collision attacks, and collisions have been demonstrated.

MD5

Warning

MD5 is a deprecated hash algorithm that has practical known collision attacks. You are strongly discouraged from using it. Existing applications should strongly consider moving away.

class cryptography.hazmat.primitives.hashes.**MD5**

MD5 is a deprecated cryptographic hash function. It produces a 128-bit message digest and has practical known collision attacks.

SM3

class cryptography.hazmat.primitives.hashes.**SM3**

Added in version 35.0.0.

SM3 is a cryptographic hash function standardized by the Chinese National Cryptography Administration in GM/T 0004-2012. It produces 256-bit message digests. (An English description is available at draft-sca-cfrg-sm3.) This hash should be used for compatibility purposes where required and is not otherwise recommended for use.

Extendable Output Functions

class cryptography.hazmat.primitives.hashes.**SHAKE128**(*digest_size*)

Added in version 2.5.

SHAKE128 is an extendable output function (XOF) based on the same core permutations as SHA3. It allows the caller to obtain an arbitrarily long digest length. Longer lengths, however, do not increase security or collision resistance and lengths shorter than 128 bit (16 bytes) will decrease it.

This class can be used with *Hash* or *XOFHash*. When used in *Hash finalize()* will return `digest_size` bytes. When used in *XOFHash* this defines the total number of bytes allowed to be squeezed.

Parameters
digest_size (*int*) – The length of output desired. Must be greater than zero.

Raises
ValueError – If the `digest_size` is invalid.

classmethod xof()

Returns a *SHAKE128* instance configured for use with *XOFHash*.

class cryptography.hazmat.primitives.hashes.**SHAKE256**(*digest_size*)

Added in version 2.5.

SHAKE256 is an extendable output function (XOF) based on the same core permutations as SHA3. It allows the caller to obtain an arbitrarily long digest length. Longer lengths, however, do not increase security or collision resistance and lengths shorter than 256 bit (32 bytes) will decrease it.

This class can be used with *Hash* or *XOFHash*. When used in *Hash finalize()* will return `digest_size` bytes. When used in *XOFHash* this defines the total number of bytes allowed to be squeezed.

Parameters
digest_size (*int*) – The length of output desired. Must be greater than zero.

Raises
ValueError – If the `digest_size` is invalid.

classmethod xof()

Returns a *SHAKE256* instance configured for use with *XOFHash*.

Interfaces

class cryptography.hazmat.primitives.hashes.**HashAlgorithm**

name

Type
str

The standard name for the hash algorithm, for example: `"sha256"` or `"blake2b"`.

digest_size

Type
int

The size of the resulting digest in bytes.

class cryptography.hazmat.primitives.hashes.**ExtendableOutputFunction**

An interface applied to hashes that act as extendable output functions (XOFs). The currently supported XOFs are *SHAKE128* and *SHAKE256*.

class cryptography.hazmat.primitives.hashes.**HashContext**

algorithm

A *HashAlgorithm* that will be used by this context.

update(*data*)

Parameters
data (*bytes*) – The data you want to hash.

finalize()

Returns
The final digest as bytes.

copy()

Returns
A *HashContext* that is a copy of the current context.

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)*.

2.4.9 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 in nearly all contexts it is necessary 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.** If *Fernet (symmetric encryption)* is not appropriate for your use-case then you may still benefit from *Authenticated encryption* which combines encryption and authentication securely.

class cryptography.hazmat.primitives.ciphers.**Cipher**(*algorithm*, *mode*)

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*.

Raises

cryptography.exceptions.UnsupportedAlgorithm – This is raised if the provided `algorithm` is unsupported.

encryptor()

Returns

An encrypting *CipherContext* instance.

If the requested combination of `algorithm` and `mode` is unsupported an *UnsupportedAlgorithm* exception will be raised.

decryptor()

Returns

A decrypting *CipherContext* instance.

If the requested combination of `algorithm` and `mode` is unsupported 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.**AES128**(*key*)

Added in version 38.0.0.

An AES class that only accepts 128 bit keys. This is identical to the standard `AES` class except that it will only accept a single key length.

Parameters

key (*bytes-like*) – The secret key. This must be kept secret. `128` *bits* long.

class cryptography.hazmat.primitives.ciphers.algorithms.**AES256**(*key*)

Added in version 38.0.0.

An AES class that only accepts 256 bit keys. This is identical to the standard `AES` class except that it will only accept a single key length.

Parameters

key (*bytes-like*) – The secret key. This must be kept secret. `256` *bits* long.

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

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 49.0.0.

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*, *nonce*)

Added 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** (*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. The nonce does not need to be kept secret and may be included with the ciphertext. This must be `128` *bits* in length. The 128-bit value is a concatenation of a 4-byte little-endian block counter followed by a 12-byte nonce, as described in **RFC 7539**.

> **Note**
>
> The first 4 bytes of the 128-bit value are a 32-bit little-endian block counter and the remaining 12 bytes are the nonce. Each ChaCha20 block encrypts 64 bytes, so an initial counter of `n` allows `(2 ** 32 - n) * 64` bytes to be encrypted before the counter would overflow, at which point a `ValueError` is raised. If you are encrypting a single message and do not need to resume from a particular point in the keystream, start the counter at zero; this gives you the full 256 GiB of keystream for that nonce. If you are continuing a stream from a known position you will need to concatenate the counter and nonce yourself before passing them.

```
>>> import struct, os
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

(continues on next page)

(continued from previous page)

```
>>> key = os.urandom(32)
>>> counter = 0
>>> nonce = struct.pack("<I", counter) + os.urandom(12)
>>> 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*)

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 48.0.0.

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.**SM4**(*key*)

Added 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.**ARC4**(*key*)

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 48.0.0.

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'
```

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*)

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 49.0.0.

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*)

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 49.0.0.

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*)

> **Warning**
>
> This algorithm has been deprecated and moved to the *Decrepit cryptography* module. If you need to continue using it then update your code to use the new module path. It will be removed from this namespace in 49.0.0.

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()
```

(continues on next page)

(continued from previous page)

```
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*)

Added 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*)

Added 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
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.

reset_nonce(*nonce*)

Added in version 43.0.0.

This method allows changing the nonce for an already existing context. Normally the nonce is set when the context is created and internally incremented as data as passed. However, in some scenarios the same key is used repeatedly but the nonce changes non-sequentially (e.g. `QUIC`), which requires updating the context with the new nonce.

This method only works for contexts using *ChaCha20* or *CTR* mode.

Parameters
nonce – The nonce to update the context with.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the algorithm does not support updating the nonce.
- **ValueError** – If the nonce is not the correct length for the algorithm.

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 addi-

tional 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**

Added 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".

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**

Added 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.

> 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.

2.4.10 Symmetric Padding

Padding is a way to take data that may or may not be a multiple of the block size for a cipher and extend it out so that it is. This is required for many block cipher modes as they require the data to be encrypted to be an exact multiple of the block size.

class cryptography.hazmat.primitives.padding.**PKCS7**(*block_size*)

PKCS7 padding is a generalization of PKCS5 padding (also known as standard padding). PKCS7 padding works by appending `N` bytes with the value of `chr(N)`, where `N` is the number of bytes required to make the final block of data the same size as the block size. A simple example of padding is:

```
>>> from cryptography.hazmat.primitives import padding
>>> padder = padding.PKCS7(128).padder()
>>> padded_data = padder.update(b"11111111111111112222222222")
>>> padded_data += padder.finalize()
>>> padded_data
b'11111111111111112222222222\x06\x06\x06\x06\x06\x06'
>>> unpadder = padding.PKCS7(128).unpadder()
```

(continues on next page)

(continued from previous page)

```
>>> data = unpadder.update(padded_data)
>>> data += unpadder.finalize()
>>> data
b'1111111111111111112222222222'
```

Parameters
block_size – The size of the block in *bits* that the data is being padded to.

Raises
ValueError – Raised if block size is not a multiple of 8 or is not between 0 and 2040 inclusive.

padder()

Returns
A padding *PaddingContext* instance.

unpadder()

Returns
An unpadding *PaddingContext* instance.

class cryptography.hazmat.primitives.padding.**ANSIX923**(*block_size*)

Added in version 1.3.

ANSI X9.23 padding works by appending `N-1` bytes with the value of `0` and a last byte with the value of `chr(N)`, where `N` is the number of bytes required to make the final block of data the same size as the block size. A simple example of padding is:

```
>>> padder = padding.ANSIX923(128).padder()
>>> padded_data = padder.update(b"1111111111111111112222222222")
>>> padded_data += padder.finalize()
>>> padded_data
b'1111111111111111112222222222\x00\x00\x00\x00\x00\x06'
>>> unpadder = padding.ANSIX923(128).unpadder()
>>> data = unpadder.update(padded_data)
>>> data += unpadder.finalize()
>>> data
b'1111111111111111112222222222'
```

Parameters
block_size – The size of the block in *bits* that the data is being padded to.

Raises
ValueError – Raised if block size is not a multiple of 8 or is not between 0 and 2040 inclusive.

padder()

Returns
A padding *PaddingContext* instance.

unpadder()

Returns
An unpadding *PaddingContext* instance.

class cryptography.hazmat.primitives.padding.**PaddingContext**

When calling `padder()` or `unpadder()` the result will conform to the `PaddingContext` 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.

update(*data*)

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

Return bytes
Returns the data that was padded or unpadded.

Raises

- **TypeError** – Raised if data is not bytes.
- ***cryptography.exceptions.AlreadyFinalized*** – See *finalize()*.
- **TypeError** – This exception is raised if `data` is not `bytes`.

finalize()

Finalize the current context and return the rest of the data.

After `finalize` has been called this object can no longer be used; *update()* and *finalize()* will raise an *AlreadyFinalized* exception.

Return bytes
Returns the remainder of the data.

Raises

- **TypeError** – Raised if data is not bytes.
- **ValueError** – When trying to remove padding from incorrectly padded data.

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.

2.4.11 Two-factor authentication

This module contains algorithms related to two-factor authentication.

Currently, it contains an algorithm for generating and verifying one time password values based on Hash-based message authentication codes (HMAC), specifically the HOTP (**RFC 4226**) algorithm and its time-based extension TOTP (**RFC 6238**).

class cryptography.hazmat.primitives.twofactor.**InvalidToken**

This is raised when the verify method of a one time password function's computed token does not match the expected token.

HOTP

cryptography.hazmat.primitives.twofactor.hotp.**HOTPHashTypes**

Added in version 40.0.0.

Type alias: A union of supported hash algorithm types: *SHA1*, *SHA256* or *SHA512*.

class cryptography.hazmat.primitives.twofactor.hotp.**HOTP**(*key*, *length*, *algorithm*, **, enforce_key_length=True*)

Added in version 0.3.

HOTP objects take a `key`, `length` and `algorithm` parameter. The `key` should be *randomly generated bytes* and is recommended to be 160 *bits* in length. The `length` parameter controls the length of the generated one time password and must be >= 6 and <= 8.

This is an implementation of **RFC 4226**.

```
>>> import os
>>> from cryptography.hazmat.primitives.twofactor.hotp import HOTP
>>> from cryptography.hazmat.primitives.hashes import SHA1
>>> key = os.urandom(20)
>>> hotp = HOTP(key, 6, SHA1())
>>> hotp_value = hotp.generate(0)
>>> hotp.verify(hotp_value, 0)
```

Parameters

- **key** (*bytes-like*) – Per-user secret key. This value must be kept secret and be at least 128 *bits*. It is recommended that the key be 160 bits.
- **length** (*int*) – Length of generated one time password as `int`.
- **algorithm** (`cryptography.hazmat.primitives.hashes.HashAlgorithm`) – A *hashes* instance (must match *HOTPHashTypes*).
- **enforce_key_length** – A boolean flag defaulting to True that toggles whether a minimum key length of 128 *bits* is enforced. This exists to work around the fact that as documented in Issue #2915, the Google Authenticator PAM module by default generates 80 bit keys. If this flag is set to False, the application developer should implement additional checks of the key length before passing it into *HOTP*.

  Added in version 1.5.

Raises

- **ValueError** – This is raised if the provided `key` is shorter than 128 *bits* or if the `length` parameter is not 6, 7 or 8.
- **TypeError** – This is raised if the provided `algorithm` is not *SHA1()*, *SHA256()* or *SHA512()* or if the `length` parameter is not an integer.

generate(*counter*)

Parameters
counter (*int*) – The counter value used to generate the one time password.

Return bytes
A one time password value.

verify(*hotp*, *counter*)

Parameters

- **hotp** (*bytes*) – The one time password value to validate.
- **counter** (*int*) – The counter value to validate against.

Raises

cryptography.hazmat.primitives.twofactor.InvalidToken – This is raised when the supplied HOTP does not match the expected HOTP.

get_provisioning_uri(*account_name*, *counter*, *issuer*)

Added in version 1.0.

Parameters

- **account_name** (*str*) – The display name of account, such as `'Alice Smith'` or `'alice@example.com'`.
- **issuer** (`str` or `None`) – The optional display name of issuer. This is typically the provider or service the user wants to access using the OTP token.
- **counter** (*int*) – The current value of counter.

Returns

A URI string.

Throttling

Due to the fact that the HOTP algorithm generates rather short tokens that are 6 - 8 digits long, brute force attacks are possible. It is highly recommended that the server that validates the token implement a throttling scheme that locks out the account for a period of time after a number of failed attempts. The number of allowed attempts should be as low as possible while still ensuring that usability is not significantly impacted.

Re-synchronization of the counter

The server's counter value should only be incremented on a successful HOTP authentication. However, the counter on the client is incremented every time a new HOTP value is requested. This can lead to the counter value being out of synchronization between the client and server.

Due to this, it is highly recommended that the server sets a look-ahead window that allows the server to calculate the next `x` HOTP values and check them against the supplied HOTP value. This can be accomplished with something similar to the following code.

```
def verify(hotp, counter, look_ahead):
    assert look_ahead >= 0
    correct_counter = None

    otp = HOTP(key, 6)
    for count in range(counter, counter + look_ahead):
        try:
            otp.verify(hotp, count)
            correct_counter = count
        except InvalidToken:
            pass

    return correct_counter
```

TOTP

class cryptography.hazmat.primitives.twofactor.totp.**TOTP**(*key*, *length*, *algorithm*, *time_step*, *, *enforce_key_length=True*)

TOTP objects take a `key`, `length`, `algorithm` and `time_step` parameter. The `key` should be *randomly gen-*

erated bytes and is recommended to be as long as your hash function's output (e.g 256-bit for SHA256). The `length` parameter controls the length of the generated one time password and must be >= 6 and <= 8.

This is an implementation of **RFC 6238**.

```
>>> import os
>>> import time
>>> from cryptography.hazmat.primitives.twofactor.totp import TOTP
>>> from cryptography.hazmat.primitives.hashes import SHA1
>>> key = os.urandom(20)
>>> totp = TOTP(key, 8, SHA1(), 30)
>>> time_value = time.time()
>>> totp_value = totp.generate(time_value)
>>> totp.verify(totp_value, time_value)
```

Parameters

- **key** (*bytes-like*) – Per-user secret key. This value must be kept secret and be at least 128 *bits*. It is recommended that the key be 160 bits.
- **length** (*int*) – Length of generated one time password as `int`.
- **algorithm** (`cryptography.hazmat.primitives.hashes.HashAlgorithm`) – A *hashes* instance.
- **time_step** (*int*) – The time step size. The recommended size is 30.
- **enforce_key_length** – A boolean flag defaulting to True that toggles whether a minimum key length of 128 *bits* is enforced. This exists to work around the fact that as documented in Issue #2915, the Google Authenticator PAM module by default generates 80 bit keys. If this flag is set to False, the application develop should implement additional checks of the key length before passing it into *TOTP*.

  Added in version 1.5.

Raises

- **ValueError** – This is raised if the provided `key` is shorter than 128 *bits* or if the `length` parameter is not 6, 7 or 8.
- **TypeError** – This is raised if the provided `algorithm` is not *SHA1()*, *SHA256()* or *SHA512()* or if the `length` parameter is not an integer.

generate(*time*)

Parameters
time (*int*) – The time value used to generate the one time password.

Return bytes
A one time password value.

verify(*totp*, *time*)

Parameters

- **totp** (*bytes*) – The one time password value to validate.
- **time** (*int*) – The time value to validate against.

Raises
cryptography.hazmat.primitives.twofactor.InvalidToken – This is raised when the supplied TOTP does not match the expected TOTP.

get_provisioning_uri(*account_name*, *issuer*)

Added in version 1.0.

Parameters

- **account_name** (*str*) – The display name of account, such as `'Alice Smith'` or `'alice@example.com'`.
- **issuer** (`str` or `None`) – The optional display name of issuer. This is typically the provider or service the user wants to access using the OTP token.

Returns

A URI string.

Provisioning URI

The provisioning URI of HOTP and TOTP is a feature of Google Authenticator and not actually part of the HOTP or TOTP RFCs. However, it is widely supported by web sites and mobile applications which are using Two-Factor authentication.

For generating a provisioning URI you can use the `get_provisioning_uri` method of HOTP/TOTP instances.

```
counter = 5
account_name = 'alice@example.com'
issuer_name = 'Example Inc'

hotp_uri = hotp.get_provisioning_uri(account_name, counter, issuer_name)
totp_uri = totp.get_provisioning_uri(account_name, issuer_name)
```

A common usage is encoding the provisioning URI into QR code and guiding users to scan it with Two-Factor authentication applications in their mobile devices.

2.5 Exceptions

class cryptography.exceptions.**UnsupportedAlgorithm**

Raised when the requested algorithm, or combination of algorithms is not supported.

class cryptography.exceptions.**AlreadyFinalized**

This is raised when a context is used after being finalized.

class cryptography.exceptions.**InvalidSignature**

This is raised when signature verification fails. This can occur with HMAC or asymmetric key signature validation.

class cryptography.exceptions.**NotYetFinalized**

This is raised when the AEAD tag property is accessed on a context before it is finalized.

class cryptography.exceptions.**AlreadyUpdated**

This is raised when additional data is added to a context after update has already been called.

class cryptography.exceptions.**InvalidKey**

This is raised when the verify method of a key derivation function's computed key does not match the expected key.

2.6 Random number generation

When generating random data for use in cryptographic operations, such as an initialization vector for encryption in *CBC* mode, you do not want to use the standard `random` module APIs. This is because they do not provide a cryptographically secure random number generator, which can result in major security issues depending on the algorithms in use.

Therefore, it is our recommendation to always use your operating system's provided random number generator, which is available as `os.urandom()`. For example, if you need 16 bytes of random data for an initialization vector, you can obtain them with:

```
>>> import os
>>> iv = os.urandom(16)
```

If you need your random number as an big integer, you can use `int.from_bytes` to convert the result of `os.urandom`:

```
>>> serial = int.from_bytes(os.urandom(16), byteorder="big")
```

In addition, the Python standard library includes the `secrets` module, which can be used for generating cryptographically secure random numbers, with specific helpers for text-based formats.

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.

2.7 ASN.1

Note

While usable, these APIs should be considered unstable and not yet subject to our backwards compatibility policy.

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.

2.7.1 Tutorial

> **Note**
>
> While usable, these APIs should be considered unstable and not yet subject to our backwards compatibility policy.

The *cryptography.hazmat.asn1* module provides a declarative API for working with ASN.1 data. ASN.1 structures are defined as Python classes with type annotations, and the module uses those definitions to serialize and deserialize instances to and from DER-encoded bytes.

Type mapping

The following table shows how ASN.1 types map to Python types:

ASN.1 type	Python type
BOOLEAN	bool
INTEGER	int
BIT STRING	*BitString*
OCTET STRING	bytes
NULL	*Null*
OBJECT IDENTIFIER	*ObjectIdentifier*
UTF8String	str
PrintableString	*PrintableString*
IA5String	*IA5String*
UTCTime	*UTCTime*
GeneralizedTime	*GeneralizedTime*
SEQUENCE	*@sequence*-decorated class
SEQUENCE OF	list[T]
SET OF	*SetOf*[T]
CHOICE	X | Y | ...
ANY	*TLV*
OPTIONAL	X | None

Defining a SEQUENCE

ASN.1 SEQUENCE types map to Python classes decorated with *@sequence*. Fields are defined as type annotations. For example, given the following ASN.1 definition:

```
Point ::= SEQUENCE {
     x      INTEGER,
     y      INTEGER }
```

The corresponding Python definition is:

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Point:
...     x: int
...     y: int
```

The decorator adds an __init__ with keyword-only parameters:

```
>>> p = Point(x=3, y=7)
>>> p.x
3
```

Encoding and decoding

Use *encode_der()* to serialize an ASN.1 object to DER bytes, and *decode_der()* to deserialize:

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Point:
...     x: int
...     y: int
>>> encoded = asn1.encode_der(Point(x=1, y=2))
>>> encoded
b'0\x06\x02\x01\x01\x02\x01\x02'
>>> point = asn1.decode_der(Point, encoded)
>>> point.x
1
>>> point.y
2
```

Primitive types can also be encoded and decoded directly, without wrapping them in a sequence:

```
>>> asn1.encode_der(42)
b'\x02\x01*'
>>> asn1.decode_der(int, b'\x02\x01*')
42
```

Nested sequences

Sequences can contain other sequences as field types:

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Name:
...     value: str
>>> @asn1.sequence
... class Certificate:
...     version: int
...     subject: Name
>>> cert = Certificate(version=1, subject=Name(value="Alice"))
>>> decoded = asn1.decode_der(Certificate, asn1.encode_der(cert))
>>> decoded.subject.value
'Alice'
```

OPTIONAL fields

A field with a `Union[X, None]` (or `X | None`) type annotation is treated as ASN.1 `OPTIONAL`. When the value is `None`, the field is omitted from the encoding:

```
>>> import typing
>>> from cryptography.hazmat import asn1
```

(continues on next page)

(continued from previous page)

```
>>> @asn1.sequence
... class Record:
...     required: int
...     optional: typing.Union[str, None]
>>> asn1.encode_der(Record(required=1, optional="hi"))
b'0\x07\x02\x01\x01\x0c\x02hi'
>>> asn1.encode_der(Record(required=1, optional=None))
b'0\x03\x02\x01\x01'
```

DEFAULT values

Use *Default* with `typing.Annotated` to specify a default value for a field. When encoding, if the field's value equals the default, it is omitted. When decoding, if the field is absent, the default is used:

```
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class VersionedRecord:
...     version: Annotated[int, asn1.Default(0)]
...     data: bytes
>>> asn1.encode_der(VersionedRecord(version=1, data=b"\x01"))
b'0\x06\x02\x01\x01\x04\x01\x01'
>>> # version=0 equals the default, so it is omitted from the encoding
>>> asn1.encode_der(VersionedRecord(version=0, data=b"\x01"))
b'0\x03\x04\x01\x01'
```

CHOICE fields

A field with a `Union` of multiple non-`None` types is treated as an ASN.1 `CHOICE`. Each variant must have a distinct ASN.1 tag:

```
>>> import typing
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     value: typing.Union[int, bool, str]
>>> asn1.decode_der(Example, asn1.encode_der(Example(value=42))).value
42
>>> asn1.decode_der(Example, asn1.encode_der(Example(value=True))).value
True
```

When multiple alternatives share the same underlying type, a plain union can't distinguish them (`Union[int, int]` is just `int`). Wrap the types with *Variant* and add *Implicit* tags to differentiate between them:

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     field: typing.Union[
...         Annotated[asn1.Variant[int, typing.Literal["IntA"]], asn1.Implicit(0)],
...         Annotated[asn1.Variant[int, typing.Literal["IntB"]], asn1.Implicit(1)],
```

(continues on next page)

(continued from previous page)

```
...     ]
>>> obj = Example(field=asn1.Variant(9, "IntA"))
>>> decoded = asn1.decode_der(Example, asn1.encode_der(obj))
>>> decoded.field.value
9
>>> decoded.field.tag
'IntA'
```

EXPLICIT and IMPLICIT tagging

Use *Explicit* and *Implicit* annotations to apply ASN.1 context-specific tags to fields:

```
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Tagged:
...     explicit_field: Annotated[int, asn1.Explicit(0)]
...     implicit_field: Annotated[int, asn1.Implicit(1)]
>>> encoded = asn1.encode_der(Tagged(explicit_field=5, implicit_field=10))
>>> decoded = asn1.decode_der(Tagged, encoded)
>>> decoded.explicit_field
5
>>> decoded.implicit_field
10
```

Tagging is typically needed to disambiguate `OPTIONAL` fields that would otherwise share the same tag:

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     a: Annotated[typing.Union[int, None], asn1.Implicit(0)]
...     b: Annotated[typing.Union[int, None], asn1.Implicit(1)]
>>> asn1.decode_der(Example, asn1.encode_der(Example(a=9, b=None))).a
9
>>> asn1.decode_der(Example, asn1.encode_der(Example(a=None, b=9))).b
9
```

SEQUENCE OF and SET OF

Use `list`[T] for `SEQUENCE OF` and *SetOf* for `SET OF`:

```
>>> import typing
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class IntList:
...     values: typing.List[int]
>>> decoded = asn1.decode_der(IntList, asn1.encode_der(IntList(values=[1, 2, 3])))
>>> decoded.values
[1, 2, 3]
```

`SET OF` elements are sorted in DER encoding:

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class IntSet:
...     values: asn1.SetOf[int]
>>> decoded = asn1.decode_der(IntSet, asn1.encode_der(IntSet(values=asn1.SetOf([3, 1,␣
↪2]))))
>>> decoded.values.as_list()
[1, 2, 3]
```

Size constraints

Use `Size` to restrict the length of collection and string fields:

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class BoundedList:
...     values: Annotated[typing.List[int], asn1.Size(min=1, max=4)]
>>> asn1.encode_der(BoundedList(values=[1, 2]))
b'0\x080\x06\x02\x01\x01\x02\x01\x02'
```

A real-world example

Here is a more complete example modeling the X.509 `Validity` structure:

```
Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time  }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime }
```

This translates to:

```
>>> import typing
>>> import datetime
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Validity:
...     not_before: typing.Union[asn1.UTCTime, asn1.GeneralizedTime]
...     not_after: typing.Union[asn1.UTCTime, asn1.GeneralizedTime]
>>> not_before = asn1.UTCTime(datetime.datetime(2025, 1, 1, tzinfo=datetime.timezone.
↪utc))
>>> not_after = asn1.UTCTime(datetime.datetime(2026, 1, 1, tzinfo=datetime.timezone.utc))
>>> validity = Validity(not_before=not_before, not_after=not_after)
>>> decoded = asn1.decode_der(Validity, asn1.encode_der(validity))
>>> decoded.not_before.as_datetime().year
2025
>>> decoded.not_after.as_datetime().year
2026
```

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.

2.7.2 ASN.1 Reference

This module provides a declarative interface for defining ASN.1 structures and serializing/deserializing them to/from DER-encoded data.

Note

While usable, these APIs should be considered unstable and not yet subject to our backwards compatibility policy.

Added in version 47.0.0.

Serialization

`cryptography.hazmat.asn1.`**`decode_der`**(*cls*, *data*)

Deserialize a DER-encoded byte string into an instance of `cls`.

Parameters

- **`cls`** (`type`) – The type object representing the ASN.1 class to decode.
- **`data`** (*`bytes`*) – The DER-encoded data.

Returns

An instance of `cls`.

Raises

`ValueError` – If the DER data could not be decoded successfully as an object of type `cls`.

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Point:
...     x: int
...     y: int
>>> point = asn1.decode_der(Point, b'0\x06\x02\x01\x01\x02\x01\x02')
>>> point.x
1
>>> point.y
2
```

`cryptography.hazmat.asn1.`**`encode_der`**(*value*)

Serialize an ASN.1 object into DER-encoded bytes.

Parameters

`value` – The ASN.1 object to encode. Must be an instance of a class decorated with *`sequence()`*, *`set()`*, or *`value_set()`*, or a primitive ASN.1 type (`int`, `bool`, `bytes`, `str`, *`ObjectIdentifier`*, *`PrintableString`*, *`IA5String`*, *`UTCTime`*, *`GeneralizedTime`*, *`BitString`*, *`Null`*).

Returns bytes

The DER-encoded data.

Raises

`ValueError` – If the value could not be encoded.

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Point:
...     x: int
...     y: int
>>> asn1.encode_der(Point(x=1, y=2))
b'0\x06\x02\x01\x01\x02\x01\x02'
```

ASN.1 types

The following built-in Python types are supported as ASN.1 field types:

- `int` – `INTEGER`
- `bool` – `BOOLEAN`
- `bytes` – `OCTET STRING`
- `str` – `UTF8String`

Additionally, *`ObjectIdentifier`* maps to `OBJECT IDENTIFIER`.

Added in version 49.0.0.

`Certificate`, *`CertificateSigningRequest`*, and *`CertificateRevocationList`* can also be used as field types. They are encoded by embedding their DER serialization, and decoded by parsing the field as the corresponding X.509 object. These fields cannot have *`Implicit`* annotations.

Added in version 50.0.0.

`Name` can also be used as a field type. It is encoded and decoded as an X.509 `Name` (a `SEQUENCE OF RelativeDistinguishedName`), and supports *`Implicit`* and *`Explicit`* annotations.

```
from cryptography import x509
from cryptography.hazmat import asn1

@asn1.sequence
class Example:
    cert: x509.Certificate
```

The following decorators and types are provided for the rest of the ASN.1 types that have no direct Python equivalent:

`@cryptography.hazmat.asn1.`**`sequence`**

A class decorator that registers a class as an ASN.1 `SEQUENCE`. Fields are defined as class-level type annotations. The decorator adds an `__init__` method with keyword-only parameters.

Fields can be annotated with *`Explicit`*, *`Implicit`*, *`Default`*, and *`Size`* using `typing.Annotated`.

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class AlgorithmIdentifier:
...     algorithm: int
...     parameters: typing.Union[bool, bytes, None]
>>> encoded = asn1.encode_der(AlgorithmIdentifier(algorithm=9, parameters=None))
```

(continues on next page)

(continued from previous page)

```
>>> asn1.decode_der(AlgorithmIdentifier, encoded).algorithm
9
```

`@cryptography.hazmat.asn1.set`

A class decorator that registers a class as an ASN.1 SET. Fields are defined as class-level type annotations. The decorator adds an `__init__` method with keyword-only parameters.

SET is similar to SEQUENCE, but the fields are encoded in ascending order by tag, rather than in definition order. When decoding, fields must appear in the correct ascending order.

Fields can be annotated with *Explicit*, *Implicit*, *Default*, and *Size* using `typing.Annotated`.

```
>>> from cryptography.hazmat import asn1
>>> @asn1.set
... class Example:
...     x: int
...     y: int
>>> encoded = asn1.encode_der(Example(x=1, y=2))
>>> decoded = asn1.decode_der(Example, encoded)
>>> decoded.x
1
>>> decoded.y
2
>>> # Decoding DER data where fields are not in sorted order
>>> # raises an error:
>>> wrong_order = b'\x31\x06\x02\x01\x02\x02\x01\x01'
>>> asn1.decode_der(Example, wrong_order)
Traceback (most recent call last):
    ...
ValueError: error parsing asn1 value: ...
```

`@cryptography.hazmat.asn1.value_set(value_type)`

A class decorator that registers an `enum.Enum` subclass as an ASN.1 value set: a set of named values of a single underlying type. All the member values must be instances of `value_type`.

Members are encoded exactly as their underlying value. When decoding, the value is decoded and mapped back to the corresponding enum member; decoding fails with `ValueError` if the decoded value does not match any member.

Fields of a value set type can be annotated with *Explicit*, *Implicit*, and *Default* using `typing.Annotated`.

Parameters

value_type (type) – The underlying ASN.1 type of the member values.

```
>>> import enum
>>> from cryptography import x509
>>> from cryptography.hazmat import asn1
>>> @asn1.value_set(x509.ObjectIdentifier)
... class HashAlgorithm(enum.Enum):
...     SHA_256 = x509.ObjectIdentifier("2.16.840.1.101.3.4.2.1")
...     SHA_384 = x509.ObjectIdentifier("2.16.840.1.101.3.4.2.2")
>>> @asn1.sequence
... class Example:
...     algorithm: HashAlgorithm
```

(continues on next page)

(continued from previous page)

```
>>> encoded = asn1.encode_der(Example(algorithm=HashAlgorithm.SHA_256))
>>> asn1.decode_der(Example, encoded).algorithm
<HashAlgorithm.SHA_256: <ObjectIdentifier(oid=2.16.840.1.101.3.4.2.1, name=sha256)>>
```

class cryptography.hazmat.asn1.**PrintableString**(*value*)

Wraps ASN.1 PrintableString values. PrintableString is a restricted subset of ASCII containing only letters, digits, and a limited set of punctuation characters.

Parameters
value (*str*) – The string value. Must contain only characters valid for ASN.1 PrintableString.

Raises
ValueError – If the string contains invalid characters.

as_str()

Returns str
The underlying string value.

class cryptography.hazmat.asn1.**IA5String**(*value*)

Wraps ASN.1 IA5String values. IA5String is equivalent to ASCII.

Parameters
value (*str*) – The string value. Must contain only valid ASCII characters (0–127).

Raises
ValueError – If the string contains invalid characters.

as_str()

Returns str
The underlying string value.

class cryptography.hazmat.asn1.**UTCTime**(*value*)

Wraps ASN.1 UTCTime values. UTCTime represents dates between 1950 and 2049 with second precision.

Parameters
value (datetime.datetime) – An aware datetime object (must have tzinfo set). Year must be between 1950 and 2049. Fractional seconds are not supported.

Raises
ValueError – If the datetime is naive, out of range, or has fractional seconds.

as_datetime()

Returns
datetime.datetime – The underlying datetime value.

class cryptography.hazmat.asn1.**GeneralizedTime**(*value*)

Wraps ASN.1 GeneralizedTime values. GeneralizedTime can represent any date and supports microsecond precision.

Parameters
value (datetime.datetime) – An aware datetime object (must have tzinfo set).

Raises
ValueError – If the datetime is naive.

as_datetime()

Returns
datetime.datetime – The underlying datetime value.

class cryptography.hazmat.asn1.**BitString**(*data*, *padding_bits*)

Wraps ASN.1 BIT STRING values.

Parameters

- **data** (*bytes*) – The raw bit string data.
- **padding_bits** (*int*) – The number of unused bits in the last byte (0–7). The unused bits in the last byte of data must all be zero.

Raises
ValueError – If padding_bits is greater than 7, or if the unused bits in the last byte are not zero.

as_bytes()

Returns bytes
The raw bit string data.

padding_bits()

Returns int
The number of unused bits in the last byte.

class cryptography.hazmat.asn1.**Null**

Represents the ASN.1 NULL type. NULL has no value and is encoded as a zero-length element.

class cryptography.hazmat.asn1.**TLV**

Represents a raw ASN.1 Tag-Length-Value element. This is useful for decoding ASN.1 ANY fields, deferring the decoding of parts of a structure, or for handling fields whose type is not known at definition time.

TLV cannot be directly constructed. It is obtained by decoding DER data. It cannot have *Implicit* or *Default* annotations, and cannot be optional (TLV | None).

tag_bytes

The raw tag bytes of the element.

Type
bytes

data

The raw content bytes (value) of the element, excluding the tag and length.

Type
memoryview

parse(*cls*)

Parse the TLV data as an instance of cls.

Parameters
cls (type) – The type to decode the TLV contents as.

Returns
An instance of cls.

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Inner:
...     foo: int
>>> # Decode a SEQUENCE's raw bytes as a TLV
>>> raw = asn1.decode_der(asn1.TLV, asn1.encode_der(Inner(foo=9)))
>>> raw.tag_bytes
b'0'
>>> inner = raw.parse(Inner)
>>> inner.foo
9
```

class cryptography.hazmat.asn1.**SetOf**(*values*)

Represents an ASN.1 `SET OF`, an unordered collection of elements of a single type.

Parameters

values (*list*) – The list of values.

as_list()

Returns list

The underlying list of values.

```
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     values: asn1.SetOf[int]
>>> encoded = asn1.encode_der(Example(values=asn1.SetOf([3, 1, 2])))
>>> decoded = asn1.decode_der(Example, encoded)
>>> decoded.values.as_list()
[1, 2, 3]
```

class cryptography.hazmat.asn1.**Variant**(*value*, *tag*)

A tagged variant for `CHOICE` fields where multiple alternatives share the same underlying type.

When a `CHOICE` field has multiple alternatives with different types, use a `Union` directly. When multiple alternatives share the same underlying type, use `Variant` to distinguish between them with a string tag.

Parameters

- **value** – The actual value.
- **tag** (*str*) – A string tag identifying which variant this is. Must match a `typing.Literal` type parameter in the type annotation.

value

The wrapped value.

tag

The string tag identifying which variant this is.

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
```

(continues on next page)

(continued from previous page)

```
...     field: typing.Union[
...         Annotated[asn1.Variant[int, typing.Literal["IntA"]], asn1.Implicit(0)],
...         Annotated[asn1.Variant[int, typing.Literal["IntB"]], asn1.Implicit(1)],
...     ]
>>> obj = Example(field=asn1.Variant(9, "IntA"))
>>> encoded = asn1.encode_der(obj)
>>> decoded = asn1.decode_der(Example, encoded)
>>> decoded.field.value
9
>>> decoded.field.tag
'IntA'
```

A field with a `Union[X, None]` (or `X | None`) type annotation is treated as ASN.1 `OPTIONAL`. When the value is `None`, the field is omitted from the encoding.

```
>>> import typing
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     required: int
...     optional: typing.Union[str, None]
>>> with_value = asn1.encode_der(Example(required=1, optional="hi"))
>>> without_value = asn1.encode_der(Example(required=1, optional=None))
>>> asn1.decode_der(Example, without_value).optional is None
True
```

A field with a `Union` of multiple non-`None` types is treated as an ASN.1 `CHOICE`. Each variant in the union must have a distinct ASN.1 tag.

```
>>> import typing
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     value: typing.Union[int, bool, str]
>>> asn1.decode_der(Example, asn1.encode_der(Example(value=42))).value
42
```

Annotations

Annotations are applied to fields using `typing.Annotated` to control ASN.1 encoding behavior.

class `cryptography.hazmat.asn1.`**`Explicit`**(*tag*)

An annotation that applies `EXPLICIT` tagging to a field. `EXPLICIT` tagging wraps the original encoding in a new tag.

Parameters

tag (*int*) – The context-specific tag number.

```
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     value: Annotated[int, asn1.Explicit(0)]
```

(continues on next page)

(continued from previous page)

```
>>> asn1.encode_der(Example(value=9))
b'0\x05\xa0\x03\x02\x01\t'
```

class cryptography.hazmat.asn1.**Implicit**(*tag*)

An annotation that applies `IMPLICIT` tagging to a field. `IMPLICIT` tagging replaces the original tag with a context-specific tag.

Cannot be used with `CHOICE` types or *TLV* fields.

Parameters

tag (*int*) – The context-specific tag number.

```
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     value: Annotated[int, asn1.Implicit(0)]
>>> asn1.encode_der(Example(value=9))
b'0\x03\x80\x01\t'
```

class cryptography.hazmat.asn1.**Default**(*value*)

An annotation that specifies a `DEFAULT` value for a field. When encoding, if the field's value equals the default, it is omitted from the output. When decoding, if the field is absent, the default value is used.

Cannot be used with `OPTIONAL` (`X | None`) fields or *TLV* fields.

Parameters

value – The default value for the field.

```
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     version: Annotated[int, asn1.Default(0)]
...     data: bytes
>>> encoded = asn1.encode_der(Example(version=0, data=b"\x01"))
>>> encoded
b'0\x03\x04\x01\x01'
>>> asn1.decode_der(Example, encoded).version
0
```

class cryptography.hazmat.asn1.**Size**(*min*, *max*)

An annotation that applies a size constraint to a field. Supported on `SEQUENCE OF`, `SET OF`, `BIT STRING`, `OCTET STRING`, `UTF8String`, `PrintableString`, and `IA5String` fields.

Parameters

- **min** (*int*) – The minimum size (inclusive).
- **max** (`int` or `None`) – The maximum size (inclusive), or `None` for no upper bound.

static exact(*n*)

Create a *Size* constraint where the minimum and maximum are both **n**.

Parameters

n (*int*) – The exact size required.

Returns
Size

```
>>> import typing
>>> from typing import Annotated
>>> from cryptography.hazmat import asn1
>>> @asn1.sequence
... class Example:
...     values: Annotated[typing.List[int], asn1.Size(min=1, max=4)]
>>> asn1.encode_der(Example(values=[1, 2]))
b'0\x080\x06\x02\x01\x01\x02\x01\x02'
```

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.

2.8 Decrepit cryptography

This module holds old, deprecated, and/or insecure cryptographic algorithms that may be needed in exceptional cases for backwards compatibility or interoperability reasons. Unless necessary their use is **strongly discouraged**.

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.

2.8.1 Decrepit Symmetric algorithms

This module contains decrepit symmetric encryption algorithms. These are algorithms that should not be used unless necessary for backwards compatibility or interoperability with legacy systems. Their use is **strongly discouraged**.

These algorithms require you to use a `Cipher` object along with the appropriate `modes`.

class `cryptography.hazmat.decrepit.ciphers.algorithms.`**`ARC4`**(*key*)

Added in version 43.0.0.

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.

```
>>> import os
>>> from cryptography.hazmat.decrepit.ciphers.algorithms import ARC4
>>> from cryptography.hazmat.primitives.ciphers import Cipher, modes
>>> key = os.urandom(16)
>>> algorithm = ARC4(key)
>>> cipher = Cipher(algorithm, mode=None)
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message")
```

(continues on next page)

(continued from previous page)

```
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct)
b'a secret message'
```

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

Added in version 43.0.0.

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.

Deprecated since version 47.0.0: Passing 64-bit or 128-bit keys is deprecated. In a future release, only 192-bit (24-byte) keys will be accepted. Users should expand shorter keys themselves (e.g., for single DES: `key + key + key`, for two-key: `key + key[:8]`).

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

Added in version 43.0.0.

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.

```
>>> import os
>>> from cryptography.hazmat.decrepit.ciphers.algorithms import CAST5
>>> from cryptography.hazmat.primitives.ciphers import Cipher, modes
>>> key = os.urandom(16)
>>> iv = os.urandom(8)
>>> algorithm = CAST5(key)
>>> cipher = Cipher(algorithm, modes.CBC(iv))
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message")
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct)
b'a secret message'
```

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

Added in version 43.0.0.

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.decrepit.ciphers.algorithms.`**`Blowfish`**(*key*)

Added in version 43.0.0.

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.

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.decrepit.ciphers.algorithms.`**`IDEA`**(*key*)

Added in version 43.0.0.

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.

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.

2.8.2 Decrepit Cipher Modes

This module contains decrepit cipher modes. These modes should not be used unless necessary for backwards compatibility or interoperability with legacy systems. Their use is **strongly discouraged**.

These modes work with `Cipher` objects in the same way as modes from `modes`.

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

Added in version 47.0.0.

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 and unpredictable. It must be the same length as the cipher's block size.

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

Added in version 47.0.0.

CFB8 (Cipher Feedback with 8-bit segment size) is a mode of operation similar to CFB. It operates on 8-bit segments rather than full blocks.

This mode does not require padding.

Parameters

initialization_vector (*bytes-like*) – Must be random and unpredictable. It must be the same length as the cipher's block size.

class `cryptography.hazmat.decrepit.ciphers.modes.`**`OFB`**(*initialization_vector*)

Added in version 47.0.0.

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 and unpredictable. It must be the same length as the cipher's block size.

2.9 Installation

You can install `cryptography`:

```
$ pip install cryptography
```

```
$ uv add cryptography
```

If this does not work please **upgrade your pip** first, as that is the single most common cause of installation problems.

2.9.1 Supported platforms

Currently we test `cryptography` on Python 3.9+ and PyPy3.11 on these operating systems.

- x86-64 CentOS Stream 9, 10
- x86-64 Fedora (latest)
- ARM64 macOS 26 Tahoe
- x86-64 Ubuntu 22.04, 24.04, 26.04, and rolling
- ARM64, ARMv7l, and `ppc64le` Ubuntu rolling
- x86-64 Debian Bookworm (12.x), Trixie (13.x), and Sid (unstable)
- x86-64 and ARM64 Alpine (latest)
- 64-bit Python on 64-bit Windows Server 2025

We test compiling with `clang` as well as `gcc` and use the following OpenSSL releases in addition to distribution provided releases from the above supported platforms:

- `OpenSSL 3.0-latest`
- `OpenSSL 3.4-latest`
- `OpenSSL 3.5-latest`
- `OpenSSL 3.6-latest`
- `OpenSSL 4.0-latest`

We also test against the latest commit of BoringSSL, the latest `aws-lc` release, and versions of LibreSSL that are receiving security support at the time of a given `cryptography` release.

2.9.2 Building cryptography on Windows

The wheel package on Windows is a statically linked build (as of 0.5) so all dependencies are included. To install `cryptography`, you will typically just run

```
$ pip install cryptography
```

If you prefer to compile it yourself you'll need to have OpenSSL installed. You can compile OpenSSL yourself as well or use a binary distribution. Be sure to download the proper version for your architecture and Python (VC2015 is required for 3.9 and above). Wherever you place your copy of OpenSSL you'll need to set the `OPENSSL_DIR` environment variable to include the proper location. For example:

```
C:\> \path\to\vcvarsall.bat x86_amd64
C:\> set OPENSSL_DIR=C:\OpenSSL-win64
C:\> pip install cryptography
```

You will also need to have *Rust installed and available*.

If you need to rebuild `cryptography` for any reason be sure to clear the local wheel cache.

2.9.3 Building cryptography on Linux

Note

You should **upgrade pip** and attempt to install `cryptography` again before following the instructions to compile it below. Most Linux platforms will receive a binary wheel and require no compiler if you have an updated `pip`!

`cryptography` ships `manylinux` wheels (as of 2.0) so all dependencies are included. For users on **pip 19.3** or above running on a `manylinux2014` (or greater) compatible distribution (or **pip 21.2.4** for `musllinux`) all you should need to do is:

```
$ pip install cryptography
```

If you want to compile `cryptography` yourself you'll need a C compiler, a Rust compiler, headers for Python (if you're not using `pypy`), and headers for the OpenSSL and `libffi` libraries available on your system.

On all Linux distributions you will need to have *Rust installed and available*.

Warning

The Rust available by default in Alpine < 3.21 is older than the minimum supported version. See the *Rust installation instructions* for information about installing a newer Rust.

```
$ sudo apk add gcc musl-dev python3-dev libffi-dev openssl-dev cargo pkgconfig
```

If you get an error with `openssl-dev` you may have to use `libressl-dev`.

Warning

The Rust available in Debian versions prior to Trixie are older than the minimum supported version. See the *Rust installation instructions* for information about installing a newer Rust.

```
$ sudo apt-get install build-essential libssl-dev libffi-dev \
    python3-dev cargo pkg-config
```

> **Warning**
>
> For RHEL and CentOS you must be on version 9.6 or newer for the command below to install a sufficiently new Rust. If your Rust is less than 1.83.0 please see the *Rust installation instructions* for information about installing a newer Rust.

```
$ sudo dnf install redhat-rpm-config gcc libffi-devel python3-devel \
    openssl-devel cargo pkg-config
```

Building

You should now be able to build and install cryptography. To avoid getting the pre-built wheel on `manylinux` compatible distributions you'll need to use `--no-binary`.

```
$ pip install cryptography --no-binary cryptography
```

Using your own OpenSSL on Linux

Python links to OpenSSL for its own purposes and this can sometimes cause problems when you wish to use a different version of OpenSSL with cryptography. If you want to use cryptography with your own build of OpenSSL you will need to make sure that the build is configured correctly so that your version of OpenSSL doesn't conflict with Python's.

The options you need to add allow the linker to identify every symbol correctly even when multiple versions of the library are linked into the same program. If you are using your distribution's source packages these will probably be patched in for you already, otherwise you'll need to use options something like this when configuring OpenSSL:

```
$ ./config -Wl,-Bsymbolic-functions -fPIC shared
```

Static Wheels

Cryptography ships statically-linked wheels for macOS, Windows, and Linux (via `manylinux` and `musllinux`). This allows compatible environments to use the most recent OpenSSL, regardless of what is shipped by default on those platforms.

If you are using a platform not covered by our wheels, you can build your own statically-linked wheels that will work on your own systems. This will allow you to continue to use relatively old Linux distributions (such as LTS releases), while making sure you have the most recent OpenSSL available to your Python programs.

To do so, you should find yourself a machine that is as similar as possible to your target environment (e.g. your production environment): for example, spin up a new cloud server running your target Linux distribution. On this machine, install the Cryptography dependencies as mentioned in *Building cryptography on Linux*. Please also make sure you have virtualenv installed: this should be available from your system package manager.

Then, paste the following into a shell script. You'll need to populate the `OPENSSL_VERSION` variable. To do that, visit openssl.org and find the latest non-FIPS release version number, then set the string appropriately. For example, for OpenSSL 3.0.9, use `OPENSSL_VERSION="3.0.9"`.

When this shell script is complete, you'll find a collection of wheel files in a directory called `wheelhouse`. These wheels can be installed by a sufficiently-recent version of `pip`. The Cryptography wheel in this directory contains a statically-linked OpenSSL binding, which ensures that you have access to the most-recent OpenSSL releases without corrupting your system dependencies.

```
set -e
```

(continues on next page)

(continued from previous page)

```
OPENSSL_VERSION="VERSIONGOESHERE"
CWD=$(pwd)

virtualenv env
. env/bin/activate
pip install -U setuptools
pip install -U wheel pip
curl -LO https://www.openssl.org/source/openssl-${OPENSSL_VERSION}.tar.gz
tar xzvf openssl-${OPENSSL_VERSION}.tar.gz
cd openssl-${OPENSSL_VERSION}
./config no-shared no-ssl2 no-ssl3 -fPIC --prefix=${CWD}/openssl
make && make install
cd ..
OPENSSL_DIR="${CWD}/openssl" pip wheel --no-cache-dir --no-binary cryptography␣
↪cryptography
```

2.9.4 Building cryptography on macOS

Note

If installation gives a `fatal error: 'openssl/aes.h' file not found` see the *FAQ* for information about how to fix this issue.

The wheel package on macOS is a statically linked build (as of 1.0.1) so for users with pip 8 or above you only need one step:

```
$ pip install cryptography
```

If you want to build cryptography yourself or are on an older macOS version, cryptography requires the presence of a C compiler, development headers, and the proper libraries. On macOS much of this is provided by Apple's Xcode development tools. To install the Xcode command line tools (on macOS 10.10+) open a terminal window and run:

```
$ xcode-select --install
```

This will install a compiler (clang) along with (most of) the required development headers.

You will also need to have *Rust installed and available*, which can be obtained from Homebrew, MacPorts, or directly from the Rust website.

Finally you need OpenSSL, which you can obtain from Homebrew or MacPorts. Cryptography does **not** support the OpenSSL/LibreSSL libraries Apple ships in its base operating system.

To build cryptography and dynamically link it:

Homebrew

```
$ brew install openssl@3 rust
$ pip install --no-binary cryptography cryptography
```

MacPorts:

```
$ sudo port install openssl rust
$ env OPENSSL_DIR="-L/opt/local" pip install --no-binary cryptography cryptography
```

You can also build cryptography statically:

Homebrew

```
$ brew install openssl@3 rust
$ env OPENSSL_STATIC=1 pip install --no-binary cryptography cryptography
```

MacPorts:

```
$ sudo port install openssl rust
$ env OPENSSL_STATIC=1 OPENSSL_DIR="/opt/local" pip install --no-binary cryptography␣
↪cryptography
```

If you need to rebuild `cryptography` for any reason be sure to clear the local wheel cache.

2.9.5 Building with BoringSSL, LibreSSL, or AWS-LC

To build against BoringSSL, LibreSSL, or AWS-LC instead of OpenSSL, you can set the `OPENSSL_DIR` environment variable to point to your BoringSSL, LibreSSL, or AWS-LC installation directory.

2.9.6 Rust

> **Note**
>
> If you are using Linux, then you should **upgrade pip** (in a virtual environment!) and attempt to install `cryptography` again before trying to install the Rust toolchain. On most Linux distributions, the latest version of `pip` will be able to install a binary wheel, so you won't need a Rust toolchain.

Building `cryptography` requires having a working Rust toolchain. The current minimum supported Rust version is 1.83.0. **This is newer than the Rust some package managers ship**, so users may need to install with the instructions below.

Instructions for installing Rust can be found on the Rust Project's website. We recommend installing Rust with `rustup` (as documented by the Rust Project) in order to ensure you have a recent version.

Rust is only required when building `cryptography`, meaning that you may install it for the duration of your `pip install` command and then remove it from a system. A Rust toolchain is not required to **use** `cryptography`. In deployments such as `docker`, you may use a multi-stage `Dockerfile` where you install Rust during the build phase but do not install it in the runtime image. This is the same as the C compiler toolchain which is also required to build `cryptography`, but not afterwards.

2.10 Changelog

2.10.1 51.0.0 - main

> **Note**
>
> This version is not yet released and is under active development.

2.10.2 50.0.1 - 2026-08-25

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 4.0.2.

2.10.3 50.0.0 - 2026-07-31

- **SECURITY ISSUE**: *pkcs7_decrypt_der()* and its PEM and S/MIME variants no longer expose distinguishable errors or timing when unwrapping a `RecipientInfo`'s `encryptedKey`, which could act as a Bleichenbacher oracle for callers that decrypt untrusted messages. A random key is now substituted on failure, as described in **RFC 3218**. Credit to **@X1AOxiang** for reporting the issue. **CVE-2026-69247**
- Deprecated Diffie-Hellman key exchange over finite fields (FFDH). Everything FFDH is deprecated, including the types in `cryptography.hazmat.primitives.asymmetric.dh` and loading FFDH keys or parameters with the key loading APIs. Users should migrate to a more modern key exchange algorithm.
- Added `xof()` class methods to *SHAKE128* and *SHAKE256* for constructing algorithm instances configured for use with *XOFHash*.
- The *X.509 verification* APIs are now considered stable and are subject to our API stability policy.
- Added the *Cobblestone (streaming symmetric encryption)* recipe, an implementation of the Cobblestone-128 and Cobblestone-256 instantiations of the C2SP chunked-encryption specification for streaming authenticated encryption of large messages.
- Parsing a Signed Certificate Timestamp list now rejects encodings that carry trailing bytes after the list or after an individual SCT, instead of silently ignoring them.
- Added support for using *Name* as a field type in the *ASN.1* module.
- Loading a public key or an EC private key now rejects DER where the `subjectPublicKey` (or EC `publicKey`) `BIT STRING` declares a non-zero number of unused bits, instead of silently ignoring it.
- Parsing a CRL entry's `InvalidityDate` extension now rejects a `GeneralizedTime` that carries fractional seconds or another non-DER form, matching the strict encoding already required for every other X.509 time field.
- *load_der_ocsp_request()* and *load_der_ocsp_response()* now reject a request or response whose `version` field is not `v1`, the only version defined by RFC 6960, matching the version validation already performed when loading certificates, CSRs and CRLs.
- *XOFHash* is now supported when building against AWS-LC.
- HMAC (and therefore PBKDF2-HMAC) with SHA-3 hashes is now supported when building against AWS-LC.
- Diffie-Hellman (*Diffie-Hellman key exchange*) is now supported when building against AWS-LC.
- *load_der_public_key()* and *load_pem_public_key()* now reject Diffie-Hellman public keys whose modulus is smaller than 512 bits, matching the minimum already enforced when loading DH private keys and when constructing *DHParameterNumbers*.
- Added *MLDSAMuHasher* for incrementally computing the ML-DSA `mu` (message representative) used by the external-mu signing and verification APIs.
- The builtin *HashAlgorithm* classes and the classes in *padding* can now be compared with ==.
- *CertificateBuilder* now supports creating unsigned certificates (**RFC 9925**) with the `create_unsigned` method.
- The *X.509 verification* APIs now permit ML-DSA-44, ML-DSA-65, and ML-DSA-87 (**RFC 9881**) public keys and signatures by default.

2.10.4 49.0.0 - 2026-06-12

- **BACKWARDS INCOMPATIBLE:** Support for `x86_64` macOS has been removed. We now only publish `arm64` wheels for macOS.
- **BACKWARDS INCOMPATIBLE:** Support for 32-bit Windows has been removed. Users should move to a 64-bit Python installation.

- **BACKWARDS INCOMPATIBLE:** Removed the deprecated `PUBLIC_KEY_TYPES`, `PRIVATE_KEY_TYPES`, `CERTIFICATE_PRIVATE_KEY_TYPES`, `CERTIFICATE_ISSUER_PUBLIC_KEY_TYPES`, and `CERTIFICATE_PUBLIC_KEY_TYPES` type aliases. Use `PublicKeyTypes`, `PrivateKeyTypes`, `CertificateIssuerPrivateKeyTypes`, `CertificateIssuerPublicKeyTypes`, and `CertificatePublicKeyTypes` instead. These were deprecated in version 40.0.
- **BACKWARDS INCOMPATIBLE:** *ChaCha20* now treats the first 4 bytes of the `nonce` as a 32-bit little-endian block counter (as defined in **RFC 7539**) and tracks the number of bytes processed. Attempting to encrypt or decrypt more data than the counter allows before it would overflow now raises a `ValueError` rather than silently diverging from RFC 7539. Setting the counter portion of the `nonce` to zero allows encrypting up to 256 GiB with a given nonce.
- **BACKWARDS INCOMPATIBLE:** Loading an X.509 certificate whose ECDSA or DSA signature `AlgorithmIdentifier` contains encoded NULL parameters now raises a `ValueError`. Such certificates are invalid, but older versions of Java emitted them; previously they loaded with a deprecation warning.
- Fixed cross-compilation of the CFFI bindings when `PYO3_CROSS_LIB_DIR` is set. The build now derives the Python include directory from `PYO3_CROSS_LIB_DIR` instead of querying the host interpreter, which previously caused the build to fail during cross-compilations for embedded systems, on hosts which have same-version Python development headers installed as the target Python.
- Added support for signing and verifying X.509 certificates, certificate signing requests, and certificate revocation lists with *ML-DSA signing* keys, as well as loading certificates that contain ML-DSA public keys.
- Added *enc_length()* to *KEM* so callers can split the encapsulated key from the ciphertext returned by *encrypt()*.
- *require_present()*, *may_be_present()*, and *require_not_present()* now accept any extension type. Previously only a fixed set of extension types was supported, which made it impossible to account for otherwise unrecognized critical extensions during path validation.
- Added support for using *Certificate*, *CertificateSigningRequest*, and *CertificateRevocationList* as field types in *ASN.1* structures.
- Added *value_set()*, a class decorator that registers an `enum.Enum` subclass as an ASN.1 value set: members are encoded as their underlying value, and decoding fails if the decoded value does not match one of the declared members.
- Added *from_bytes()* for parsing a *Name* from DER bytes, the inverse of *public_bytes()*.
- Added the `rsa_padding` keyword-only parameter to *public_key()*. Passing the *PSS* class (not an instance) encodes an RSA subject public key in the certificate's `subjectPublicKeyInfo` with the `id-RSASSA-PSS` OID and no parameters.
- Added external mu (message representative) support to *ML-DSA signing* via the `sign_mu` and `verify_mu` methods, which sign and verify a precomputed 64-byte `mu` as defined in FIPS 204.

2.10.5 48.0.1 - 2026-06-09

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 4.0.1.

2.10.6 48.0.0 - 2026-05-04

- **BACKWARDS INCOMPATIBLE:** Support for Python 3.8 has been removed. `cryptography` now requires Python 3.9 or later.
- **BACKWARDS INCOMPATIBLE:** Loading an X.509 CRL whose inner `TBSCertList.signature` algorithm does not match the outer `signatureAlgorithm` now raises `ValueError`. Previously, such CRLs were parsed successfully and only rejected during signature validation.

- Added support for *ML-KEM key encapsulation* and *ML-DSA signing* when using OpenSSL 3.5.0 or later, in addition to the existing AWS-LC and BoringSSL support. This means post-quantum algorithms are now available to users of our wheels.
  - **Note:** Going forward, we do not guarantee that all functionality in `cryptography` will be available when building against OpenSSL. See *The State of OpenSSL for pyca/cryptography* for more information.

2.10.7 47.0.0 - 2026-04-24

- Support for Python 3.8 is deprecated and will be removed in the next `cryptography` release.
- **BACKWARDS INCOMPATIBLE:** Support for binary elliptic curves (`SECT*` classes) has been removed. These curves are rarely used and have additional security considerations that make them undesirable.
- **BACKWARDS INCOMPATIBLE:** Support for OpenSSL 1.1.x has been removed. OpenSSL 3.0.0 or later is now required. LibreSSL, BoringSSL, and AWS-LC continue to be supported.
- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 4.1.
- **BACKWARDS INCOMPATIBLE:** Loading keys with unsupported algorithms or keys with unsupported explicit curve encodings now raises *`UnsupportedAlgorithm`* instead of `ValueError`. This change affects *`load_pem_private_key()`*, *`load_der_private_key()`*, *`load_pem_public_key()`*, *`load_der_public_key()`*, and *`public_key()`* when called on certificates with unsupported public key algorithms.
- **BACKWARDS INCOMPATIBLE:** When parsing elliptic curve private keys, we now reject keys that incorrectly encode a private key of the wrong length because such keys are impossible to process in a constant-time manner. We do not believe keys with this problem are in wide use, however we may revert this change based on the feedback we receive.
- Deprecated passing 64-bit (8-byte) and 128-bit (16-byte) keys to *`TripleDES`*. In a future release, only 192-bit (24-byte) keys will be accepted. Users should expand shorter keys themselves (e.g., for single DES: `key + key + key`, for two-key: `key + key[:8]`).
- Updated the minimum supported Rust version (MSRV) to 1.83.0, from 1.74.0.
- Support for `x86_64` macOS (including publishing wheels) is deprecated and will be removed in the next release. We will switch to publishing an `arm64` only wheel for macOS.
- Support for 32-bit Windows (including publishing wheels) is deprecated and will be removed in the next release. Users should move to a 64-bit Python installation.
- `public_bytes` and `private_bytes` methods on keys now raise `TypeError` (instead of `ValueError`) if an invalid encoding is provided for the given `format`.
- Moved *`CFB`*, *`OFB`*, and *`CFB8`* into *Decrepit cryptography* and deprecated them in the `modes` module. They will be removed from the `modes` module in 49.0.0.
- Moved *`Camellia`* into *Decrepit cryptography* and deprecated it in the `cipher` module. It will be removed from the `cipher` module in 49.0.0.
- Added *`extract()`* to *`HKDF`*. The previous private implementation will be removed in 49.0.0.
- Added support for loading elliptic curve keys that contain explicit encodings of the curves `secp256r1`, `secp384r1`, and `secp521r1`.
- Added support for *`Argon2d`* and *`Argon2i`* when using OpenSSL 3.2.0+.
- Added `derive_into` methods to *`HKDF`*, *`HKDFExpand`*, *`ConcatKDFHash`*, *`ConcatKDFHMAC`*, *`Argon2id`*, *`PBKDF2HMAC`*, *`KBKDFHMAC`*, *`KBKDFCMAC`*, *`Scrypt`*, and *`X963KDF`* to allow deriving keys directly into pre-allocated buffers.

- Added `encrypt_into` and `decrypt_into` methods to *AESCCM*, *AESGCM*, *AESGCMSIV*, *AESOCB3*, *AESSIV*, and *ChaCha20Poly1305* to allow encrypting directly into a pre-allocated buffer.
- Added support for PKCS1v15 signing without DigestInfo using *NoDigestInfo*.
- Added *hash()*, a one-shot method for computing hashes.
- Added *HPKE (Hybrid Public Key Encryption)* support implementing **RFC 9180** for hybrid authenticated encryption.
- Added new *ML-KEM key encapsulation* module with support for ML-KEM key encapsulation with AWS-LC and BoringSSL.
  - **Note:** Post-quantum algorithm support requires AWS-LC or BoringSSL. As we ship our wheels with OpenSSL, most users will not have access to these APIs yet. See *The State of OpenSSL for pyca/cryptography* for more information on OpenSSL support.
- Added new *ML-DSA signing* module with support for ML-DSA signing and verification with AWS-LC and BoringSSL.
  - **Note:** Post-quantum algorithm support requires AWS-LC or BoringSSL. As we ship our wheels with OpenSSL, most users will not have access to these APIs yet. See *The State of OpenSSL for pyca/cryptography* for more information on OpenSSL support.
- Added new *ASN.1* module with support for declaratively defining custom ASN.1 types and encoding/decoding them.
- Fixed compilation when using LibreSSL 4.3.0 and OpenSSL 4.0.0.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 4.0.0.

2.10.8 46.0.7 - 2026-04-07

- **SECURITY ISSUE**: Fixed an issue where non-contiguous buffers could be passed to APIs that accept Python buffers, which could lead to buffer overflow. **CVE-2026-39892**
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.6.

2.10.9 46.0.6 - 2026-03-25

- **SECURITY ISSUE**: Fixed a bug where name constraints were not applied to peer names during verification when the leaf certificate contains a wildcard DNS SAN. Ordinary X.509 topologies are not affected by this bug, including those used by the Web PKI. Credit to **Oleh Konko (1seal)** for reporting the issue. **CVE-2026-34073**

2.10.10 46.0.5 - 2026-02-10

- An attacker could create a malicious public key that reveals portions of your private key when using certain uncommon elliptic curves (binary curves). This version now includes additional security checks to prevent this attack. This issue only affects binary elliptic curves, which are rarely used in real-world applications. Credit to **XlabAI Team of Tencent Xuanwu Lab and Atuin Automated Vulnerability Discovery Engine** for reporting the issue. **CVE-2026-26007**
- Support for `SECT*` binary elliptic curves is deprecated and will be removed in the next release.

2.10.11 46.0.4 - 2026-01-27

- Dropped support for win_arm64 wheels.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.5.

2.10.12 46.0.3 - 2025-10-15

- Fixed compilation when using LibreSSL 4.2.0.

2.10.13 46.0.2 - 2025-09-30

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.4.

2.10.14 46.0.1 - 2025-09-16

- Fixed an issue where users installing via `pip` on Python 3.14 development versions would not properly install a dependency.
- Fixed an issue building the free-threaded macOS 3.14 wheels.

2.10.15 46.0.0 - 2025-09-16

- **BACKWARDS INCOMPATIBLE:** Support for Python 3.7 has been removed.
- Support for OpenSSL < 3.0 is deprecated and will be removed in the next release.
- Support for `x86_64` macOS (including publishing wheels) is deprecated and will be removed in two releases. We will switch to publishing an `arm64` only wheel for macOS.
- Support for 32-bit Windows (including publishing wheels) is deprecated and will be removed in two releases. Users should move to a 64-bit Python installation.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.3.
- We now build `ppc64le` `manylinux` wheels and publish them to PyPI.
- We now build `win_arm64` (Windows on Arm) wheels and publish them to PyPI.
- Added support for free-threaded Python 3.14.
- Removed the deprecated `get_attribute_for_oid` method on *`CertificateSigningRequest`*. Users should use *`get_attribute_for_oid()`* instead.
- Removed the deprecated `CAST5`, `SEED`, `IDEA`, and `Blowfish` classes from the cipher module. These are still available in *Decrepit cryptography*.
- In X.509, when performing a PSS signature with a SHA-3 hash, it is now encoded with the official NIST SHA3 OID.

2.10.16 45.0.7 - 2025-09-01

- Added a function to support an upcoming `pyOpenSSL` release.

2.10.17 45.0.6 - 2025-08-05

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.2.

2.10.18 45.0.5 - 2025-07-02

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.1.

2.10.19 45.0.4 - 2025-06-09

- Fixed decrypting PKCS#8 files encrypted with SHA1-RC4. (This is not considered secure, and is supported only for backwards compatibility.)

2.10.20 45.0.3 - 2025-05-25

- Fixed decrypting PKCS#8 files encrypted with long salts (this impacts keys encrypted by Bouncy Castle).
- Fixed decrypting PKCS#8 files encrypted with DES-CBC-MD5. While wildly insecure, this remains prevalent.

2.10.21 45.0.2 - 2025-05-17

- Fixed using `mypy` with `cryptography` on older versions of Python.

2.10.22 45.0.1 - 2025-05-17

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.5.0.

2.10.23 45.0.0 - 2025-05-17 (YANKED)

- Support for Python 3.7 is deprecated and will be removed in the next `cryptography` release.
- Updated the minimum supported Rust version (MSRV) to 1.74.0, from 1.65.0.
- Added support for serialization of PKCS#12 Java truststores in *serialize_java_truststore()*
- Added *derive_phc_encoded()* and *verify_phc_encoded()* methods to support password hashing in the PHC string format
- Added support for PKCS7 decryption and encryption using AES-256 as the content algorithm, in addition to AES-128.
- **BACKWARDS INCOMPATIBLE:** Made SSH private key loading more consistent with other private key loading: *load_ssh_private_key()* now raises a `TypeError` if the key is unencrypted but a password is provided (previously no exception was raised), and raises a `TypeError` if the key is encrypted but no password is provided (previously a `ValueError` was raised).
- Added `__copy__` to the *EllipticCurvePrivateKey*, *EllipticCurvePublicKey*, *Ed25519PublicKey*, *Ed25519PrivateKey*, *Ed448PublicKey*, *Ed448PrivateKey*, *X25519PublicKey*, *X25519PrivateKey*, *X448PublicKey*, *X448PrivateKey*, *RSAPrivateKey*, *RSAPublicKey*, *DSAPrivateKey*, *DSAPublicKey*, *DHPrivateKey*, and *DHPublicKey* abstract base classes.
- We significantly refactored how private key loading (*load_pem_private_key()* and *load_der_private_key()*) works. This is intended to be backwards compatible for all well-formed keys, therefore if you discover a key that now raises an exception, please file a bug with instructions for reproducing.
- Added `unsafe_skip_rsa_key_validation` keyword-argument to *load_ssh_private_key()*.
- Added *XOFHash* to support repeated *squeeze()* operations on extendable output functions.
- Added *add_response_by_hash()* method to allow creating OCSP responses using certificate hash values rather than full certificates.
- Extended the *X.509 path validation* API to support user-configured extension policies via the *PolicyBuilder.extension_policies* method.
- Deprecated the `subject`, `verification_time` and `max_chain_depth` properties on *ClientVerifier* and *ServerVerifier* in favor of a new `policy` property. These properties will be removed in the next release of `cryptography`.

- **BACKWARDS INCOMPATIBLE:** The `VerifiedClient.subject` property can now be *None* since a custom extension policy may allow certificates without a Subject Alternative Name extension.
- Changed the behavior when the OpenSSL 3 legacy provider fails to load. Instead of raising an exception, a warning is now emitted. The `CRYPTOGRAPHY_OPENSSL_NO_LEGACY` environment variable can still be used to disable the legacy provider at runtime.
- Added support for the `CRYPTOGRAPHY_BUILD_OPENSSL_NO_LEGACY` environment variable during build time, which prevents the library from ever attempting to load the legacy provider.
- Added support for the `PrivateKeyUsagePeriod` X.509 extension. This extension defines the period during which the private key corresponding to the certificate's public key may be used.
- Added support for compiling against aws-lc.
- Parsing X.509 structures now more strictly enforces that `Name` structures do not have malformed ASN.1.
- We now publish `py311` wheels that utilize the faster `pyo3::buffer::PyBuffer` interface, resulting in significantly improved performance for operations involving small buffers.
- Added `ssh_key_fingerprint()` for computing fingerprints of SSH public keys.
- Added support for deterministic ECDSA signing via the new keyword-only argument `ecdsa_deterministic` in `sign()`, `sign()` and `sign()`.

2.10.24 44.0.3 - 2025-05-02

- Fixed compilation when using LibreSSL 4.1.0.

2.10.25 44.0.2 - 2025-03-01

- We now build wheels for PyPy 3.11.

2.10.26 44.0.1 - 2025-02-11

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.4.1.
- We now build `armv7l` `manylinux` wheels and publish them to PyPI.
- We now build `manylinux_2_34` wheels and publish them to PyPI.

2.10.27 44.0.0 - 2024-11-27

- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 3.9.
- Deprecated Python 3.7 support. Python 3.7 is no longer supported by the Python core team. Support for Python 3.7 will be removed in a future `cryptography` release.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.4.0.
- macOS wheels are now built against the macOS 10.13 SDK. Users on older versions of macOS should upgrade, or they will need to build `cryptography` themselves.
- Enforce the **RFC 5280** requirement that extended key usage extensions must not be empty.
- Added support for timestamp extraction to the `MultiFernet` class.
- Relax the Authority Key Identifier requirements on root CA certificates during X.509 verification to allow fields permitted by **RFC 5280** but forbidden by the CA/Browser BRs.
- Added support for `Argon2id` when using OpenSSL 3.2.0+.
- Added support for the `Admissions` certificate extension.

- Added basic support for PKCS7 decryption (including S/MIME 3.2) via *pkcs7_decrypt_der()*, *pkcs7_decrypt_pem()*, and *pkcs7_decrypt_smime()*.

2.10.28 43.0.3 - 2024-10-18

- Fixed release metadata for `cryptography-vectors`

2.10.29 43.0.2 - 2024-10-18

- Fixed compilation when using LibreSSL 4.0.0.

2.10.30 43.0.1 - 2024-09-03

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.3.2.

2.10.31 43.0.0 - 2024-07-20

- **BACKWARDS INCOMPATIBLE:** Support for OpenSSL less than 1.1.1e has been removed. Users on older version of OpenSSL will need to upgrade.
- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 3.8.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.3.1.
- Updated the minimum supported Rust version (MSRV) to 1.65.0, from 1.63.0.
- *generate_private_key()* now enforces a minimum RSA key size of 1024-bit. Note that 1024-bit is still considered insecure, users should generally use a key size of 2048-bits.
- *serialize_certificates()* now emits ASN.1 that more closely follows the recommendations in **RFC 2315**.
- Added new *Decrepit cryptography* module which contains outdated and insecure cryptographic primitives. `CAST5`, `SEED`, `IDEA`, and `Blowfish`, which were deprecated in 37.0.0, have been added to this module. They will be removed from the `cipher` module in 45.0.0.
- Moved *TripleDES* and *ARC4* into *Decrepit cryptography* and deprecated them in the `cipher` module. They will be removed from the `cipher` module in 48.0.0.
- Added support for deterministic *ECDSA* (**RFC 6979**)
- Added support for client certificate verification to the *X.509 path validation* APIs in the form of *ClientVerifier*, *VerifiedClient*, and `PolicyBuilder` *build_client_verifier()*.
- Added Certificate *public_key_algorithm_oid* and Certificate Signing Request *public_key_algorithm_oid* to determine the *PublicKeyAlgorithmOID* Object Identifier of the public key found inside the certificate.
- Added *invalidity_date_utc*, a timezone-aware alternative to the naïve `datetime` attribute *invalidity_date*.
- Added support for parsing empty DN string in *from_rfc4514_string()*.
- Added the following properties that return timezone-aware `datetime` objects: *produced_at_utc()*, *revocation_time_utc()*, *this_update_utc()*, *next_update_utc()*, *revocation_time_utc()*, *this_update_utc()*, *next_update_utc()*, These are timezone-aware variants of existing properties that return naïve `datetime` objects.
- Added *rsa_recover_private_exponent()*
- Added *reset_nonce()* for altering the `nonce` of a cipher context without initializing a new instance. See the docs for additional restrictions.

- *NameAttribute* now raises an exception when attempting to create a common name whose length is shorter or longer than **RFC 5280** permits.
- Added basic support for PKCS7 encryption (including SMIME) via *PKCS7EnvelopeBuilder*.

2.10.32 42.0.8 - 2024-06-04

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.2.2.

2.10.33 42.0.7 - 2024-05-06

- Restored Windows 7 compatibility for our pre-built wheels. Note that we do not test on Windows 7 and wheels for our next release will not support it. Microsoft no longer provides support for Windows 7 and users are encouraged to upgrade.

2.10.34 42.0.6 - 2024-05-04

- Fixed compilation when using LibreSSL 3.9.1.

2.10.35 42.0.5 - 2024-02-23

- Limit the number of name constraint checks that will be performed in *X.509 path validation* to protect against denial of service attacks.
- Upgrade `pyo3` version, which fixes building on PowerPC.

2.10.36 42.0.4 - 2024-02-20

- Fixed a null-pointer-dereference and segfault that could occur when creating a PKCS#12 bundle. Credit to **Alexander-Programming** for reporting the issue. **CVE-2024-26130**
- Fixed ASN.1 encoding for PKCS7/SMIME signed messages. The fields `SMIMECapabilities` and `SignatureAlgorithmIdentifier` should now be correctly encoded according to the definitions in **RFC 2633** **RFC 3370**.

2.10.37 42.0.3 - 2024-02-15

- Fixed an initialization issue that caused key loading failures for some users.

2.10.38 42.0.2 - 2024-01-30

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.2.1.
- Fixed an issue that prevented the use of Python buffer protocol objects in `sign` and `verify` methods on asymmetric keys.
- Fixed an issue with incorrect keyword-argument naming with `EllipticCurvePrivateKey` *exchange()*, `X25519PrivateKey` *exchange()*, `X448PrivateKey` *exchange()*, and `DHPrivateKey` *exchange()*.

2.10.39 42.0.1 - 2024-01-24

- Fixed an issue with incorrect keyword-argument naming with `EllipticCurvePrivateKey` *sign()*.
- Resolved compatibility issue with loading certain RSA public keys in *load_pem_public_key()*.

2.10.40 42.0.0 - 2024-01-22

- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 3.7.
- **BACKWARDS INCOMPATIBLE:** Loading a PKCS7 with no content field using *load_pem_pkcs7_certificates()* or *load_der_pkcs7_certificates()* will now raise a `ValueError` rather than return an empty list.
- Parsing SSH certificates no longer permits malformed critical options with values, as documented in the 41.0.2 release notes.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.2.0.
- Updated the minimum supported Rust version (MSRV) to 1.63.0, from 1.56.0.
- We now publish both `py37` and `py39 abi3` wheels. This should resolve some errors relating to initializing a module multiple times per process.
- Support *PSS* for X.509 certificate signing requests and certificate revocation lists with the keyword-only argument `rsa_padding` on the `sign` methods for *CertificateSigningRequestBuilder* and *CertificateRevocationListBuilder*.
- Added support for obtaining X.509 certificate signing request signature algorithm parameters (including PSS) via *signature_algorithm_parameters()*.
- Added support for obtaining X.509 certificate revocation list signature algorithm parameters (including PSS) via *signature_algorithm_parameters()*.
- Added `mgf` property to *PSS*.
- Added `algorithm` and `mgf` properties to *OAEP*.
- Added the following properties that return timezone-aware `datetime` objects: *not_valid_before_utc()*, *not_valid_after_utc()*, *revocation_date_utc()*, *next_update_utc()*, *last_update_utc()*. These are timezone-aware variants of existing properties that return naïve `datetime` objects.
- Deprecated the following properties that return naïve `datetime` objects: *not_valid_before()*, *not_valid_after()*, *revocation_date()*, *next_update()*, *last_update()* in favor of the new timezone-aware variants mentioned above.
- Added support for *ChaCha20* on LibreSSL.
- Added support for RSA PSS signatures in PKCS7 with *add_signer()*.
- In the next release (43.0.0) of cryptography, loading an X.509 certificate with a negative serial number will raise an exception. This has been deprecated since 36.0.0.
- Added support for *AESGCMSIV* when using OpenSSL 3.2.0+.
- Added the *X.509 path validation* APIs for *Certificate* chains. These APIs should be considered unstable and not subject to our stability guarantees until documented as such in a future release.
- Added support for *SM4 GCM* when using OpenSSL 3.0 or greater.

2.10.41 41.0.7 - 2023-11-27

- Fixed compilation when using LibreSSL 3.8.2.

2.10.42 41.0.6 - 2023-11-27

- Fixed a null-pointer-dereference and segfault that could occur when loading certificates from a PKCS#7 bundle. Credit to **pkuzco** for reporting the issue. **CVE-2023-49083**

2.10.43 41.0.5 - 2023-10-24

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.1.4.
- Added a function to support an upcoming `pyOpenSSL` release.

2.10.44 41.0.4 - 2023-09-19

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.1.3.

2.10.45 41.0.3 - 2023-08-01

- Fixed performance regression loading DH public keys.
- Fixed a memory leak when using *ChaCha20Poly1305*.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.1.2.

2.10.46 41.0.2 - 2023-07-10

- Fixed bugs in creating and parsing SSH certificates where critical options with values were handled incorrectly. Certificates are now created correctly and parsing accepts correct values as well as the previously generated invalid forms with a warning. In the next release, support for parsing these invalid forms will be removed.

2.10.47 41.0.1 - 2023-06-01

- Temporarily allow invalid ECDSA signature algorithm parameters in X.509 certificates, which are generated by older versions of Java.
- Allow null bytes in pass phrases when serializing private keys.

2.10.48 41.0.0 - 2023-05-30

- **BACKWARDS INCOMPATIBLE:** Support for OpenSSL less than 1.1.1d has been removed. Users on older version of OpenSSL will need to upgrade.
- **BACKWARDS INCOMPATIBLE:** Support for Python 3.6 has been removed.
- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 3.6.
- Updated the minimum supported Rust version (MSRV) to 1.56.0, from 1.48.0.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.1.1.
- Added support for the *OCSPAcceptableResponses* OCSP extension.
- Added support for the *MSCertificateTemplate* proprietary Microsoft certificate extension.
- Implemented support for equality checks on all asymmetric public key types.
- Added support for `aes256-gcm@openssh.com` encrypted keys in *load_ssh_private_key()*.
- Added support for obtaining X.509 certificate signature algorithm parameters (including PSS) via *signature_algorithm_parameters()*.
- Support signing *PSS* X.509 certificates via the new keyword-only argument `rsa_padding` on *sign()*.
- Added support for *ChaCha20Poly1305* on BoringSSL.

2.10.49 40.0.2 - 2023-04-14

- Fixed compilation when using LibreSSL 3.7.2.
- Added some functions to support an upcoming `pyOpenSSL` release.

2.10.50 40.0.1 - 2023-03-24

- Fixed a bug where certain operations would fail if an object happened to be in the top-half of the memory-space. This only impacted 32-bit systems.

2.10.51 40.0.0 - 2023-03-24

- **BACKWARDS INCOMPATIBLE:** As announced in the 39.0.0 changelog, the way `cryptography` links OpenSSL has changed. This only impacts users who build `cryptography` from source (i.e., not from a `wheel`), and specify their own version of OpenSSL. For those users, the `CFLAGS`, `LDFLAGS`, `INCLUDE`, `LIB`, and `CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS` environment variables are no longer valid. Instead, users need to configure their builds as documented here.
- Support for Python 3.6 is deprecated and will be removed in the next release.
- Deprecated the current minimum supported Rust version (MSRV) of 1.48.0. In the next release we will raise MSRV to 1.56.0. Users with the latest `pip` will typically get a wheel and not need Rust installed, but check *Installation* for documentation on installing a newer `rustc` if required.
- Deprecated support for OpenSSL less than 1.1.1d. The next release of `cryptography` will drop support for older versions.
- Deprecated support for DSA keys in *load_ssh_public_key()* and *load_ssh_private_key()*.
- Deprecated support for OpenSSH serialization in *DSAPublicKey* and *DSAPrivateKey*.
- The minimum supported version of PyPy3 is now 7.3.10.
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.1.0.
- Added support for parsing SSH certificates in addition to public keys with *load_ssh_public_identity()*. *load_ssh_public_key()* continues to support only public keys.
- Added support for generating SSH certificates with *SSHCertificateBuilder*.
- Added *verify_directly_issued_by()* to *Certificate*.
- Added a check to *NameConstraints* to ensure that *DNSName* constraints do not contain any * wildcards.
- Removed many unused CFFI OpenSSL bindings. This will not impact you unless you are using `cryptography` to directly invoke OpenSSL's C API. Note that these have never been considered a stable, supported, public API by `cryptography`, this note is included as a courtesy.
- The X.509 builder classes now raise `UnsupportedAlgorithm` instead of `ValueError` if an unsupported hash algorithm is passed.
- Added public union type aliases for type hinting:
  - Asymmetric types: *PublicKeyTypes*, *PrivateKeyTypes*, *CertificatePublicKeyTypes*, *CertificateIssuerPublicKeyTypes*, *CertificateIssuerPrivateKeyTypes*.
  - SSH keys: *SSHPublicKeyTypes*, *SSHPrivateKeyTypes*, *SSHCertPublicKeyTypes*, *SSHCertPrivateKeyTypes*.
  - PKCS12: *PKCS12PrivateKeyTypes*
  - PKCS7: *PKCS7HashTypes*, *PKCS7PrivateKeyTypes*.
  - Two-factor: *HOTPHashTypes*

- Deprecated previously undocumented but not private type aliases in the `cryptography.hazmat.primitives.asymmetric.types` module in favor of new ones above.

2.10.52 39.0.2 - 2023-03-02

- Fixed a bug where the content type header was not properly encoded for PKCS7 signatures when using the `Text` option and `SMIME` encoding.

2.10.53 39.0.1 - 2023-02-07

- **SECURITY ISSUE** - Fixed a bug where `Cipher.update_into` accepted Python buffer protocol objects, but allowed immutable buffers. **CVE-2023-23931**
- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.8.

2.10.54 39.0.0 - 2023-01-01

- **BACKWARDS INCOMPATIBLE:** Support for OpenSSL 1.1.0 has been removed. Users on older version of OpenSSL will need to upgrade.
- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL < 3.5. The new minimum LibreSSL version is 3.5.0. Going forward our policy is to support versions of LibreSSL that are available in versions of OpenBSD that are still receiving security support.
- **BACKWARDS INCOMPATIBLE:** Removed the `encode_point` and `from_encoded_point` methods on *`EllipticCurvePublicNumbers`*, which had been deprecated for several years. *`public_bytes()`* and *`from_encoded_point()`* should be used instead.
- **BACKWARDS INCOMPATIBLE:** Support for using MD5 or SHA1 in *`CertificateBuilder`*, other X.509 builders, and PKCS7 has been removed.
- **BACKWARDS INCOMPATIBLE:** Dropped support for macOS 10.10 and 10.11, macOS users must upgrade to 10.12 or newer.
- **ANNOUNCEMENT:** The next version of `cryptography` (40.0) will change the way we link OpenSSL. This will only impact users who build `cryptography` from source (i.e., not from a `wheel`), and specify their own version of OpenSSL. For those users, the `CFLAGS`, `LDFLAGS`, `INCLUDE`, `LIB`, and `CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS` environment variables will no longer be respected. Instead, users will need to configure their builds as documented here.
- Added support for *disabling the legacy provider in OpenSSL 3.0.x*.
- Added support for disabling RSA key validation checks when loading RSA keys via *`load_pem_private_key()`*, *`load_der_private_key()`*, and *`private_key()`*. This speeds up key loading but is *unsafe* if you are loading potentially attacker supplied keys.
- Significantly improved performance for *`ChaCha20Poly1305`* when repeatedly calling `encrypt` or `decrypt` with the same key.
- Added support for creating OCSP requests with precomputed hashes using *`add_certificate_by_hash()`*.
- Added support for loading multiple PEM-encoded X.509 certificates from a single input via *`load_pem_x509_certificates()`*.

2.10.55 38.0.4 - 2022-11-27

- Fixed compilation when using LibreSSL 3.6.0.
- Fixed error when using `py2app` to build an application with a `cryptography` dependency.

2.10.56 38.0.3 - 2022-11-01

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.7, which resolves *CVE-2022-3602* and *CVE-2022-3786*.

2.10.57 38.0.2 - 2022-10-11 (YANKED)

> **Attention**
>
> This release was subsequently yanked from PyPI due to a regression in OpenSSL.

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.6.

2.10.58 38.0.1 - 2022-09-07

- Fixed parsing TLVs in ASN.1 with length greater than 65535 bytes (typically seen in large CRLs).

2.10.59 38.0.0 - 2022-09-06

- Final deprecation of OpenSSL 1.1.0. The next release of `cryptography` will drop support.
- We no longer ship `manylinux2010` wheels. Users should upgrade to the latest `pip` to ensure this doesn't cause issues downloading wheels on their platform. We now ship `manylinux_2_28` wheels for users on new enough platforms.
- Updated the minimum supported Rust version (MSRV) to 1.48.0, from 1.41.0. Users with the latest `pip` will typically get a wheel and not need Rust installed, but check *Installation* for documentation on installing a newer `rustc` if required.
- *`decrypt()`* and related methods now accept both `str` and `bytes` tokens.
- Parsing `CertificateSigningRequest` restores the behavior of enforcing that the `Extension` `critical` field must be correctly encoded DER. See the issue for complete details.
- Added two new OpenSSL functions to the bindings to support an upcoming `pyOpenSSL` release.
- When parsing *`CertificateRevocationList`* and *`CertificateSigningRequest`* values, it is now enforced that the `version` value in the input must be valid according to the rules of **RFC 2986** and **RFC 5280**.
- Using MD5 or SHA1 in *`CertificateBuilder`* and other X.509 builders is deprecated and support will be removed in the next version.
- Added additional APIs to *`SignedCertificateTimestamp`*, including *`signature_hash_algorithm`*, *`signature_algorithm`*, *`signature`*, and *`extension_bytes`*.
- Added *`tbs_precertificate_bytes`*, allowing users to access the to-be-signed pre-certificate data needed for signed certificate timestamp verification.
- *`KBKDFHMAC`* and *`KBKDFCMAC`* now support *`MiddleFixed`* counter location.
- Fixed **RFC 4514** name parsing to reverse the order of the RDNs according to the section 2.1 of the RFC, affecting method *`from_rfc4514_string()`*.
- It is now possible to customize some aspects of encryption when serializing private keys, using *`encryption_builder()`*.
- Removed several legacy symbols from our OpenSSL bindings. Users of pyOpenSSL versions older than 22.0 will need to upgrade.

- Added *AES128* and *AES256* classes. These classes do not replace *AES* (which allows all AES key lengths), but are intended for applications where developers want to be explicit about key length.

2.10.60 37.0.4 - 2022-07-05

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.5.

2.10.61 37.0.3 - 2022-06-21 (YANKED)

> **Attention**
>
> This release was subsequently yanked from PyPI due to a regression in OpenSSL.

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.4.

2.10.62 37.0.2 - 2022-05-03

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.3.
- Added a constant needed for an upcoming pyOpenSSL release.

2.10.63 37.0.1 - 2022-04-27

- Fixed an issue where parsing an encrypted private key with the public loader functions would hang waiting for console input on OpenSSL 3.0.x rather than raising an error.
- Restored some legacy symbols for older `pyOpenSSL` users. These will be removed again in the future, so `pyOpenSSL` users should still upgrade to the latest version of that package when they upgrade `cryptography`.

2.10.64 37.0.0 - 2022-04-26

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 3.0.2.
- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL 2.9.x and 3.0.x. The new minimum LibreSSL version is 3.1+.
- **BACKWARDS INCOMPATIBLE:** Removed `signer` and `verifier` methods from the public key and private key classes. These methods were originally deprecated in version 2.0, but had an extended deprecation timeline due to usage. Any remaining users should transition to `sign` and `verify`.
- Deprecated OpenSSL 1.1.0 support. OpenSSL 1.1.0 is no longer supported by the OpenSSL project. The next release of `cryptography` will be the last to support compiling with OpenSSL 1.1.0.
- Deprecated Python 3.6 support. Python 3.6 is no longer supported by the Python core team. Support for Python 3.6 will be removed in a future `cryptography` release.
- Deprecated the current minimum supported Rust version (MSRV) of 1.41.0. In the next release we will raise MSRV to 1.48.0. Users with the latest `pip` will typically get a wheel and not need Rust installed, but check *Installation* for documentation on installing a newer `rustc` if required.
- Deprecated `CAST5`, `SEED`, `IDEA`, and `Blowfish` because they are legacy algorithms with extremely low usage. These will be removed in a future version of `cryptography`.
- Added limited support for distinguished names containing a bit string.
- We now ship `universal2` wheels on macOS, which contain both `arm64` and `x86_64` architectures. Users on macOS should upgrade to the latest `pip` to ensure they can use this wheel, although we will continue to ship `x86_64` specific wheels for now to ease the transition.

- This will be the final release for which we ship `manylinux2010` wheels. Going forward the minimum supported `manylinux` ABI for our wheels will be `manylinux2014`. The vast majority of users will continue to receive `manylinux` wheels provided they have an up to date `pip`. For PyPy wheels this release already requires `manylinux2014` for compatibility with binaries distributed by upstream.
- Added support for multiple *OCSPSingleResponse* in a *OCSPResponse*.
- Restored support for signing certificates and other structures in *X.509* with SHA3 hash algorithms.
- *TripleDES* is disabled in FIPS mode.
- Added support for serialization of PKCS#12 CA friendly names/aliases in *serialize_key_and_certificates()*
- Added support for 12-15 byte (96 to 120 bit) nonces to *AESOCB3*. This class previously supported only 12 byte (96 bit).
- Added support for *AESSIV* when using OpenSSL 3.0.0+.
- Added support for serializing PKCS7 structures from a list of certificates with *serialize_certificates*.
- Added support for parsing **RFC 4514** strings with *from_rfc4514_string()*.
- Added *AUTO* to *PSS*. This can be used to verify a signature where the salt length is not already known.
- Added *DIGEST_LENGTH* to *PSS*. This constant will set the salt length to the same length as the PSS hash algorithm.
- Added support for loading RSA-PSS key types with *load_pem_private_key()* and *load_der_private_key()*. This functionality is limited to OpenSSL 1.1.1e+ and loads the key as a normal RSA private key, discarding the PSS constraint information.

2.10.65 36.0.2 - 2022-03-15

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 1.1.1n.

2.10.66 36.0.1 - 2021-12-14

- Updated Windows, macOS, and Linux wheels to be compiled with OpenSSL 1.1.1m.

2.10.67 36.0.0 - 2021-11-21

- **FINAL DEPRECATION** Support for `verifier` and `signer` on our asymmetric key classes was deprecated in version 2.0. These functions had an extended deprecation due to usage, however the next version of `cryptography` will drop support. Users should migrate to `sign` and `verify`.
- The entire *X.509* layer is now written in Rust. This allows alternate asymmetric key implementations that can support cloud key management services or hardware security modules provided they implement the necessary interface (for example: *EllipticCurvePrivateKey*).
- *Deprecated the backend argument* for all functions.
- Added support for *AESOCB3*.
- Added support for iterating over arbitrary request *attributes*.
- Deprecated the `get_attribute_for_oid` method on *CertificateSigningRequest* in favor of *get_attribute_for_oid()* on the new *Attributes* object.
- Fixed handling of PEM files to allow loading when certificate and key are in the same file.
- Fixed parsing of *CertificatePolicies* extensions containing legacy `BMPString` values in their `explicitText`.

- Allow parsing of negative serial numbers in certificates. Negative serial numbers are prohibited by **RFC 5280** so a deprecation warning will be raised whenever they are encountered. A future version of `cryptography` will drop support for parsing them.
- Added support for parsing PKCS12 files with friendly names for all certificates with *load_pkcs12()*, which will return an object of type *PKCS12KeyAndCertificates*.
- *rfc4514_string()* and related methods now have an optional `attr_name_overrides` parameter to supply custom OID to name mappings, which can be used to match vendor-specific extensions.
- **BACKWARDS INCOMPATIBLE:** Reverted the nonstandard formatting of email address fields as `E` in *rfc4514_string()* methods from version 35.0.

  The previous behavior can be restored with: `name.rfc4514_string({NameOID.EMAIL_ADDRESS: "E"})`
- Allow *X25519PublicKey* and *X448PublicKey* to be used as public keys when parsing certificates or creating them with *CertificateBuilder*. These key types must be signed with a different signing algorithm as `X25519` and `X448` do not support signing.
- Extension values can now be serialized to a DER byte string by calling *public_bytes()*.
- Added experimental support for compiling against BoringSSL. As BoringSSL does not commit to a stable API, `cryptography` tests against the latest commit only. Please note that several features are not available when building against BoringSSL.
- Parsing `CertificateSigningRequest` from DER and PEM now, for a limited time period, allows the `Extension critical` field to be incorrectly encoded. See the issue for complete details. This will be reverted in a future `cryptography` release.
- When *OCSPNonce* are parsed and generated their value is now correctly wrapped in an ASN.1 `OCTET STRING`. This conforms to **RFC 6960** but conflicts with the original behavior specified in **RFC 2560**. For a temporary period for backwards compatibility, we will also parse values that are encoded as specified in **RFC 2560** but this behavior will be removed in a future release.

2.10.68 35.0.0 - 2021-09-29

- Changed the *version scheme*. This will result in us incrementing the major version more frequently, but does not change our existing backwards compatibility policy.
- **BACKWARDS INCOMPATIBLE:** The *X.509* PEM parsers now require that the PEM string passed have PEM delimiters of the correct type. For example, parsing a private key PEM concatenated with a certificate PEM will no longer be accepted by the PEM certificate parser.
- **BACKWARDS INCOMPATIBLE:** The X.509 certificate parser no longer allows negative serial numbers. **RFC 5280** has always prohibited these.
- **BACKWARDS INCOMPATIBLE:** Additional forms of invalid ASN.1 found during *X.509* parsing will raise an error on initial parse rather than when the malformed field is accessed.
- Rust is now required for building `cryptography`, the `CRYPTOGRAPHY_DONT_BUILD_RUST` environment variable is no longer respected.
- Parsers for *X.509* no longer use OpenSSL and have been rewritten in Rust. This should be backwards compatible (modulo the items listed above) and improve both security and performance.
- Added support for OpenSSL 3.0.0 as a compilation target.
- Added support for *SM3* and *SM4*, when using OpenSSL 1.1.1. These algorithms are provided for compatibility in regions where they may be required, and are not generally recommended.
- We now ship `manylinux_2_24` and `musllinux_1_1` wheels, in addition to our `manylinux2010` and `manylinux2014` wheels. Users on distributions like Alpine Linux should ensure they upgrade to the latest `pip` to correctly receive wheels.

- Added `rfc4514_attribute_name` attribute to *x509.NameAttribute*.
- Added *KBKDFCMAC*.

2.10.69 3.4.8 - 2021-08-24

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1l.

2.10.70 3.4.7 - 2021-03-25

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1k.

2.10.71 3.4.6 - 2021-02-16

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1j.

2.10.72 3.4.5 - 2021-02-13

- Various improvements to type hints.
- Lower the minimum supported Rust version (MSRV) to >=1.41.0. This change improves compatibility with system-provided Rust on several Linux distributions.
- `cryptography` will be switching to a new versioning scheme with its next feature release. More information is available in our *API stability* documentation.

2.10.73 3.4.4 - 2021-02-09

- Added a `py.typed` file so that `mypy` will know to use our type annotations.
- Fixed an import cycle that could be triggered by certain import sequences.

2.10.74 3.4.3 - 2021-02-08

- Specify our supported Rust version (>=1.45.0) in our `setup.py` so users on older versions will get a clear error message.

2.10.75 3.4.2 - 2021-02-08

- Improvements to make the rust transition a bit easier. This includes some better error messages and small dependency fixes. If you experience installation problems **Be sure to update pip** first, then check the *FAQ*.

2.10.76 3.4.1 - 2021-02-07

- Fixed a circular import issue.
- Added additional debug output to assist users seeing installation errors due to outdated `pip` or missing `rustc`.

2.10.77 3.4 - 2021-02-07

- **BACKWARDS INCOMPATIBLE:** Support for Python 2 has been removed.
- We now ship `manylinux2014` wheels and no longer ship `manylinux1` wheels. Users should upgrade to the latest `pip` to ensure this doesn't cause issues downloading wheels on their platform.
- `cryptography` now incorporates Rust code. Users building `cryptography` themselves will need to have the Rust toolchain installed. Users who use an officially produced wheel will not need to make any changes. The minimum supported Rust version is 1.45.0.

- `cryptography` now has **PEP 484** type hints on nearly all of of its public APIs. Users can begin using them to type check their code with `mypy`.

2.10.78 3.3.2 - 2021-02-07

- **SECURITY ISSUE:** Fixed a bug where certain sequences of `update()` calls when symmetrically encrypting very large payloads (>2GB) could result in an integer overflow, leading to buffer overflows. *CVE-2020-36242* **Update:** This fix is a workaround for *CVE-2021-23840* in OpenSSL, fixed in OpenSSL 1.1.1j.

2.10.79 3.3.1 - 2020-12-09

- Re-added a legacy symbol causing problems for older `pyOpenSSL` users.

2.10.80 3.3 - 2020-12-08

- **BACKWARDS INCOMPATIBLE:** Support for Python 3.5 has been removed due to low usage and maintenance burden.
- **BACKWARDS INCOMPATIBLE:** The *GCM* and *AESGCM* now require 64-bit to 1024-bit (8 byte to 128 byte) initialization vectors. This change is to conform with an upcoming OpenSSL release that will no longer support sizes outside this window.
- **BACKWARDS INCOMPATIBLE:** When deserializing asymmetric keys we now raise `ValueError` rather than `UnsupportedAlgorithm` when an unsupported cipher is used. This change is to conform with an upcoming OpenSSL release that will no longer distinguish between error types.
- **BACKWARDS INCOMPATIBLE:** We no longer allow loading of finite field Diffie-Hellman parameters of less than 512 bits in length. This change is to conform with an upcoming OpenSSL release that no longer supports smaller sizes. These keys were already wildly insecure and should not have been used in any application outside of testing.
- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1i.
- Python 2 support is deprecated in `cryptography`. This is the last release that will support Python 2.
- Added the *recover_data_from_signature()* function to *RSAPublicKey* for recovering the signed data from an RSA signature.

2.10.81 3.2.1 - 2020-10-27

- Disable blinding on RSA public keys to address an error with some versions of OpenSSL.

2.10.82 3.2 - 2020-10-25

- **SECURITY ISSUE:** Attempted to make RSA PKCS#1v1.5 decryption more constant time, to protect against Bleichenbacher vulnerabilities. Due to limitations imposed by our API, we cannot completely mitigate this vulnerability and a future release will contain a new API which is designed to be resilient to these for contexts where it is required. Credit to **Hubert Kario** for reporting the issue. *CVE-2020-25659*
- Support for OpenSSL 1.0.2 has been removed. Users on older version of OpenSSL will need to upgrade.
- Added basic support for PKCS7 signing (including SMIME) via *PKCS7SignatureBuilder*.

2.10.83 3.1.1 - 2020-09-22

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1h.

2.10.84 3.1 - 2020-08-26

- **BACKWARDS INCOMPATIBLE:** Removed support for `idna` based *U-label* parsing in various X.509 classes. This support was originally deprecated in version 2.1 and moved to an extra in 2.5.
- Deprecated OpenSSL 1.0.2 support. OpenSSL 1.0.2 is no longer supported by the OpenSSL project. The next version of `cryptography` will drop support for it.
- Deprecated support for Python 3.5. This version sees very little use and will be removed in the next release.
- `backend` arguments to functions are no longer required and the default backend will automatically be selected if no `backend` is provided.
- Added initial support for parsing certificates from PKCS7 files with *load_pem_pkcs7_certificates()* and *load_der_pkcs7_certificates()* .
- Calling `update` or `update_into` on *CipherContext* with `data` longer than 2^{31} bytes no longer raises an `OverflowError`. This also resolves the same issue in *Fernet (symmetric encryption)*.

2.10.85 3.0 - 2020-07-20

- **BACKWARDS INCOMPATIBLE:** Removed support for passing an *Extension* instance to *from_issuer_subject_key_identifier()*, as per our deprecation policy.
- **BACKWARDS INCOMPATIBLE:** Support for LibreSSL 2.7.x, 2.8.x, and 2.9.0 has been removed (2.9.1+ is still supported).
- **BACKWARDS INCOMPATIBLE:** Dropped support for macOS 10.9, macOS users must upgrade to 10.10 or newer.
- **BACKWARDS INCOMPATIBLE:** RSA *generate_private_key()* no longer accepts `public_exponent` values except 65537 and 3 (the latter for legacy purposes).
- **BACKWARDS INCOMPATIBLE:** X.509 certificate parsing now enforces that the `version` field contains a valid value, rather than deferring this check until *version* is accessed.
- Deprecated support for Python 2. At the time there is no time table for actually dropping support, however we strongly encourage all users to upgrade their Python, as Python 2 no longer receives support from the Python core team.

  If you have trouble suppressing this warning in tests view the *FAQ entry addressing this issue*.
- Added support for `OpenSSH` serialization format for `ec`, `ed25519`, `rsa` and `dsa` private keys: *load_ssh_private_key()* for loading and *OpenSSH* for writing.
- Added support for `OpenSSH` certificates to *load_ssh_public_key()*.
- Added *encrypt_at_time()* and *decrypt_at_time()* to *Fernet*.
- Added support for the *SubjectInformationAccess* X.509 extension.
- Added support for parsing *SignedCertificateTimestamps* in OCSP responses.
- Added support for parsing attributes in certificate signing requests via `CertificateSigningRequest.get_attribute_for_oid`.
- Added support for encoding attributes in certificate signing requests via *add_attribute()*.
- On OpenSSL 1.1.1d and higher `cryptography` now uses OpenSSL's built-in CSPRNG instead of its own OS random engine because these versions of OpenSSL properly reseed on fork.
- Added initial support for creating PKCS12 files with *serialize_key_and_certificates()*.

2.10.86 2.9.2 - 2020-04-22

- Updated the macOS wheel to fix an issue where it would not run on macOS versions older than 10.15.

2.10.87 2.9.1 - 2020-04-21

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1g.

2.10.88 2.9 - 2020-04-02

- **BACKWARDS INCOMPATIBLE:** Support for Python 3.4 has been removed due to low usage and maintenance burden.
- **BACKWARDS INCOMPATIBLE:** Support for OpenSSL 1.0.1 has been removed. Users on older version of OpenSSL will need to upgrade.
- **BACKWARDS INCOMPATIBLE:** Support for LibreSSL 2.6.x has been removed.
- Removed support for calling *public_bytes()* with no arguments, as per our deprecation policy. You must now pass `encoding` and `format`.
- **BACKWARDS INCOMPATIBLE:** Reversed the order in which *rfc4514_string()* returns the RDNs as required by **RFC 4514**.
- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1f.
- Added support for parsing *single_extensions* in an OCSP response.
- *NameAttribute* values can now be empty strings.

2.10.89 2.8 - 2019-10-16

- Updated Windows, macOS, and `manylinux` wheels to be compiled with OpenSSL 1.1.1d.
- Added support for Python 3.8.
- Added class methods *Poly1305.generate_tag* and *Poly1305.verify_tag* for Poly1305 sign and verify operations.
- Deprecated support for OpenSSL 1.0.1. Support will be removed in `cryptography` 2.9.
- We now ship `manylinux2010` wheels in addition to our `manylinux1` wheels.
- Added support for `ed25519` and `ed448` keys in the *CertificateBuilder*, *CertificateSigningRequestBuilder*, *CertificateRevocationListBuilder* and *OCSPResponseBuilder*.
- `cryptography` no longer depends on `asn1crypto`.
- *FreshestCRL* is now allowed as a *CertificateRevocationList* extension.

2.10.90 2.7 - 2019-05-30

- **BACKWARDS INCOMPATIBLE:** We no longer distribute 32-bit `manylinux1` wheels. Continuing to produce them was a maintenance burden.
- **BACKWARDS INCOMPATIBLE:** Removed the `cryptography.hazmat.primitives.mac.MACContext` interface. The `CMAC` and `HMAC` APIs have not changed, but they are no longer registered as `MACContext` instances.
- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.1c.
- Removed support for running our tests with `setup.py test`. Users interested in running our tests can continue to follow the directions in our *development documentation*.

- Add support for *Poly1305* when using OpenSSL 1.1.1 or newer.
- Support serialization with `Encoding.OpenSSH` and `PublicFormat.OpenSSH` in *Ed25519PublicKey.public_bytes* .
- Correctly allow passing a `SubjectKeyIdentifier` to *from_issuer_subject_key_identifier()* and deprecate passing an `Extension` object. The documentation always required `SubjectKeyIdentifier` but the implementation previously required an `Extension`.

2.10.91 2.6.1 - 2019-02-27

- Resolved an error in our build infrastructure that broke our Python3 wheels for macOS and Linux.

2.10.92 2.6 - 2019-02-27

- **BACKWARDS INCOMPATIBLE:** Removed `cryptography.hazmat.primitives.asymmetric.utils.encode_rfc6979_signature` and `cryptography.hazmat.primitives.asymmetric.utils.decode_rfc6979_signature`, which had been deprecated for nearly 4 years. Use *encode_dss_signature()* and *decode_dss_signature()* instead.
- **BACKWARDS INCOMPATIBLE**: Removed `cryptography.x509.Certificate.serial`, which had been deprecated for nearly 3 years. Use *serial_number* instead.
- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.1b.
- Added support for *Ed448 signing* when using OpenSSL 1.1.1b or newer.
- Added support for *Ed25519 signing* when using OpenSSL 1.1.1b or newer.
- *load_ssh_public_key()* can now load `ed25519` public keys.
- Add support for easily mapping an object identifier to its elliptic curve class via *get_curve_for_oid()*.
- Add support for OpenSSL when compiled with the `no-engine` (`OPENSSL_NO_ENGINE`) flag.

2.10.93 2.5 - 2019-01-22

- **BACKWARDS INCOMPATIBLE:** *U-label* strings were deprecated in version 2.1, but this version removes the default `idna` dependency as well. If you still need this deprecated path please install cryptography with the `idna` extra: `pip install cryptography[idna]`.
- **BACKWARDS INCOMPATIBLE:** The minimum supported PyPy version is now 5.4.
- Numerous classes and functions have been updated to allow *bytes-like* types for keying material and passwords, including symmetric algorithms, AEAD ciphers, KDFs, loading asymmetric keys, and one time password classes.
- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.1a.
- Added support for *SHA512_224* and *SHA512_256* when using OpenSSL 1.1.1.
- Added support for *SHA3_224*, *SHA3_256*, *SHA3_384*, and *SHA3_512* when using OpenSSL 1.1.1.
- Added support for *X448 key exchange* when using OpenSSL 1.1.1.
- Added support for *SHAKE128* and *SHAKE256* when using OpenSSL 1.1.1.
- Added initial support for parsing PKCS12 files with *load_key_and_certificates()*.
- Added support for *IssuingDistributionPoint*.
- Added `rfc4514_string()` method to *x509.Name*, *x509.RelativeDistinguishedName*, and *x509.NameAttribute* to format the name or component an **RFC 4514** Distinguished Name string.

- Added *from_encoded_point()*, which immediately checks if the point is on the curve and supports compressed points. Deprecated the previous method `cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers.from_encoded_point`.
- Added *signature_hash_algorithm* to `OCSPResponse`.
- Updated *X25519 key exchange* support to allow additional serialization methods. Calling *public_bytes()* with no arguments has been deprecated.
- Added support for encoding compressed and uncompressed points via *public_bytes()*. Deprecated the previous method `cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers.encode_point`.

2.10.94 2.4.2 - 2018-11-21

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0j.

2.10.95 2.4.1 - 2018-11-11

- Fixed a build breakage in our `manylinux1` wheels.

2.10.96 2.4 - 2018-11-11

- **BACKWARDS INCOMPATIBLE:** Dropped support for LibreSSL 2.4.x.
- Deprecated OpenSSL 1.0.1 support. OpenSSL 1.0.1 is no longer supported by the OpenSSL project. At this time there is no time table for dropping support, however we strongly encourage all users to upgrade or install `cryptography` from a wheel.
- Added initial *OCSP* support.
- Added support for *PrecertPoison*.

2.10.97 2.3.1 - 2018-08-14

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0i.

2.10.98 2.3 - 2018-07-18

- **SECURITY ISSUE:** *finalize_with_tag()* allowed tag truncation by default which can allow tag forgery in some cases. The method now enforces the `min_tag_length` provided to the *GCM* constructor. *CVE-2018-10903*
- Added support for Python 3.7.
- Added *extract_timestamp()* to get the authenticated timestamp of a *Fernet* token.
- Support for Python 2.7.x without `hmac.compare_digest` has been deprecated. We will require Python 2.7.7 or higher (or 2.7.6 on Ubuntu) in the next `cryptography` release.
- Fixed multiple issues preventing `cryptography` from compiling against LibreSSL 2.7.x.
- Added *get_revoked_certificate_by_serial_number* for quick serial number searches in CRLs.
- The *RelativeDistinguishedName* class now preserves the order of attributes. Duplicate attributes now raise an error instead of silently discarding duplicates.
- *aes_key_unwrap()* and *aes_key_unwrap_with_padding()* now raise *InvalidUnwrap* if the wrapped key is an invalid length, instead of `ValueError`.

2.10.99 2.2.2 - 2018-03-27

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0h.

2.10.100 2.2.1 - 2018-03-20

- Reverted a change to `GeneralNames` which prohibited having zero elements, due to breakages.
- Fixed a bug in *aes_key_unwrap_with_padding()* that caused it to raise `InvalidUnwrap` when key length modulo 8 was zero.

2.10.101 2.2 - 2018-03-19

- **BACKWARDS INCOMPATIBLE:** Support for Python 2.6 has been dropped.
- Resolved a bug in `HKDF` that incorrectly constrained output size.
- Added *BrainpoolP256R1*, *BrainpoolP384R1*, and *BrainpoolP512R1* to support inter-operating with systems like German smart meters.
- Added token rotation support to *Fernet* with *rotate()*.
- Fixed a memory leak in *derive_private_key()*.
- Added support for AES key wrapping with padding via *aes_key_wrap_with_padding()* and *aes_key_unwrap_with_padding()* .
- Allow loading DSA keys with 224 bit `q`.

2.10.102 2.1.4 - 2017-11-29

- Added `X509_up_ref` for an upcoming `pyOpenSSL` release.

2.10.103 2.1.3 - 2017-11-02

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0g.

2.10.104 2.1.2 - 2017-10-24

- Corrected a bug with the `manylinux1` wheels where OpenSSL's stack was marked executable.

2.10.105 2.1.1 - 2017-10-12

- Fixed support for install with the system `pip` on Ubuntu 16.04.

2.10.106 2.1 - 2017-10-11

- **FINAL DEPRECATION** Python 2.6 support is deprecated, and will be removed in the next release of `cryptography`.
- **BACKWARDS INCOMPATIBLE:** `Whirlpool`, `RIPEMD160`, and `UnsupportedExtension` have been removed in accordance with our *API stability* policy.
- **BACKWARDS INCOMPATIBLE:** *DNSName.value*, *RFC822Name.value*, and *UniformResourceIdentifier.value* will now return an *A-label* string when parsing a certificate containing an internationalized domain name (IDN) or if the caller passed a *U-label* to the constructor. See below for additional deprecations related to this change.
- Installing `cryptography` now requires `pip` 6 or newer.

- Deprecated passing *U-label* strings to the *DNSName*, *UniformResourceIdentifier*, and *RFC822Name* constructors. Instead, users should pass values as *A-label* strings with `idna` encoding if necessary. This change will not affect anyone who is not processing internationalized domains.
- Added support for *ChaCha20*. In most cases users should choose *ChaCha20Poly1305* rather than using this unauthenticated form.
- Added *is_signature_valid()* to *CertificateRevocationList*.
- Support *BLAKE2b* and *BLAKE2s* with *HMAC*.
- Added support for *XTS* mode for AES.
- Added support for using labels with *OAEP* when using OpenSSL 1.0.2 or greater.
- Improved compatibility with NSS when issuing certificates from an issuer that has a subject with non-`UTF8String` string types.
- Add support for the *DeltaCRLIndicator* extension.
- Add support for the *TLSFeature* extension. This is commonly used for enabling `OCSP Must-Staple` in certificates.
- Add support for the *FreshestCRL* extension.

2.10.107 2.0.3 - 2017-08-03

- Fixed an issue with weak linking symbols when compiling on macOS versions older than 10.12.

2.10.108 2.0.2 - 2017-07-27

- Marked all symbols as hidden in the `manylinux1` wheel to avoid a bug with symbol resolution in certain scenarios.

2.10.109 2.0.1 - 2017-07-26

- Fixed a compilation bug affecting OpenBSD.
- Altered the `manylinux1` wheels to statically link OpenSSL instead of dynamically linking and bundling the shared object. This should resolve crashes seen when using `uwsgi` or other binaries that link against OpenSSL independently.
- Fixed the stack level for the `signer` and `verifier` warnings.

2.10.110 2.0 - 2017-07-17

- **BACKWARDS INCOMPATIBLE:** Support for Python 3.3 has been dropped.
- We now ship `manylinux1` wheels linked against OpenSSL 1.1.0f. These wheels will be automatically used with most Linux distributions if you are running the latest pip.
- Deprecated the use of `signer` on *RSAPrivateKey*, *DSAPrivateKey*, and *EllipticCurvePrivateKey* in favor of `sign`.
- Deprecated the use of `verifier` on *RSAPublicKey*, *DSAPublicKey*, and *EllipticCurvePublicKey* in favor of `verify`.
- Added support for parsing *SignedCertificateTimestamp* objects from X.509 certificate extensions.
- Added support for *ChaCha20Poly1305*.
- Added support for *AESCCM*.

- Added *AESGCM*, a "one shot" API for AES GCM encryption.
- Added support for *X25519 key exchange*.
- Added support for serializing and deserializing Diffie-Hellman parameters with *load_pem_parameters()*, *load_der_parameters()*, and *parameter_bytes()* .
- The `extensions` attribute on *Certificate*, *CertificateSigningRequest*, *CertificateRevocationList*, and *RevokedCertificate* now caches the computed `Extensions` object. There should be no performance change, just a performance improvement for programs accessing the `extensions` attribute multiple times.

2.10.111 1.9 - 2017-05-29

- **BACKWARDS INCOMPATIBLE:** Elliptic Curve signature verification no longer returns `True` on success. This brings it in line with the interface's documentation, and our intent. The correct way to use *verify()* has always been to check whether or not *InvalidSignature* was raised.
- **BACKWARDS INCOMPATIBLE:** Dropped support for macOS 10.7 and 10.8.
- **BACKWARDS INCOMPATIBLE:** The minimum supported PyPy version is now 5.3.
- Python 3.3 support has been deprecated, and will be removed in the next `cryptography` release.
- Add support for providing `tag` during *GCM* finalization via *finalize_with_tag()*.
- Fixed an issue preventing `cryptography` from compiling against LibreSSL 2.5.x.
- Added *key_size()* and *key_size()* as convenience methods for determining the bit size of a secret scalar for the curve.
- Accessing an unrecognized extension marked critical on an X.509 object will no longer raise an `UnsupportedExtension` exception, instead an *UnrecognizedExtension* object will be returned. This behavior was based on a poor reading of the RFC, unknown critical extensions only need to be rejected on certificate verification.
- The CommonCrypto backend has been removed.
- MultiBackend has been removed.
- `Whirlpool` and `RIPEMD160` have been deprecated.

2.10.112 1.8.2 - 2017-05-26

- Fixed a compilation bug affecting OpenSSL 1.1.0f.
- Updated Windows and macOS wheels to be compiled against OpenSSL 1.1.0f.

2.10.113 1.8.1 - 2017-03-10

- Fixed macOS wheels to properly link against 1.1.0 rather than 1.0.2.

2.10.114 1.8 - 2017-03-09

- Added support for Python 3.6.
- Windows and macOS wheels now link against OpenSSL 1.1.0.
- macOS wheels are no longer universal. This change significantly shrinks the size of the wheels. Users on macOS 32-bit Python (if there are any) should migrate to 64-bit or build their own packages.
- Changed ASN.1 dependency from `pyasn1` to `asn1crypto` resulting in a general performance increase when encoding/decoding ASN.1 structures. Also, the `pyasn1_modules` test dependency is no longer required.

- Added support for *update_into()* on *CipherContext*.
- Added *private_bytes()* to *DHPrivateKey*.
- Added *public_bytes()* to *DHPublicKey*.
- *load_pem_private_key()* and *load_der_private_key()* now require that `password` must be bytes if provided. Previously this was documented but not enforced.
- Added support for subgroup order in *Diffie-Hellman key exchange*.

2.10.115 1.7.2 - 2017-01-27

- Updated Windows and macOS wheels to be compiled against OpenSSL 1.0.2k.

2.10.116 1.7.1 - 2016-12-13

- Fixed a regression in `int_from_bytes` where it failed to accept `bytearray`.

2.10.117 1.7 - 2016-12-12

- Support for OpenSSL 1.0.0 has been removed. Users on older version of OpenSSL will need to upgrade.
- Added support for Diffie-Hellman key exchange using *exchange()*.
- The OS random engine for OpenSSL has been rewritten to improve compatibility with embedded Python and other edge cases. More information about this change can be found in the pull request.

2.10.118 1.6 - 2016-11-22

- Deprecated support for OpenSSL 1.0.0. Support will be removed in `cryptography` 1.7.
- Replaced the Python-based OpenSSL locking callbacks with a C version to fix a potential deadlock that could occur if a garbage collection cycle occurred while inside the lock.
- Added support for *BLAKE2b* and *BLAKE2s* when using OpenSSL 1.1.0.
- Added *signature_algorithm_oid* support to *Certificate*.
- Added *signature_algorithm_oid* support to *CertificateSigningRequest*.
- Added *signature_algorithm_oid* support to *CertificateRevocationList*.
- Added support for *Scrypt* when using OpenSSL 1.1.0.
- Added a workaround to improve compatibility with Python application bundling tools like `PyInstaller` and `cx_freeze`.
- Added support for generating a *random_serial_number()*.
- Added support for encoding `IPv4Network` and `IPv6Network` in X.509 certificates for use with *NameConstraints*.
- Added *public_bytes()* to *Name*.
- Added *RelativeDistinguishedName*
- *DistributionPoint* now accepts *RelativeDistinguishedName* for *relative_name*. Deprecated use of *Name* as *relative_name*.
- *Name* now accepts an iterable of *RelativeDistinguishedName*. RDNs can be accessed via the *rdns* attribute. When constructed with an iterable of *NameAttribute*, each attribute becomes a single-valued RDN.
- Added *derive_private_key()*.

- Added support for signing and verifying RSA, DSA, and ECDSA signatures with *Prehashed* digests.

2.10.119 1.5.3 - 2016-11-05

- **SECURITY ISSUE**: Fixed a bug where `HKDF` would return an empty byte-string if used with a `length` less than `algorithm.digest_size`. Credit to **Markus Döring** for reporting the issue. *CVE-2016-9243*

2.10.120 1.5.2 - 2016-09-26

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2j.

2.10.121 1.5.1 - 2016-09-22

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2i.
- Resolved a `UserWarning` when used with cffi 1.8.3.
- Fixed a memory leak in name creation with X.509.
- Added a workaround for old versions of setuptools.
- Fixed an issue preventing `cryptography` from compiling against OpenSSL 1.0.2i.

2.10.122 1.5 - 2016-08-26

- Added *calculate_max_pss_salt_length()*.
- Added "one shot" *sign()* and *verify()* methods to DSA keys.
- Added "one shot" *sign()* and *verify()* methods to ECDSA keys.
- Switched back to the older callback model on Python 3.5 in order to mitigate the locking callback problem with OpenSSL <1.1.0.
- *CertificateBuilder*, *CertificateRevocationListBuilder*, and *RevokedCertificateBuilder* now accept timezone aware `datetime` objects as method arguments
- `cryptography` now supports OpenSSL 1.1.0 as a compilation target.

2.10.123 1.4 - 2016-06-04

- Support for OpenSSL 0.9.8 has been removed. Users on older versions of OpenSSL will need to upgrade.
- Added *KBKDFHMAC*.
- Added support for `OpenSSH` public key serialization.
- Added support for SHA-2 in RSA *OAEP* when using OpenSSL 1.0.2 or greater.
- Added "one shot" *sign()* and *verify()* methods to RSA keys.
- Deprecated the `serial` attribute on *Certificate*, in favor of *serial_number*.

2.10.124 1.3.4 - 2016-06-03

- Added another OpenSSL function to the bindings to support an upcoming `pyOpenSSL` release.

2.10.125 1.3.3 - 2016-06-02

- Added two new OpenSSL functions to the bindings to support an upcoming `pyOpenSSL` release.

2.10.126 1.3.2 - 2016-05-04

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2h.
- Fixed an issue preventing `cryptography` from compiling against LibreSSL 2.3.x.

2.10.127 1.3.1 - 2016-03-21

- Fixed a bug that caused an `AttributeError` when using `mock` to patch some `cryptography` modules.

2.10.128 1.3 - 2016-03-18

- Added support for padding ANSI X.923 with *ANSIX923*.
- Deprecated support for OpenSSL 0.9.8. Support will be removed in `cryptography` 1.4.
- Added support for the *PolicyConstraints* X.509 extension including both parsing and generation using *CertificateBuilder* and *CertificateSigningRequestBuilder*.
- Added *is_signature_valid* to *CertificateSigningRequest*.
- Fixed an intermittent `AssertionError` when performing an RSA decryption on an invalid ciphertext, `ValueError` is now correctly raised in all cases.
- Added *from_issuer_subject_key_identifier()*.

2.10.129 1.2.3 - 2016-03-01

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2g.

2.10.130 1.2.2 - 2016-01-29

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2f.

2.10.131 1.2.1 - 2016-01-08

- Reverts a change to an OpenSSL `EVP_PKEY` object that caused errors with `pyOpenSSL`.

2.10.132 1.2 - 2016-01-08

- **BACKWARDS INCOMPATIBLE:** *RevokedCertificate extensions* now uses extension classes rather than returning raw values inside the *Extension value*. The new classes are:
  - *CertificateIssuer*
  - *CRLReason*
  - *InvalidityDate*
- Deprecated support for OpenSSL 0.9.8 and 1.0.0. At this time there is no time table for actually dropping support, however we strongly encourage all users to upgrade, as those versions no longer receive support from the OpenSSL project.
- The *Certificate* class now has *signature* and *tbs_certificate_bytes* attributes.
- The *CertificateSigningRequest* class now has *signature* and *tbs_certrequest_bytes* attributes.
- The *CertificateRevocationList* class now has *signature* and *tbs_certlist_bytes* attributes.

- *NameConstraints* are now supported in the *CertificateBuilder* and *CertificateSigningRequestBuilder*.
- Support serialization of certificate revocation lists using the *public_bytes()* method of *CertificateRevocationList*.
- Add support for parsing *CertificateRevocationList extensions()* in the OpenSSL backend. The following extensions are currently supported:
  - *AuthorityInformationAccess*
  - *AuthorityKeyIdentifier*
  - *CRLNumber*
  - *IssuerAlternativeName*
- Added *CertificateRevocationListBuilder* and *RevokedCertificateBuilder* to allow creation of CRLs.
- Unrecognized non-critical X.509 extensions are now parsed into an *UnrecognizedExtension* object.

2.10.133 1.1.2 - 2015-12-10

- Fixed a SIGBUS crash with the OS X wheels caused by redefinition of a method.
- Fixed a runtime error `undefined symbol EC_GFp_nistp224_method` that occurred with some OpenSSL installations.
- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2e.

2.10.134 1.1.1 - 2015-11-19

- Fixed several small bugs related to compiling the OpenSSL bindings with unusual OpenSSL configurations.
- Resolved an issue where, depending on the method of installation and which Python interpreter they were using, users on El Capitan (OS X 10.11) may have seen an `InternalError` on import.

2.10.135 1.1 - 2015-10-28

- Added support for Elliptic Curve Diffie-Hellman with *ECDH*.
- Added *X963KDF*.
- Added support for parsing certificate revocation lists (CRLs) using *load_pem_x509_crl()* and *load_der_x509_crl()*.
- Add support for AES key wrapping with *aes_key_wrap()* and *aes_key_unwrap()*.
- Added a `__hash__` method to *Name*.
- Add support for encoding and decoding elliptic curve points to a byte string form using `cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers.encode_point` and `cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers.from_encoded_point`.
- Added *get_extension_for_class()*.
- *CertificatePolicies* are now supported in the *CertificateBuilder*.
- `countryName` is now encoded as a `PrintableString` when creating subject and issuer distinguished names with the Certificate and CSR builder classes.

2.10.136 1.0.2 - 2015-09-27

- **SECURITY ISSUE**: The OpenSSL backend prior to 1.0.2 made extensive use of assertions to check response codes where our tests could not trigger a failure. However, when Python is run with `-O` these asserts are optimized away. If a user ran Python with this flag and got an invalid response code this could result in undefined behavior or worse. Accordingly, all response checks from the OpenSSL backend have been converted from `assert` to a true function call. Credit **Emilia Käsper (Google Security Team)** for the report.

2.10.137 1.0.1 - 2015-09-05

- We now ship OS X wheels that statically link OpenSSL by default. When installing a wheel on OS X 10.10+ (and using a Python compiled against the 10.10 SDK) users will no longer need to compile. See *Installation* for alternate installation methods if required.
- Set the default string mask to UTF-8 in the OpenSSL backend to resolve character encoding issues with older versions of OpenSSL.
- Several new OpenSSL bindings have been added to support a future pyOpenSSL release.
- Raise an error during install on PyPy < 2.6. 1.0+ requires PyPy 2.6+.

2.10.138 1.0 - 2015-08-12

- Switched to the new cffi `set_source` out-of-line API mode for compilation. This results in significantly faster imports and lowered memory consumption. Due to this change we no longer support PyPy releases older than 2.6 nor do we support any released version of PyPy3 (until a version supporting cffi 1.0 comes out).
- Fix parsing of OpenSSH public keys that have spaces in comments.
- Support serialization of certificate signing requests using the `public_bytes` method of *`CertificateSigningRequest`*.
- Support serialization of certificates using the `public_bytes` method of *`Certificate`*.
- Add `get_provisioning_uri` method to *`HOTP`* and *`TOTP`* for generating provisioning URIs.
- Add *`ConcatKDFHash`* and *`ConcatKDFHMAC`*.
- Raise a `TypeError` when passing objects that are not text as the value to *`NameAttribute`*.
- Add support for *`OtherName`* as a general name type.
- Added new X.509 extension support in *`Certificate`* The following new extensions are now supported:
  - *`OCSPNoCheck`*
  - *`InhibitAnyPolicy`*
  - *`IssuerAlternativeName`*
  - *`NameConstraints`*
- Extension support was added to *`CertificateSigningRequest`*.
- Add support for creating signed certificates with *`CertificateBuilder`*. This includes support for the following extensions:
  - *`BasicConstraints`*
  - *`SubjectAlternativeName`*
  - *`KeyUsage`*
  - *`ExtendedKeyUsage`*
  - *`SubjectKeyIdentifier`*

  - *AuthorityKeyIdentifier*
  - *AuthorityInformationAccess*
  - *CRLDistributionPoints*
  - *InhibitAnyPolicy*
  - *IssuerAlternativeName*
  - *OCSPNoCheck*
- Add support for creating certificate signing requests with *CertificateSigningRequestBuilder*. This includes support for the same extensions supported in the `CertificateBuilder`.
- Deprecate `encode_rfc6979_signature` and `decode_rfc6979_signature` in favor of *encode_dss_signature()* and *decode_dss_signature()*.

2.10.139 0.9.3 - 2015-07-09

- Updated Windows wheels to be compiled against OpenSSL 1.0.2d.

2.10.140 0.9.2 - 2015-07-04

- Updated Windows wheels to be compiled against OpenSSL 1.0.2c.

2.10.141 0.9.1 - 2015-06-06

- **SECURITY ISSUE**: Fixed a double free in the OpenSSL backend when using DSA to verify signatures. Note that this only affects PyPy 2.6.0 and (presently unreleased) CFFI versions greater than 1.1.0.

2.10.142 0.9 - 2015-05-13

- Removed support for Python 3.2. This version of Python is rarely used and caused support headaches. Users affected by this should upgrade to 3.3+.
- Deprecated support for Python 2.6. At the time there is no time table for actually dropping support, however we strongly encourage all users to upgrade their Python, as Python 2.6 no longer receives support from the Python core team.
- Add support for the *SECP256K1* elliptic curve.
- Fixed compilation when using an OpenSSL which was compiled with the `no-comp` (`OPENSSL_NO_COMP`) option.
- Support *DER* serialization of public keys using the `public_bytes` method of *RSAPublicKey*, *DSAPublicKey*, and *EllipticCurvePublicKey*.
- Support *DER* serialization of private keys using the `private_bytes` method of *RSAPrivateKey*, *DSAPrivateKey*, and *EllipticCurvePrivateKey*.
- Add support for parsing X.509 certificate signing requests (CSRs) with *load_pem_x509_csr()* and *load_der_x509_csr()*.
- Moved `cryptography.exceptions.InvalidToken` to *cryptography.hazmat.primitives.twofactor.InvalidToken* and deprecated the old location. This was moved to minimize confusion between this exception and *cryptography.fernet.InvalidToken*.
- Added support for X.509 extensions in *Certificate* objects. The following extensions are supported as of this release:
  - *BasicConstraints*
  - *AuthorityKeyIdentifier*

- *SubjectKeyIdentifier*
- *KeyUsage*
- *SubjectAlternativeName*
- *ExtendedKeyUsage*
- *CRLDistributionPoints*
- *AuthorityInformationAccess*
- *CertificatePolicies*

Note that unsupported extensions with the critical flag raise `UnsupportedExtension` while unsupported extensions set to non-critical are silently ignored. Read the *X.509 documentation* for more information.

2.10.143 0.8.2 - 2015-04-10

- Fixed a race condition when initializing the OpenSSL or CommonCrypto backends in a multi-threaded scenario.

2.10.144 0.8.1 - 2015-03-20

- Updated Windows wheels to be compiled against OpenSSL 1.0.2a.

2.10.145 0.8 - 2015-03-08

- *load_ssh_public_key()* can now load elliptic curve public keys.
- Added *signature_hash_algorithm* support to *Certificate*.
- Added *rsa_recover_prime_factors()*
- *KeyDerivationFunction* was moved from `cryptography.hazmat.primitives.interfaces` to *kdf*.
- Added support for parsing X.509 names. See the *X.509 documentation* for more information.
- Added *load_der_private_key()* to support loading of DER encoded private keys and *load_der_public_key()* to support loading DER encoded public keys.
- Fixed building against LibreSSL, a compile-time substitute for OpenSSL.
- FreeBSD 9.2 was removed from the continuous integration system.
- Updated Windows wheels to be compiled against OpenSSL 1.0.2.
- *load_pem_public_key()* and *load_der_public_key()* now support PKCS1 RSA public keys (in addition to the previous support for SubjectPublicKeyInfo format for RSA, EC, and DSA).
- Added `EllipticCurvePrivateKeyWithSerialization` and deprecated `EllipticCurvePrivateKeyWithNumbers`.
- Added *private_bytes()* to *EllipticCurvePrivateKey*.
- Added `RSAPrivateKeyWithSerialization` and deprecated `RSAPrivateKeyWithNumbers`.
- Added *private_bytes()* to *RSAPrivateKey*.
- Added `DSAPrivateKeyWithSerialization` and deprecated `DSAPrivateKeyWithNumbers`.
- Added *private_bytes()* to *DSAPrivateKey*.
- Added `RSAPublicKeyWithSerialization` and deprecated `RSAPublicKeyWithNumbers`.
- Added `public_bytes` to *RSAPublicKey*.
- Added `EllipticCurvePublicKeyWithSerialization` and deprecated `EllipticCurvePublicKeyWithNumbers`.

- Added `public_bytes` to *EllipticCurvePublicKey*.
- Added `DSAPublicKeyWithSerialization` and deprecated `DSAPublicKeyWithNumbers`.
- Added `public_bytes` to *DSAPublicKey*.
- *HashAlgorithm* and *HashContext* were moved from `cryptography.hazmat.primitives.interfaces` to *hashes*.
- *CipherContext*, *AEADCipherContext*, *AEADEncryptionContext*, *CipherAlgorithm*, and *BlockCipherAlgorithm* were moved from `cryptography.hazmat.primitives.interfaces` to *ciphers*.
- *Mode*, *ModeWithInitializationVector*, *ModeWithNonce*, and *ModeWithAuthenticationTag* were moved from `cryptography.hazmat.primitives.interfaces` to *modes*.
- *PaddingContext* was moved from `cryptography.hazmat.primitives.interfaces` to *padding*.
- *AsymmetricPadding* was moved from `cryptography.hazmat.primitives.interfaces` to *padding*.
- `AsymmetricSignatureContext` and `AsymmetricVerificationContext` were moved from `cryptography.hazmat.primitives.interfaces` to `cryptography.hazmat.primitives.asymmetric`.
- *DSAParameters*, `DSAParametersWithNumbers`, *DSAPrivateKey*, `DSAPrivateKeyWithNumbers`, *DSAPublicKey* and `DSAPublicKeyWithNumbers` were moved from `cryptography.hazmat.primitives.interfaces` to *dsa*
- *EllipticCurve*, *EllipticCurveSignatureAlgorithm*, *EllipticCurvePrivateKey*, `EllipticCurvePrivateKeyWithNumbers`, *EllipticCurvePublicKey*, and `EllipticCurvePublicKeyWithNumbers` were moved from `cryptography.hazmat.primitives.interfaces` to *ec*.
- *RSAPrivateKey*, `RSAPrivateKeyWithNumbers`, *RSAPublicKey* and `RSAPublicKeyWithNumbers` were moved from `cryptography.hazmat.primitives.interfaces` to *rsa*.

2.10.146 0.7.2 - 2015-01-16

- Updated Windows wheels to be compiled against OpenSSL 1.0.1l.
- `enum34` is no longer installed on Python 3.4, where it is included in the standard library.
- Added a new function to the OpenSSL bindings to support additional functionality in pyOpenSSL.

2.10.147 0.7.1 - 2014-12-28

- Fixed an issue preventing compilation on platforms where `OPENSSL_NO_SSL3` was defined.

2.10.148 0.7 - 2014-12-17

- Cryptography has been relicensed from the Apache Software License, Version 2.0, to being available under *either* the Apache Software License, Version 2.0, or the BSD license.
- Added key-rotation support to *Fernet* with *MultiFernet*.
- More bit-lengths are now supported for `p` and `q` when loading DSA keys from numbers.
- Added `MACContext` as a common interface for CMAC and HMAC and deprecated `CMACContext`.
- Added support for encoding and decoding **RFC 6979** signatures in *Asymmetric Utilities*.
- Added *load_ssh_public_key()* to support the loading of OpenSSH public keys (**RFC 4253**). Only RSA and DSA keys are currently supported.
- Added initial support for X.509 certificate parsing. See the *X.509 documentation* for more information.

2.10.149 0.6.1 - 2014-10-15

- Updated Windows wheels to be compiled against OpenSSL 1.0.1j.
- Fixed an issue where OpenSSL 1.0.1j changed the errors returned by some functions.
- Added our license file to the `cryptography-vectors` package.
- Implemented DSA hash truncation support (per FIPS 186-3) in the OpenSSL backend. This works around an issue in 1.0.0, 1.0.0a, and 1.0.0b where truncation was not implemented.

2.10.150 0.6 - 2014-09-29

- Added *load_pem_private_key()* to ease loading private keys, and *load_pem_public_key()* to support loading public keys.
- Removed the, deprecated in 0.4, support for the `salt_length` argument to the *MGF1* constructor. The `salt_length` should be passed to *PSS* instead.
- Fix compilation on OS X Yosemite.
- Deprecated `elliptic_curve_private_key_from_numbers` and `elliptic_curve_public_key_from_numbers` in favor of `load_elliptic_curve_private_numbers` and `load_elliptic_curve_public_numbers` on `EllipticCurveBackend`.
- Added `EllipticCurvePrivateKeyWithNumbers` and `EllipticCurvePublicKeyWithNumbers` support.
- Work around three GCM related bugs in CommonCrypto and OpenSSL.
  - On the CommonCrypto backend adding AAD but not subsequently calling update would return null tag bytes.
  - One the CommonCrypto backend a call to update without an empty add AAD call would return null ciphertext bytes.
  - On the OpenSSL backend with certain versions adding AAD only would give invalid tag bytes.
- Support loading EC private keys from PEM.

2.10.151 0.5.4 - 2014-08-20

- Added several functions to the OpenSSL bindings to support new functionality in pyOpenSSL.
- Fixed a redefined constant causing compilation failure with Solaris 11.2.

2.10.152 0.5.3 - 2014-08-06

- Updated Windows wheels to be compiled against OpenSSL 1.0.1i.

2.10.153 0.5.2 - 2014-07-09

- Add `TraditionalOpenSSLSerializationBackend` support to `multibackend`.
- Fix compilation error on OS X 10.8 (Mountain Lion).

2.10.154 0.5.1 - 2014-07-07

- Add `PKCS8SerializationBackend` support to `multibackend`.

2.10.155 0.5 - 2014-07-07

- **BACKWARDS INCOMPATIBLE:** *GCM* no longer allows truncation of tags by default. Previous versions of `cryptography` allowed tags to be truncated by default, applications wishing to preserve this behavior (not recommended) can pass the `min_tag_length` argument.
- Windows builds now statically link OpenSSL by default. When installing a wheel on Windows you no longer need to install OpenSSL separately. Windows users can switch between static and dynamic linking with an environment variable. See *Installation* for more details.
- Added *HKDFExpand*.
- Added *CFB8* support for *AES* and *TripleDES* on `commoncrypto` and `openssl`.
- Added `AES` *CTR* support to the OpenSSL backend when linked against 0.9.8.
- Added `PKCS8SerializationBackend` and `TraditionalOpenSSLSerializationBackend` support to `openssl`.
- Added *Elliptic curve cryptography* and `EllipticCurveBackend`.
- Added *ECB* support for *TripleDES* on `commoncrypto` and `openssl`.
- Deprecated the concrete `RSAPrivateKey` class in favor of backend specific providers of the *cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey* interface.
- Deprecated the concrete `RSAPublicKey` in favor of backend specific providers of the *cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey* interface.
- Deprecated the concrete `DSAPrivateKey` class in favor of backend specific providers of the *cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey* interface.
- Deprecated the concrete `DSAPublicKey` class in favor of backend specific providers of the *cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey* interface.
- Deprecated the concrete `DSAParameters` class in favor of backend specific providers of the *cryptography.hazmat.primitives.asymmetric.dsa.DSAParameters* interface.
- Deprecated `encrypt_rsa`, `decrypt_rsa`, `create_rsa_signature_ctx` and `create_rsa_verification_ctx` on `RSABackend`.
- Deprecated `create_dsa_signature_ctx` and `create_dsa_verification_ctx` on `DSABackend`.

2.10.156 0.4 - 2014-05-03

- Deprecated `salt_length` on *MGF1* and added it to *PSS*. It will be removed from `MGF1` in two releases per our *API stability* policy.
- Added `SEED` support.
- Added *CMAC*.
- Added decryption support to *RSAPrivateKey* and encryption support to *RSAPublicKey*.
- Added signature support to *DSAPrivateKey* and verification support to *DSAPublicKey*.

2.10.157 0.3 - 2014-03-27

- Added *HOTP*.
- Added *TOTP*.
- Added `IDEA` support.
- Added signature support to *RSAPrivateKey* and verification support to *RSAPublicKey*.

- Moved test vectors to the new `cryptography_vectors` package.

2.10.158 0.2.2 - 2014-03-03

- Removed a constant definition that was causing compilation problems with specific versions of OpenSSL.

2.10.159 0.2.1 - 2014-02-22

- Fix a bug where importing cryptography from multiple paths could cause initialization to fail.

2.10.160 0.2 - 2014-02-20

- Added `commoncrypto`.
- Added initial `commoncrypto`.
- Removed `register_cipher_adapter` method from `CipherBackend`.
- Added support for the OpenSSL backend under Windows.
- Improved thread-safety for the OpenSSL backend.
- Fixed compilation on systems where OpenSSL's `ec.h` header is not available, such as CentOS.
- Added *PBKDF2HMAC*.
- Added *HKDF*.
- Added `multibackend`.
- Set default random for `openssl` to the OS random engine.
- Added `CAST5` (CAST-128) support.

2.10.161 0.1 - 2014-01-08

- Initial release.

2.11 Frequently asked questions

2.11.1 What issues can you help with in your issue tracker?

The primary purpose of our issue tracker is to enable us to identify and resolve bugs and feature requests in `cryptography`, so any time a user files a bug, we start by asking: Is this a `cryptography` bug, or is it a bug somewhere else?

That said, we do our best to help users to debug issues that are in their code or environments. Please note, however, that there's a limit to our ability to assist users in resolving problems that are specific to their environments, particularly when we have no way to reproduce the issue.

Lastly, we're not able to provide support for general Python or Python packaging issues.

2.11.2 I cannot suppress the deprecation warning that `cryptography` emits on import

Hint

The deprecation warning emitted on import does not inherit `DeprecationWarning` but inherits `UserWarning` instead.

If your pytest setup follows the best practices of failing on emitted warnings (`filterwarnings = error`), you may ignore it by adding the following line at the end of the list:

```
ignore:Python 2 is no longer supported by the Python core team. Support for it is now␣
↪deprecated in cryptography, and will be removed in a future release.:UserWarning
```

Note: Using `cryptography.utils.CryptographyDeprecationWarning` is not possible here because specifying it triggers `import cryptography` internally that emits the warning before the ignore rule even kicks in.

Ref: https://github.com/pytest-dev/pytest/issues/7524

The same applies when you use `filterwarnings()` in your code or invoke CPython with `-W` command line option.

2.11.3 `cryptography` failed to install!

If you are having issues installing `cryptography` the first troubleshooting step is to upgrade `pip` and then try to install again. For most users this will take the form of `pip install -U pip`, but on Windows you should do `python -m pip install -U pip`. If you are still seeing errors after upgrading and trying `pip install cryptography` again, please see the *Installation* documentation.

2.11.4 How does `cryptography` compare to NaCl (Networking and Cryptography Library)?

While `cryptography` and NaCl both share the goal of making cryptography easier, and safer, to use for developers, `cryptography` is designed to be a general purpose library, interoperable with existing systems, while NaCl features a collection of hand selected algorithms.

`cryptography`'s *recipes* layer has similar goals to NaCl.

If you prefer NaCl's design, we highly recommend PyNaCl, which is also maintained by the PyCA team.

2.11.5 Why use `cryptography`?

If you've done cryptographic work in Python before you have likely encountered other libraries in Python such as *M2Crypto*, *PyCrypto*, or *PyOpenSSL*. In building `cryptography` we wanted to address a few issues we observed in the legacy libraries:

- Extremely error prone APIs and insecure defaults.
- Use of poor implementations of algorithms (i.e. ones with known side-channel attacks).
- Lack of maintenance.
- Lack of high level APIs.
- Lack of PyPy and Python 3 support.
- Absence of algorithms such as *AES-GCM* and *HKDF*.

2.11.6 Why does `cryptography` require Rust?

`cryptography` uses OpenSSL (see: *Use of OpenSSL*) for its cryptographic operations. OpenSSL is the de facto standard for cryptographic libraries and provides high performance along with various certifications that may be relevant to developers. However, it is written in C and lacks memory safety. We want `cryptography` to be as secure as possible while retaining the advantages of OpenSSL, so we've chosen to rewrite non-cryptographic operations (such as ASN.1 parsing) in a high performance memory safe language: Rust.

2.11.7 `cryptography` raised an `InternalError` and I'm not sure what to do?

Frequently `InternalError` is raised when there are errors on the OpenSSL error stack that were placed there by other libraries that are also using OpenSSL. Try removing the other libraries and see if the problem persists. If you have no other libraries using OpenSSL in your process, or they do not appear to be at fault, it's possible that this is a bug in `cryptography`. Please file an issue with instructions on how to reproduce it.

2.11.8 Installing cryptography with OpenSSL older than 3.0.0 fails

The OpenSSL project has dropped support for the 0.9.8, 1.0.0, 1.0.1, 1.0.2, 1.1.0, and 1.1.1 release series. Since they are no longer receiving security patches from upstream, `cryptography` is also dropping support for them. To fix this issue you should upgrade to OpenSSL 3.0.0 or later. This may require you to upgrade to a newer operating system.

2.11.9 Installing `cryptography` fails with `error: Can not find Rust compiler`

Building `cryptography` from source requires you have *Rust installed and available* on your `PATH`. You may be able to fix this by upgrading to a newer version of `pip` which will install a pre-compiled `cryptography` wheel. If not, you'll need to install Rust. Follow the *instructions* to ensure you install a recent Rust version.

Rust is only required during the build phase of `cryptography`, you do not need to have Rust installed after you've built `cryptography`. This is the same as the C compiler toolchain which is also required to build `cryptography`, but not afterwards.

2.11.10 I'm getting errors installing or importing `cryptography` on AWS Lambda

Make sure you're following AWS's documentation either for building .zip archives for Lambda or building container images for Lambda.

2.11.11 Why are there no wheels for my Python3.x version?

Our Python3 wheels are `abi3` wheels. This means they support multiple versions of Python. The `abi3` wheel can be used with any version of Python greater than or equal to the version it specifies. Recent versions of `pip` will automatically install `abi3` wheels.

2.11.12 Why can't I import my PEM file?

PEM is a format (defined by several RFCs, but originally **RFC 1421**) for encoding keys, certificates, and others cryptographic data into a regular form. The data is encoded as base64 and wrapped with a header and footer.

If you are having trouble importing PEM files, make sure your file fits the following rules:

- has a one-line header like this: `-----BEGIN [FILE TYPE]-----` (where `[FILE TYPE]` is `CERTIFICATE`, `PUBLIC KEY`, `PRIVATE KEY`, etc.)
- has a one-line footer like this: `-----END [FILE TYPE]-----`
- all lines, except for the final one, must consist of exactly 64 characters.

For example, this is a PEM file for a RSA Public Key:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA7CsKFSzq20NLb2VQDXma
9DsDXtKADv0ziI5hT1KG6Bex5seE9pUoEcUxNv4uXo2jzAUgyRweRl/DLU8SoN8+
WWd6YWik4GZvNv7j0z28h9Q5jRySxy4dmElFtIRHGiKhqd1Z06z4AzrmKEzgxkOk
LJjY9cvwD+iXjpK2oJwNNyavvjb5YZq6V60RhpyNtKpMh2+zRLgIk9sROEPQeYfK
22zj2CnGBMg5Gm2uPOsGDltl/I/Fdh1aO3X4i1GXwCuPf1kSAg6lPJD0batftkSG
v0X0heUaV0j1HSNlBWamT4IR9+iJfKJHekOqvHQBcaCu7Ja4kXzx6GZ3M2j/Ja3A
2QIDAQAB
-----END PUBLIC KEY-----
```

2.11.13 What happened to the backend argument?

`cryptography` stopped requiring the use of `backend` arguments in version 3.1 and deprecated their use in version 36.0. If you are on an older version that requires these arguments please view the appropriate documentation version or upgrade to the latest release.

Note that for forward compatibility `backend` is still silently accepted by functions that previously required it, but it is ignored and no longer documented.

2.11.14 Will you upload wheels for my non-x86 non-ARM64 CPU architecture?

Maybe! But there's some pre-requisites. For us to build wheels and upload them to PyPI, we consider it necessary to run our tests for that architecture as a part of our CI (i.e. for every commit). If we don't run the tests, it's hard to have confidence that everything works – particularly with cryptography, which frequently employs per-architecture assembly code.

For us to add something to CI we need a provider which offers builds on that architecture, which integrate into our workflows, has sufficient capacity, and performs well enough not to regress the contributor experience. We don't think this is an insurmountable bar, but it's also not one that can be cleared lightly.

If you are interested in helping support a new CPU architecture, we encourage you to reach out, discuss, and contribute that support. We will attempt to be supportive, but we cannot commit to doing the work ourselves.

2.12 Development

As an open source project, `cryptography` welcomes contributions of all forms. The sections below will help you get started.

File bugs and feature requests on our issue tracker on GitHub. If it is a bug check out what to put in your bug report.

2.12.1 Getting started

Development dependencies

Working on `cryptography` requires the installation of a small number of development dependencies in addition to the dependencies for *Installation* (including *Rust*). These are handled by the use of `nox`, which can be installed with `pip`.

```
$ # Create a virtualenv and activate it
$ # Set up your cryptography build environment
$ pip install nox
$ nox -e local
```

OpenSSL on macOS

You must have installed OpenSSL (via Homebrew or MacPorts) before invoking `nox` or else pip will fail to compile.

Running tests

`cryptography` unit tests are found in the `tests/` directory and are designed to be run using pytest. `nox` automatically invokes `pytest` and other required checks for `cryptography`:

```
$ nox -e local
```

You can also specify a subset of tests to run as positional arguments:

```
$ # run the whole x509 testsuite, plus the fernet tests
$ nox -e local -- tests/x509/ tests/test_fernet.py
```

Building the docs

Building the docs on non-Windows platforms requires manually installing the C library `libenchant` (installation instructions). The docs can be built using `nox`:

```
$ nox -e docs
```

2.12.2 Submitting patches

- Always make a new branch for your work.
- Patches should be small to facilitate easier review. Studies have shown that review quality falls off as patch size grows. Sometimes this will result in many small PRs to land a single large feature.
- Larger changes should be discussed on our mailing list before submission.
- New features and significant bug fixes should be documented in the *Changelog*.
- You must have legal permission to distribute any code you contribute to `cryptography`, and it must be available under both the BSD and Apache Software License Version 2.0 licenses.

If you believe you've identified a security issue in `cryptography`, please follow the directions on the *security page*.

Code

When in doubt, refer to **PEP 8** for Python code. You can check if your code meets our automated requirements by formatting it with `ruff format` and running `ruff` against it. If you've installed the development requirements this will automatically use our configuration. You can also run the `nox` job with `nox -e flake`.

Write comments as complete sentences.

Class names which contains acronyms or initialisms should always be capitalized. A class should be named `HTTPClient`, not `HttpClient`.

Every code file must start with the boilerplate licensing notice:

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.
```

API considerations

Most projects' APIs are designed with a philosophy of "make easy things easy, and make hard things possible". One of the perils of writing cryptographic code is that secure code looks just like insecure code, and its results are almost always indistinguishable. As a result, `cryptography` has, as a design philosophy: "make it hard to do insecure things". Here are a few strategies for API design that should be both followed, and should inspire other API choices:

If it is necessary to compare a user provided value with a computed value (for example, verifying a signature), there should be an API provided that performs the verification in a secure way (for example, using a constant time comparison), rather than requiring the user to perform the comparison themselves.

If it is incorrect to ignore the result of a method, it should raise an exception, and not return a boolean `True`/`False` flag. For example, a method to verify a signature should raise `InvalidSignature`, and not return whether the signature was valid.

```
# This is bad.
def verify(sig: bytes) -> bool:
    # ...
    return is_valid

# Good!
def verify(sig: bytes) -> None:
    # ...
    if not is_valid:
        raise InvalidSignature
```

Every recipe should include a version or algorithmic marker of some sort in its output in order to allow transparent upgrading of the algorithms in use, as the algorithms or parameters needed to achieve a given security margin evolve.

C bindings

More information on C bindings can be found in *the dedicated section of the documentation*.

Tests

All code changes must be accompanied by unit tests with 100% code coverage (as measured by the combined metrics across our build matrix).

When implementing a new primitive or recipe `cryptography` requires that you provide a set of test vectors. See *Test vectors* for more details.

Documentation

All features should be documented with prose in the `docs` section. To ensure it builds you can run `nox -e docs`.

Because of the inherent challenges in implementing correct cryptographic systems, we want to make our documentation point people in the right directions as much as possible. To that end:

- When documenting a generic interface, use a strong algorithm in examples. (e.g. when showing a hashing example, don't use *MD5*)
- When giving prescriptive advice, always provide references and supporting material.
- When there is real disagreement between cryptographic experts, represent both sides of the argument and describe the trade-offs clearly.

When documenting a new module in the `hazmat` package, its documentation should begin with the "Hazardous Materials" warning:

```
.. hazmat::
```

Always prefer terminology that is most broadly accepted. For example:

- When referring to class instances use "an instance of `Foo`" instead of "a `Foo` provider".

When referring to a hypothetical individual (such as "a person receiving an encrypted message") use gender neutral pronouns (they/them/their).

Docstrings are typically only used when writing abstract classes, but should be written like this if required:

```
def some_function(some_arg):
    """
    Does some things.

    :param some_arg: Some argument.
    """
```

So, specifically:

- Always use three double quotes.
- Put the three double quotes on their own line.
- No blank line at the end.
- Use Sphinx parameter/attribute documentation syntax.

2.12.3 Reviewing and merging patches

Everyone is encouraged to review open pull requests. We only ask that you try and think carefully, ask questions and are excellent to one another. Code review is our opportunity to share knowledge, design ideas and make friends.

When reviewing a patch try to keep each of these concepts in mind:

Intent

- What is the change being proposed?
- Do we want this feature or is the bug they're fixing really a bug?

Architecture

- Is the proposed change being made in the correct place? Is it a fix in the backend when it should be in the primitives?

Implementation

- Does the change do what the author claims?
- Are there sufficient tests?
- Has it been documented?
- Will this change introduce new bugs?

Grammar and style

These are small things that are not caught by the automated style checkers.

- Does a variable need a better name?
- Should this be a keyword argument?

Merge requirements

Because cryptography is so complex, and the implications of getting it wrong so devastating, `cryptography` has a strict merge policy for committers:

- Patches must *never* be pushed directly to `main`, all changes (even the most trivial typo fixes!) must be submitted as a pull request.
- A committer may *never* merge their own pull request, a second party must merge their changes. If multiple people work on a pull request, it must be merged by someone who did not work on it.
- A patch that breaks tests, or introduces regressions by changing or removing existing tests should not be merged. Tests must always be passing on `main`.
- If somehow the tests get into a failing state on `main` (such as by a backwards incompatible release of a dependency) no pull requests may be merged until this is rectified.
- All merged patches must have 100% test coverage.

The purpose of these policies is to minimize the chances we merge a change that jeopardizes our users' security.

2.12.4 Test vectors

Testing the correctness of the primitives implemented in `cryptography` requires trusted test vectors. Where possible these vectors are obtained from official sources such as NIST or IETF RFCs. When this is not possible `cryptography` has chosen to create a set of custom vectors using an official vector file as input.

Vectors are kept in the `cryptography_vectors` package rather than within our main test suite.

Sources

Project Wycheproof

We run vectors from Project Wycheproof – a collection of known edge-cases for various cryptographic algorithms. These are not included in the repository (or `cryptography_vectors` package), but rather cloned from Git in our continuous integration environments.

Asymmetric ciphers

- RSA PKCS #1 from the RSA FTP site (ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/ and ftp://ftp.rsa.com/pub/rsalabs/tmp/).
- RSA FIPS 186-2 and PKCS1 v1.5 vulnerability test vectors from NIST CAVP.
- FIPS 186-2 and FIPS 186-3 DSA test vectors from NIST CAVP.
- FIPS 186-2 and FIPS 186-3 ECDSA test vectors from NIST CAVP.
- DH and ECDH and ECDH+KDF(17.4) test vectors from NIST CAVP.
- Ed25519 test vectors from the Ed25519 website.
- `asymmetric/PEM_Serialization/rsa-bad-1025-q-is-2.pem` from badkeys.
- OpenSSL PEM DSA serialization vectors from the GnuTLS example keys.

- PKCS #8 PEM serialization vectors from
  - GnuTLS: enc-rsa-pkcs8.pem, enc2-rsa-pkcs8.pem, unenc-rsa-pkcs8.pem, pkcs12_s2k_pem.c. The encoding error in unenc-rsa-pkcs8.pem was fixed, and the contents of enc-rsa-pkcs8.pem was re-encrypted to include it. The contents of enc2-rsa-pkcs8.pem was re-encrypted using a stronger PKCS#8 cipher.
  - Botan's ECC private keys.
- asymmetric/public/PKCS1/dsa.pub.pem is a PKCS1 DSA public key from the Ruby test suite.
- X25519 and X448 test vectors from **RFC 7748**.
- RSA OAEP with custom label from the BoringSSL evp tests.
- Ed448 test vectors from **RFC 8032**.
- Deterministic ECDSA (**RFC 6979**) from OpenSSL's RFC 6979 test vectors.
- `asymmetric/PKCS8/rsa-40bitrc2.pem` a PKCS8 encoded RSA key from GnuTLS encrypted with `pbeWithSHAAnd40BitRC2-CBC`. The password is `baz`.
- `asymmetric/PKCS8/rsa-rc2-cbc.pem` a PKCS8 encoded RSA key from GnuTLS encrypted with `RC2-CBC`. The password is `Red Hat Enterprise Linux 7.4`.
- `asymmetric/PKCS8/rsa_pkcs8_pbes2_pbkdf2_2048_3des_sha224.pem` a PKCS8 encoded RSA key from Mbed-TLS using `sha224` as the PRF for PBKDF2. The password is `PolarSSLTest`.
- `asymmetric/PKCS8/rsa_pkcs8_pbes2_pbkdf2_2048_3des_sha384.pem` a PKCS8 encoded RSA key from Mbed-TLS using `sha384` as the PRF for PBKDF2. The password is `PolarSSLTest`.
- `asymmetric/PKCS8/rsa_pkcs8_pbes2_pbkdf2_2048_3des_sha512.pem` a PKCS8 encoded RSA key from Mbed-TLS using `sha512` as the PRF for PBKDF2. The password is `PolarSSLTest`.
- `asymmetric/PKCS8/rsa-aes-192-cbc.pem` a PKCS8 encoded RSA key from Mbed-TLS encrypted with `AES-192-CBC`. The password is `PolarSSLTest`.
- `asymmetric/PKCS8/ed25519-scrypt.pem` a PKCS8 encoded Ed25519 key from RustCrypto using scrypt as the KDF. The password is `hunter42`.
- FIPS 203 ML-KEM-{512,768,1024} KAT vectors from post-quantum-cryptography/KAT.
- FIPS 204 ML-DSA-{44,65,87} KAT vectors from post-quantum-cryptography/KAT.

Custom asymmetric vectors

SECP256K1 vector creation

This page documents the code that was used to generate the SECP256K1 elliptic curve test vectors as well as code used to verify them against another implementation.

Creation

The vectors are generated using a pure Python ecdsa implementation. The test messages and combinations of algorithms are derived from the NIST vector data.

```
import hashlib
import os
from binascii import hexlify
from collections import defaultdict

from ecdsa import SECP256k1, SigningKey
from ecdsa.util import sigdecode_der, sigencode_der
```

(continues on next page)

(continued from previous page)

```
from cryptography_vectors import open_vector_file
from tests.utils import load_fips_ecdsa_signing_vectors, load_vectors_from_file

HASHLIB_HASH_TYPES = {
    "SHA-1": hashlib.sha1,
    "SHA-224": hashlib.sha224,
    "SHA-256": hashlib.sha256,
    "SHA-384": hashlib.sha384,
    "SHA-512": hashlib.sha512,
}


class TruncatedHash:
    def __init__(self, hasher):
        self.hasher = hasher

    def __call__(self, data):
        self.hasher.update(data)
        return self

    def digest(self):
        return self.hasher.digest()[: 256 // 8]


def build_vectors(fips_vectors):
    vectors = defaultdict(list)
    for vector in fips_vectors:
        vectors[vector["digest_algorithm"]].append(vector["message"])

    for digest_algorithm, messages in vectors.items():
        if digest_algorithm not in HASHLIB_HASH_TYPES:
            continue

        yield ""
        yield f"[K-256,{digest_algorithm}]"
        yield ""

        for message in messages:
            # Make a hash context
            hash_func = TruncatedHash(HASHLIB_HASH_TYPES[digest_algorithm]())

            # Sign the message using warner/ecdsa
            secret_key = SigningKey.generate(curve=SECP256k1)
            public_key = secret_key.get_verifying_key()
            signature = secret_key.sign(
                message, hashfunc=hash_func, sigencode=sigencode_der
            )

            r, s = sigdecode_der(signature, None)

            yield f"Msg = {hexlify(message)}"
```

(continues on next page)

(continued from previous page)

```
            yield f"d = {secret_key.privkey.secret_multiplier:x}"
            yield f"Qx = {public_key.pubkey.point.x():x}"
            yield f"Qy = {public_key.pubkey.point.y():x}"
            yield f"R = {r:x}"
            yield f"S = {s:x}"
            yield ""


def write_file(lines, dest):
    for line in lines:
        print(line)
        print(line, file=dest)


source_path = os.path.join("asymmetric", "ECDSA", "FIPS_186-3", "SigGen.txt")
dest_path = os.path.join("asymmetric", "ECDSA", "SECP256K1", "SigGen.txt")

fips_vectors = load_vectors_from_file(
    source_path, load_fips_ecdsa_signing_vectors
)

with open_vector_file(dest_path, "w") as dest_file:
    write_file(build_vectors(fips_vectors), dest_file)
```

Download link: `generate_secp256k1.py`

Verification

`cryptography` was modified to support the SECP256K1 curve. Then the following python script was run to generate the vector files.

```
import os

from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric.utils import (
    encode_dss_signature,
)
from tests.utils import load_fips_ecdsa_signing_vectors, load_vectors_from_file

CRYPTOGRAPHY_HASH_TYPES = {
    "SHA-1": hashes.SHA1,
    "SHA-224": hashes.SHA224,
    "SHA-256": hashes.SHA256,
    "SHA-384": hashes.SHA384,
    "SHA-512": hashes.SHA512,
}


def verify_one_vector(vector):
    digest_algorithm = vector["digest_algorithm"]
    message = vector["message"]
```

(continues on next page)

(continued from previous page)

```
    x = vector["x"]
    y = vector["y"]
    signature = encode_dss_signature(vector["r"], vector["s"])

    numbers = ec.EllipticCurvePublicNumbers(x, y, ec.SECP256K1())

    key = numbers.public_key()

    verifier = key.verifier(
        signature, ec.ECDSA(CRYPTOGRAPHY_HASH_TYPES[digest_algorithm]())
    )
    verifier.update(message)
    verifier.verify()


def verify_vectors(vectors):
    for vector in vectors:
        verify_one_vector(vector)


vector_path = os.path.join("asymmetric", "ECDSA", "SECP256K1", "SigGen.txt")

secp256k1_vectors = load_vectors_from_file(
    vector_path, load_fips_ecdsa_signing_vectors
)

verify_vectors(secp256k1_vectors)
```

Download link: `verify_secp256k1.py`

RSA OAEP SHA2 vector creation

This page documents the code that was used to generate the RSA OAEP SHA2 test vectors as well as code used to verify them against another implementation.

Creation

`cryptography` was modified to allow the use of SHA2 in OAEP encryption. Then the following python script was run to generate the vector files.

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.

import binascii
import itertools
import os

from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa
from tests.utils import load_pkcs1_vectors, load_vectors_from_file
```

(continues on next page)

(continued from previous page)

```
def build_vectors(mgf1alg, hashalg, filename):
    vectors = load_vectors_from_file(filename, load_pkcs1_vectors)

    output = []
    for vector in vectors:
        # RSA keys for this must be long enough to accommodate the length of
        # the underlying hash function. This means we can't use the keys from
        # the sha1 test vectors for sha512 tests because 1024-bit keys are too
        # small. Instead we parse the vectors for the test cases, then
        # generate our own 2048-bit keys for each.
        private, _ = vector
        skey = rsa.generate_private_key(65537, 2048)
        pn = skey.private_numbers()
        examples = private["examples"]
        output.append("# ============================================")
        output.append("# Example")
        output.append("# Public key")
        output.append("# Modulus:")
        output.append(format(pn.public_numbers.n, "x"))
        output.append("# Exponent:")
        output.append(format(pn.public_numbers.e, "x"))
        output.append("# Private key")
        output.append("# Modulus:")
        output.append(format(pn.public_numbers.n, "x"))
        output.append("# Public exponent:")
        output.append(format(pn.public_numbers.e, "x"))
        output.append("# Exponent:")
        output.append(format(pn.d, "x"))
        output.append("# Prime 1:")
        output.append(format(pn.p, "x"))
        output.append("# Prime 2:")
        output.append(format(pn.q, "x"))
        output.append("# Prime exponent 1:")
        output.append(format(pn.dmp1, "x"))
        output.append("# Prime exponent 2:")
        output.append(format(pn.dmq1, "x"))
        output.append("# Coefficient:")
        output.append(format(pn.iqmp, "x"))
        pkey = skey.public_key()
        vectorkey = rsa.RSAPrivateNumbers(
            p=private["p"],
            q=private["q"],
            d=private["private_exponent"],
            dmp1=private["dmp1"],
            dmq1=private["dmq1"],
            iqmp=private["iqmp"],
            public_numbers=rsa.RSAPublicNumbers(
                e=private["public_exponent"], n=private["modulus"]
            ),
        ).private_key()
        count = 1
```

(continues on next page)

(continued from previous page)

```
        for example in examples:
            message = vectorkey.decrypt(
                binascii.unhexlify(example["encryption"]),
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA1()),
                    algorithm=hashes.SHA1(),
                    label=None,
                ),
            )
            assert message == binascii.unhexlify(example["message"])
            ct = pkey.encrypt(
                message,
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=mgf1alg),
                    algorithm=hashalg,
                    label=None,
                ),
            )
            output.append(
                f"# OAEP Example {count} alg={hashalg.name} "
                f"mgf1={mgf1alg.name}"
            )
            count += 1
            output.append("# Message:")
            output.append(example["message"].decode("utf-8"))
            output.append("# Encryption:")
            output.append(binascii.hexlify(ct).decode("utf-8"))

    return "\n".join(output)


def write_file(data, filename):
    with open(filename, "w") as f:
        f.write(data)


oaep_path = os.path.join(
    "asymmetric", "RSA", "pkcs-1v2-1d2-vec", "oaep-vect.txt"
)
hashalgs = [
    hashes.SHA1(),
    hashes.SHA224(),
    hashes.SHA256(),
    hashes.SHA384(),
    hashes.SHA512(),
]
for hashtuple in itertools.product(hashalgs, hashalgs):
    if isinstance(hashtuple[0], hashes.SHA1) and isinstance(
        hashtuple[1], hashes.SHA1
    ):
        continue
```

(continues on next page)

(continued from previous page)

```
    write_file(
        build_vectors(hashtuple[0], hashtuple[1], oaep_path),
        f"oaep-{hashtuple[0].name}-{hashtuple[1].name}.txt",
    )
```

Download link: `generate_rsa_oaep_sha2.py`

Verification

A Java 8 program was written using Bouncy Castle to load and verify the test vectors.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigInteger;
import java.security.AlgorithmParameters;
import java.security.GeneralSecurityException;
import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.Security;
import java.security.spec.AlgorithmParameterSpec;
import java.security.spec.MGF1ParameterSpec;
import java.security.spec.RSAPrivateKeySpec;
import java.util.Arrays;

import javax.crypto.Cipher;
import javax.crypto.spec.OAEPParameterSpec;
import javax.crypto.spec.PSource;
import javax.xml.bind.DatatypeConverter;

import org.bouncycastle.jce.provider.BouncyCastleProvider;

class TestVectorData {
    public BigInteger pub_key_modulus;
    public BigInteger pub_key_exponent;
    public BigInteger priv_key_public_exponent;
    public BigInteger priv_key_modulus;
    public BigInteger priv_key_exponent;
    public BigInteger priv_key_prime_1;
    public BigInteger priv_key_prime_2;
    public BigInteger priv_key_prime_exponent_1;
    public BigInteger priv_key_prime_exponent_2;
    public BigInteger priv_key_coefficient;
    public byte[] plaintext;
    public byte[] ciphertext;
}

class TestVectorLoader {
    private static final String FILE_HEADER = "# RSA OAEP SHA2 vectors built";
    private static final String EXAMPLE_HEADER = "# =====";
    private static final String EXAMPLE = "# Example";
```

(continues on next page)

(continued from previous page)

```
    private static final String PUBLIC_KEY = "# Public key";
    private static final String PUB_MODULUS = "# Modulus:";
    private static final String PUB_EXPONENT = "# Exponent:";
    private static final String PRIVATE_KEY = "# Private key";
    private static final String PRIV_MODULUS = "# Modulus:";
    private static final String PRIV_PUBLIC_EXPONENT = "# Public exponent:";
    private static final String PRIV_EXPONENT = "# Exponent:";
    private static final String PRIV_PRIME_1 = "# Prime 1:";
    private static final String PRIV_PRIME_2 = "# Prime 2:";
    private static final String PRIV_PRIME_EXPONENT_1 = "# Prime exponent 1:";
    private static final String PRIV_PRIME_EXPONENT_2 = "# Prime exponent 2:";
    private static final String PRIV_COEFFICIENT = "# Coefficient:";
    private static final String OAEP_EXAMPLE_HEADER = "# OAEP Example";
    private static final String MESSAGE = "# Message:";
    private static final String ENCRYPTION = "# Encryption:";

    private BufferedReader m_reader = null;
    private FileReader m_file_reader = null;
    private TestVectorData m_data = null;

    TestVectorLoader() {

    }

    protected void finalize() {
        close();
    }

    public void open(String path) throws IOException {
        close();
        m_file_reader = new FileReader(path);
        m_reader = new BufferedReader(m_file_reader);
        m_data = new TestVectorData();
    }

    public void close() {
        try {
            if (m_reader != null) {
                m_reader.close();
                m_reader = null;
            }
            if (m_file_reader != null) {
                m_file_reader.close();
                m_file_reader = null;
            }
            m_data = null;
        } catch (IOException e) {
            System.out.println("Exception closing files");
            e.printStackTrace();
        }
    }
```

(continues on next page)

(continued from previous page)

```
    public TestVectorData loadNextTest() throws IOException {
        if (m_file_reader == null || m_reader == null || m_data == null) {
            throw new IOException("A test vector file must be opened first");
        }

        String line = m_reader.readLine();

        if (line == null) {
            // end of file
            return null;
        }

        if (line.startsWith(FILE_HEADER)) {
            // start of file
            skipFileHeader(m_reader);
            line = m_reader.readLine();
        }

        if (line.startsWith(OAEP_EXAMPLE_HEADER)) {
            // Next example, keep existing keys and load next message
            loadMessage(m_reader, m_data);
            return m_data;
        }

        // otherwise it's a new example
        if (!line.startsWith(EXAMPLE_HEADER)) {
            throw new IOException("Test Header Missing");
        }
        startNewTest(m_reader);
        m_data = new TestVectorData();

        line = m_reader.readLine();
        if (!line.startsWith(PUBLIC_KEY))
            throw new IOException("Public Key Missing");
        loadPublicKey(m_reader, m_data);

        line = m_reader.readLine();
        if (!line.startsWith(PRIVATE_KEY))
            throw new IOException("Private Key Missing");
        loadPrivateKey(m_reader, m_data);

        line = m_reader.readLine();
        if (!line.startsWith(OAEP_EXAMPLE_HEADER))
            throw new IOException("Message Missing");
        loadMessage(m_reader, m_data);

        return m_data;
    }

    private byte[] unhexlify(String line) {
        byte[] bytes = DatatypeConverter.parseHexBinary(line);
        return bytes;
```

(continues on next page)

(continued from previous page)

```
    }

    private BigInteger readBigInteger(BufferedReader br) throws IOException {
        return new BigInteger(br.readLine(), 16);
    }

    private void skipFileHeader(BufferedReader br) throws IOException {
        br.readLine(); // # # Derived from the NIST OAEP SHA1 vectors
        br.readLine(); // # # Verified against the Bouncy Castle OAEP SHA2 implementation
        br.readLine(); // #
    }

    private void startNewTest(BufferedReader br) throws IOException {
        String line = br.readLine();
        if (!line.startsWith(EXAMPLE))
            throw new IOException("Example Header Missing");
    }

    private void loadPublicKey(BufferedReader br, TestVectorData data) throws␣
↪IOException {
        String line = br.readLine();
        if (!line.startsWith(PUB_MODULUS))
            throw new IOException("Public Key Modulus Missing");
        data.pub_key_modulus = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PUB_EXPONENT))
            throw new IOException("Public Key Exponent Missing");
        data.pub_key_exponent = readBigInteger(br);
    }

    private void loadPrivateKey(BufferedReader br, TestVectorData data) throws␣
↪IOException {
        String line = br.readLine();
        if (!line.startsWith(PRIV_MODULUS))
            throw new IOException("Private Key Modulus Missing");
        data.priv_key_modulus = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_PUBLIC_EXPONENT))
            throw new IOException("Private Key Public Exponent Missing");
        data.priv_key_public_exponent = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_EXPONENT))
            throw new IOException("Private Key Exponent Missing");
        data.priv_key_exponent = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_PRIME_1))
            throw new IOException("Private Key Prime 1 Missing");
        data.priv_key_prime_1 = readBigInteger(br);
```

(continues on next page)

(continued from previous page)

```
        line = br.readLine();
        if (!line.startsWith(PRIV_PRIME_2))
            throw new IOException("Private Key Prime 2 Missing");
        data.priv_key_prime_2 = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_PRIME_EXPONENT_1))
            throw new IOException("Private Key Prime Exponent 1 Missing");
        data.priv_key_prime_exponent_1 = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_PRIME_EXPONENT_2))
            throw new IOException("Private Key Prime Exponent 2 Missing");
        data.priv_key_prime_exponent_2 = readBigInteger(br);

        line = br.readLine();
        if (!line.startsWith(PRIV_COEFFICIENT))
            throw new IOException("Private Key Coefficient Missing");
        data.priv_key_coefficient = readBigInteger(br);
    }

    private void loadMessage(BufferedReader br, TestVectorData data) throws IOException {
        String line = br.readLine();
        if (!line.startsWith(MESSAGE))
            throw new IOException("Plaintext Missing");
        data.plaintext = unhexlify(br.readLine());

        line = br.readLine();
        if (!line.startsWith(ENCRYPTION))
            throw new IOException("Ciphertext Missing");
        data.ciphertext = unhexlify(br.readLine());
    }

}

public class VerifyRSAOAEPSHA2 {

    public enum SHAHash {
        SHA1, SHA224, SHA256, SHA384, SHA512
    }

    private SHAHash m_mgf1_hash;
    private SHAHash m_alg_hash;
    private Cipher m_cipher;
    private PrivateKey m_private_key;
    private AlgorithmParameters m_algo_param;

    VerifyRSAOAEPSHA2(SHAHash mgf1_hash, SHAHash alg_hash, TestVectorData test_data)␣
↪throws Exception {

        m_mgf1_hash = mgf1_hash;
```

(continues on next page)

(continued from previous page)

```
        m_alg_hash = alg_hash;

        MGF1ParameterSpec mgf1_spec = getMGF1ParameterSpec(m_mgf1_hash);
        AlgorithmParameterSpec algo_param_spec = getAlgorithmParameterSpec(m_alg_hash,␣
↪mgf1_spec);

        m_algo_param = AlgorithmParameters.getInstance("OAEP");
        m_algo_param.init(algo_param_spec);

        m_private_key = loadPrivateKey(test_data);

        m_cipher = getCipher(m_alg_hash);
    }

    private Cipher getCipher(SHAHash alg_hash) throws GeneralSecurityException {
        Cipher cipher = null;

        switch (alg_hash) {

        case SHA1:
            cipher = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding", "BC");
            break;

        case SHA224:
            cipher = Cipher.getInstance("RSA/ECB/OAEPwithSHA-224andMGF1Padding", "BC");
            break;

        case SHA256:
            cipher = Cipher.getInstance("RSA/ECB/OAEPwithSHA-256andMGF1Padding", "BC");
            break;

        case SHA384:
            cipher = Cipher.getInstance("RSA/ECB/OAEPwithSHA-384andMGF1Padding", "BC");
            break;

        case SHA512:
            cipher = Cipher.getInstance("RSA/ECB/OAEPwithSHA-512andMGF1Padding", "BC");
            break;
        }

        return cipher;
    }

    private MGF1ParameterSpec getMGF1ParameterSpec(SHAHash mgf1_hash) {
        MGF1ParameterSpec mgf1 = null;

        switch (mgf1_hash) {

        case SHA1:
            mgf1 = MGF1ParameterSpec.SHA1;
            break;
        case SHA224:
```

(continues on next page)

(continued from previous page)

```
            mgf1 = MGF1ParameterSpec.SHA224;
            break;

        case SHA256:
            mgf1 = MGF1ParameterSpec.SHA256;
            break;

        case SHA384:
            mgf1 = MGF1ParameterSpec.SHA384;
            break;

        case SHA512:
            mgf1 = MGF1ParameterSpec.SHA512;
            break;
        }

        return mgf1;
    }

    private AlgorithmParameterSpec getAlgorithmParameterSpec(SHAHash alg_hash,␣
↪MGF1ParameterSpec mgf1_spec) {

        OAEPParameterSpec oaep_spec = null;

        switch (alg_hash) {

        case SHA1:
            oaep_spec = new OAEPParameterSpec("SHA1", "MGF1", mgf1_spec, PSource.
↪PSpecified.DEFAULT);
            break;

        case SHA224:
            oaep_spec = new OAEPParameterSpec("SHA-224", "MGF1", mgf1_spec, PSource.
↪PSpecified.DEFAULT);
            break;

        case SHA256:
            oaep_spec = new OAEPParameterSpec("SHA-256", "MGF1", mgf1_spec, PSource.
↪PSpecified.DEFAULT);
            break;

        case SHA384:
            oaep_spec = new OAEPParameterSpec("SHA-384", "MGF1", mgf1_spec, PSource.
↪PSpecified.DEFAULT);
            break;

        case SHA512:
            oaep_spec = new OAEPParameterSpec("SHA-512", "MGF1", mgf1_spec, PSource.
↪PSpecified.DEFAULT);
            break;
        }
```

(continues on next page)

(continued from previous page)

```
        return oaep_spec;
    }

    private PrivateKey loadPrivateKey(TestVectorData test_data) throws Exception {
        KeyFactory kf = KeyFactory.getInstance("RSA");

        RSAPrivateKeySpec keySpec = new RSAPrivateKeySpec(test_data.priv_key_modulus,␣
↪test_data.priv_key_exponent);

        return kf.generatePrivate(keySpec);
    }

    public void testDecrypt(byte[] plaintext, byte[] ciphertext) throws Exception {
        System.out.println("Verifying OAEP with mgf1_hash: " + m_mgf1_hash + " alg_hash:
↪" + m_alg_hash + " - "
                + ciphertext.length + " bytes ciphertext - "
                + plaintext.length + " bytes plaintext");

        m_cipher.init(Cipher.DECRYPT_MODE, m_private_key, m_algo_param);
        byte[] java_plaintext = m_cipher.doFinal(ciphertext);

        if (Arrays.equals(java_plaintext, plaintext) == false) {
            throw new Exception("Verification failure - plaintext does not match after␣
↪decryption.");
        }
    }

    public static void main(String[] args) {
        Security.addProvider(new BouncyCastleProvider());

        // assume current directory if no path given on command line
        String vector_path = "./vectors/cryptography_vectors/asymmetric/RSA/oaep-custom";

        if (args.length > 0) {
            vector_path = args[0];
        }

        System.out.println("Vector file path: " + vector_path);

        try {
            // loop over each combination of hash loading the vector file
            // to verify for each
            for (SHAHash mgf1_hash : SHAHash.values()) {
                for (SHAHash alg_hash : SHAHash.values()) {
                    if (mgf1_hash.name().toLowerCase().equals("sha1") &&
                        alg_hash.name().toLowerCase().equals("sha1")) {
                        continue;
                    }
                    String filename = "oaep-" + mgf1_hash.name().toLowerCase() +
                                          "-" + alg_hash.name().toLowerCase() + ".txt";

                    System.out.println("Loading " + filename + "...");
```

(continues on next page)

(continued from previous page)

```
                    TestVectorLoader loader = new TestVectorLoader();
                    loader.open(vector_path + filename);

                    TestVectorData test_data;

                    // load each test in the file and verify
                    while ((test_data = loader.loadNextTest()) != null) {
                        VerifyRSAOAEPSHA2 verify = new VerifyRSAOAEPSHA2(mgf1_hash, alg_
↪hash, test_data);
                        verify.testDecrypt(test_data.plaintext, test_data.ciphertext);
                    }

                    System.out.println("Verifying " + filename + " completed␣
↪successfully.");
                }
            }

            System.out.println("All verification completed successfully");

        } catch (Exception e) {
            // if any exception is thrown the verification has failed
            e.printStackTrace();
            System.out.println("Verification Failed!");
        }
    }
}
```

Download link: `VerifyRSAOAEPSHA2.java`

Using the Verifier

Download and install the Java SDK. Initial verification was performed using `jdk-8u77-macosx-x64.dmg`.

Download the latest Bouncy Castle JAR. Initial verification was performed using `bcprov-jdk15on-154.jar`.

Set the `-classpath` to include the Bouncy Castle jar and the path to `VerifyRSAOAEPSHA2.java` and compile the program.

```
$ javac -classpath ~/Downloads/bcprov-jdk15on-154.jar:./ VerifyRSAOAEPSHA2.java
```

Finally, run the program with the path to the SHA-2 vectors:

```
$ java -classpath ~/Downloads/bcprov-jdk15on-154.jar:./ VerifyRSAOAEPSHA2
```

- `asymmetric/PEM_Serialization/ec_private_key.pem` and `asymmetric/DER_Serialization/ec_private_key.der` - Contains an Elliptic Curve key generated by OpenSSL from the curve `secp256r1`.
- `asymmetric/PEM_Serialization/ec_private_key_encrypted.pem` and `asymmetric/DER_Serialization/ec_private_key_encrypted.der`- Contains the same Elliptic Curve key as `ec_private_key.pem`, except that it is encrypted with AES-128 with the password "123456".
- `asymmetric/PEM_Serialization/ec_public_key.pem` and `asymmetric/DER_Serialization/ec_public_key.der`- Contains the public key corresponding to `ec_private_key.pem`, generated using OpenSSL.

- `asymmetric/PEM_Serialization/ec_public_key_rsa_delimiter.pem` - Contains the public key corresponding to `ec_private_key.pem`, but with the wrong PEM delimiter (`RSA PUBLIC KEY` when it should be `PUBLIC KEY`).
- `asymmetric/PEM_Serialization/rsa_private_key.pem` - Contains an RSA 2048 bit key generated using OpenSSL, protected by the secret "123456" with DES3 encryption.
- `asymmetric/PEM_Serialization/rsa_public_key.pem` and `asymmetric/DER_Serialization/rsa_public_key.der`- Contains an RSA 2048 bit public generated using OpenSSL from `rsa_private_key.pem`.
- `asymmetric/PEM_Serialization/rsa_wrong_delimiter_public_key.pem` - Contains an RSA 2048 bit public key generated from `rsa_private_key.pem`, but with the wrong PEM delimiter (`RSA PUBLIC KEY` when it should be `PUBLIC KEY`).
- `asymmetric/PEM_Serialization/dsa_4096.pem` - Contains a 4096-bit DSA private key generated using OpenSSL.
- `asymmetric/PEM_Serialization/dsaparam.pem` - Contains 2048-bit DSA parameters generated using OpenSSL; contains no keys.
- `asymmetric/PEM_Serialization/dsa_private_key.pem` - Contains a DSA 2048 bit key generated using OpenSSL from the parameters in `dsaparam.pem`, protected by the secret "123456" with DES3 encryption.
- `asymmetric/PEM_Serialization/dsa_public_key.pem` and `asymmetric/DER_Serialization/dsa_public_key.der` - Contains a DSA 2048 bit key generated using OpenSSL from `dsa_private_key.pem`.
- `asymmetric/DER_Serialization/dsa_public_key_no_params.der` - Contains a DSA public key with the optional parameters removed.
- `asymmetric/DER_Serialization/dsa_public_key_invalid_bit_string.der` - Contains a DSA public key with the bit string padding value set to 2 rather than the required 0.
- `asymmetric/PKCS8/unenc-dsa-pkcs8.pem` and `asymmetric/DER_Serialization/unenc-dsa-pkcs8.der` - Contains a DSA 1024 bit key generated using OpenSSL.
- `asymmetric/PKCS8/unenc-dsa-pkcs8.pub.pem` and `asymmetric/DER_Serialization/unenc-dsa-pkcs8.pub.der` - Contains a DSA 2048 bit public key generated using OpenSSL from `unenc-dsa-pkcs8.pem`.
- DER conversions of the GnuTLS example keys for DSA.
- DER conversions of enc-rsa-pkcs8.pem, enc2-rsa-pkcs8.pem, and unenc-rsa-pkcs8.pem.
- `asymmetric/public/PKCS1/rsa.pub.pem` and `asymmetric/public/PKCS1/rsa.pub.der` are PKCS1 conversions of the public key from `asymmetric/PKCS8/unenc-rsa-pkcs8.pem` using PEM and DER encoding.
- `x509/custom/ca/ca_key.pem` - An unencrypted PCKS8 `secp256r1` key. It is the private key for the certificate `x509/custom/ca/ca.pem`.
- `pkcs12/ca/ca_key.pem` - An unencrypted PCKS8 `secp256r1` key. It is the private key for the certificate `pkcs12/ca/ca.pem`. This key is encoded in several of the PKCS12 custom vectors.
- `x509/custom/ca/rsa_key.pem` - An unencrypted PCKS8 4096 bit RSA key. It is the private key for the certificate `x509/custom/ca/rsa_ca.pem`.
- `asymmetric/EC/compressed_points.txt` - Contains compressed public points generated using OpenSSL.
- `asymmetric/EC/explicit_parameters_private_key.pem` - Contains an EC private key with an curve defined by explicit parameters.

- `asymmetric/EC/explicit_parameters_wap_wsg_idm_ecid_wtls11_private_key.pem` - Contains an EC private key with over the `wap-wsg-idm-ecid-wtls11` curve, encoded with explicit parameters.
- `asymmetric/EC/secp256k1-explicit-no-seed.pem` - An unencrypted PKCS8 private key with the `secp256k1` curve explicitly encoded (`secp256k1` does not have a seed).
- `asymmetric/EC/secp256k1-pub-explicit-no-seed.pem` - A public key with the `secp256k1` curve explicitly encoded. This is the public key for the private key `asymmetric/EC/secp256k1-explicit-no-seed.pem`.
- `asymmetric/EC/secp256r1-explicit-no-seed.pem` - An unencrypted PKCS8 private key with the `secp256r1` curve explicitly encoded and with the seed omitted.
- `asymmetric/EC/secp256r1-pub-explicit-no-seed.pem` - A public key with the `secp256r1` curve explicitly encoded and with the seed omitted. This is the public key for the private key `asymmetric/EC/secp256r1-explicit-no-seed.pem`.
- `asymmetric/EC/secp256r1-explicit-seed.pem` - An unencrypted PKCS8 private key with the `secp256r1` curve explicitly encoded.
- `asymmetric/EC/secp256r1-pub-explicit-seed.pem` - A public key with the `secp256r1` curve explicitly encoded. This is the public key for the private key `asymmetric/EC/secp256r1-explicit-seed.pem`.
- `asymmetric/EC/secp384r1-explicit-no-seed.pem` - An unencrypted PKCS8 private key with the `secp384r1` curve explicitly encoded and with the seed omitted.
- `asymmetric/EC/secp384r1-pub-explicit-no-seed.pem` - A public key with the `secp384r1` curve explicitly encoded and with the seed omitted. This is the public key for the private key `asymmetric/EC/secp384r1-explicit-no-seed.pem`.
- `asymmetric/EC/secp384r1-explicit-seed.pem` - An unencrypted PKCS8 private key with the `secp384r1` curve explicitly encoded.
- `asymmetric/EC/secp384r1-pub-explicit-seed.pem` - A public key with the `secp384r1` curve explicitly encoded. This is the public key for the private key `asymmetric/EC/secp384r1-explicit-seed.pem`.
- `asymmetric/EC/secp521r1-explicit-no-seed.pem` - An unencrypted PKCS8 private key with the `secp521r1` curve explicitly encoded and with the seed omitted.
- `asymmetric/EC/secp521r1-pub-explicit-no-seed.pem` - A public key with the `secp521r1` curve explicitly encoded and with the seed omitted. This is the public key for the private key `asymmetric/EC/secp521r1-explicit-no-seed.pem`.
- `asymmetric/EC/secp521r1-explicit-seed.pem` - An unencrypted PKCS8 private key with the `secp521r1` curve explicitly encoded.
- `asymmetric/EC/secp521r1-pub-explicit-seed.pem` - A public key with the `secp521r1` curve explicitly encoded. This is the public key for the private key `asymmetric/EC/secp521r1-explicit-seed.pem`.
- `asymmetric/EC/secp128r1_private_key.pem` - Contains an EC private key on the curve `secp128r1`.
- `asymmetric/EC/sect163k1-spki.pem` - Contains an EC SPKI on the curve `sect163k1`.
- `asymmetric/EC/sect163r2-spki.pem` - Contains an EC SPKI on the curve `sect163r2`.
- `asymmetric/EC/sect233k1-spki.pem` - Contains an EC SPKI on the curve `sect233k1`.
- `asymmetric/EC/sect233r1-spki.pem` - Contains an EC SPKI on the curve `sect233r1`.
- `asymmetric/EC/ec_private_key_non_zero_unused_bits.der` - Contains a `secp256r1` private key whose `publicKey` bit string has the unused-bits count set to 1 rather than the required 0.
- `asymmetric/X448/x448-pkcs8-enc.pem` and `asymmetric/X448/x448-pkcs8-enc.der` contain an X448 key encrypted with AES 256 CBC with the password `password`.

- `asymmetric/X448/x448-pkcs8.pem` and `asymmetric/X448/x448-pkcs8.der` contain an unencrypted X448 key.
- `asymmetric/X448/x448-pub.pem` and `asymmetric/X448/x448-pub.der` contain an X448 public key.
- `asymmetric/Ed25519/ed25519-pkcs8-enc.pem` and `asymmetric/Ed25519/ed25519-pkcs8-enc.der` contain an Ed25519 key encrypted with AES 256 CBC with the password `password`.
- `asymmetric/Ed25519/ed25519-pkcs8.pem` and `asymmetric/Ed25519/ed25519-pkcs8.der` contain an unencrypted Ed25519 key.
- `asymmetric/Ed25519/ed25519-pub.pem` and `asymmetric/Ed25519/ed25519-pub.der` contain an Ed25519 public key.
- `asymmetric/Ed25519/ed25519-pub-non-zero-unused-bits.der` contains an Ed25519 SPKI whose `subjectPublicKey` bit string has the unused-bits count set to 1 rather than the required 0.
- `asymmetric/X25519/x25519-pkcs8-enc.pem` and `asymmetric/X25519/x25519-pkcs8-enc.der` contain an X25519 key encrypted with AES 256 CBC with the password `password`.
- `asymmetric/X25519/x25519-pkcs8.pem` and `asymmetric/X25519/x25519-pkcs8.der` contain an unencrypted X25519 key.
- `asymmetric/X25519/x25519-pub.pem` and `asymmetric/X25519/x25519-pub.der` contain an X25519 public key.
- `asymmetric/Ed448/ed448-pkcs8-enc.pem` and `asymmetric/Ed448/ed448-pkcs8-enc.der` contain an Ed448 key encrypted with AES 256 CBC with the password `password`.
- `asymmetric/Ed448/ed448-pkcs8.pem` and `asymmetric/Ed448/ed448-pkcs8.der` contain an unencrypted Ed448 key.
- `asymmetric/Ed448/ed448-pub.pem` and `asymmetric/Ed448/ed448-pub.der` contain an Ed448 public key.
- `asymmetric/PKCS8/rsa_pss_2048.pem` - A 2048-bit RSA PSS key with no explicit parameters set.
- `asymmetric/PKCS8/rsa_pss_2048_pub.der` - The public key corresponding to `asymmetric/PKCS8/rsa_pss_2048.pem`.
- `asymmetric/PKCS8/rsa_pss_2048_hash.pem` - A 2048-bit RSA PSS key with the hash algorithm PSS parameter set to SHA256.
- `asymmetric/PKCS8/rsa_pss_2048_hash_mask.pem` - A 2048-bit RSA PSS key with with the hash (SHA256) and mask algorithm (SHA256) PSS parameters set.
- `asymmetric/PKCS8/rsa_pss_2048_hash_mask_diff.pem` - A 2048-bit RSA PSS key with the hash (SHA256) and mask algorithm (SHA512) PSS parameters set.
- `asymmetric/PKCS8/rsa_pss_2048_hash_mask_salt.pem` - A 2048-bit RSA PSS key with the hash (SHA256), mask algorithm (SHA256), and salt length (32) PSS parameters set.
- `asymmetric/Traditional_OpenSSL_Serialization/testrsa.pem` - A 2048-bit RSA key, encoded as a "traditional" `RSA PRIVATE KEY` PEM block, rather than a `PRIVATE KEY` block.
- `asymmetric/Traditional_OpenSSL_Serialization/testrsa-encrypted.pem` - The above, encrypted at the PEM level with AES-128-CBC and password "password".
- `asymmetric/Traditional_OpenSSL_Serialization/key1.pem` - The above, encrypted at the PEM level with DES-EDE3-CBC and password "123456".
- `asymmetric/Traditional_OpenSSL_Serialization/key2.pem` - The above, encrypted at the PEM level with AES-128-CBC and password "a123456".
- `asymmetric/DER_Serialization/testrsa.der` - The above as a DER-encoded RSAPrivateKey structure.

- `asymmetric/DSA/custom/nilpotent.pem` – A key where the field is actually a ring and the generator of the multiplicative subgroup is actually nilpotent with low degree. Taken from BoringSSL (see `TEST(DSATest, NilpotentGenerator)`).
- `asymmetric/PKCS8/ec-invalid-private-scalar.pem` - Contains a PKCS8 encoded PEM with a `secp256r1` OID and an invalid (very large) private scalar.
- `asymmetric/PKCS8/invalid-version.der` - Contains a PKCS8 encoded DER with an invalid version field.
- `asymmetric/PKCS8/unknown-oid.der` - Contains a PKCS8 encoded DER with an unknown OID.
- `asymmetric/Traditional_OpenSSL_Serialization/rsa-wrong-version.pem` - An RSA key, encoded as a "traditional" `RSA PRIVATE KEY` PEM block, with an invalid version number.
- `asymmetric/Traditional_OpenSSL_Serialization/dsa-wrong-version.pem` - A DSA key, encoded as a "traditional" `DSA PRIVATE KEY` PEM block, with an invalid version number.
- `asymmetric/PKCS8/ec-inconsistent-curve.pem` - A PKCS8 encoded EC key where the the curve OID in the parameters does not match the curve OID in the key.
- `asymmetric/PKCS8/ec-inconsistent-curve2.pem` - A PKCS8 encoded EC key where the the curve OID in the parameters does not match the curve OID in the key (the OIDs are reversed from `ec-inconsistent-curve.pem`).
- `asymmetric/EC/ec-missing-curve.pem` - A PKCS#1 encoded EC key where the curve OID is missing.
- `asymmetric/PKCS8/ec-consistent-curve.pem` - A PKCS8 encoded EC key where the the curve OID in the parameters is the same as the curve OID in the key (encoding the curve OID twice is duplicative, as the inner curve is optional).
- `asymmetric/PKCS8/ec-invalid-version.pem` - A PKCS8 encoded EC key with an invalid elliptic curve version field.
- `asymmetric/PKCS8/enc-rsa-3des.pem` - A PKCS8 encoded RSA key encrypted with 3DES, with the password "password".
- `asymmetric/PKCS8/enc-unknown-algorithm.pem` - A PKCS8 encoded key with an unknown encryption algorithm.
- `asymmetric/PKCS8/enc-unknown-pbkdf2-prf.pem` - A PKCS8 encoded key encrypted using PBKDF2 with an unknown PRF.
- `asymmetric/PKCS8/enc-unknown-kdf.pem` - A PKCS8 encoded key encrypted using an unknown KDF.
- `asymmetric/Traditional_OpenSSL_Serialization/key1-no-dek-info.pem` - An RSA key in an encrypted PEM with no `DEK-Info` header.
- `asymmetric/Traditional_OpenSSL_Serialization/key1-malformed-dek-info.pem` - An RSA key in an encrypted PEM with a malformed `DEK-Info` header (no comma).
- `asymmetric/Traditional_OpenSSL_Serialization/key1-malformed-iv.pem` - An RSA key in an encrypted PEM with a malformed IV (not valid hex).
- `asymmetric/Traditional_OpenSSL_Serialization/key1-short-iv.pem` - An RSA key in an encrypted PEM with an IV that's too short (less than 8 bytes).
- `asymmetric/PKCS8/rsa-pbewithmd5anddescbc.pem` - A PKCS8 encoded RSA key encrypted using the `pbeWithMD5AndDES-CBC` algorithm with the password `hunter2`.
- `asymmetric/PKCS8/rsa-pbe-3des-long-salt.pem` - A PKCS8 encoded RSA key encrypted with a 20 byte salt with the password `password`.
- `asymmetric/PKCS8/wrong-pem-delimiter-rsa.pem` - A PKCS8 encoded RSA key with the wrong PEM delimiter.

- `asymmetric/EC/high-bit-set.pem` - A PKCS#1 encoded EC key where the private key value has its high bit set.
- `asymmetric/EC/truncated-private-key.der` - A PKCS#1 encoded EC key where the private key value is not encoded to the order's length.
- `asymmetric/PKCS8/rsa-rc2-cbc-effective-key-length.pem` a PKCS8 encoded key encrypted with `RC2-CBC` with the `effectiveKeyLength` parameter set to 258. This is an invalid key.
- `asymmetric/PKCS8/enc-ec-sha1-128-rc4.pem` a PKCS8 encoded ECDSA P-256 key encrypted with `pbeWithSHA1And128BitRC4`. The password is `password`.
- `asymmetric/PKCS8/enc-rsa-pkcs8-pbkdf2-0iter.pem` a PKCS8 encoded structure with an invalid PBKDF2 iteration count (zero).
- `asymmetric/MLDSA/mldsa44_pub.der` and `asymmetric/MLDSA/mldsa44_priv.der` a PKCS8 encoded key pair for ML-DSA-44 generated by OpenSSL.
- `asymmetric/MLDSA/mldsa65_noseed_priv.der` a PKCS8 encoded ML-DSA-65 private key serialized without the seed.
- `asymmetric/MLKEM/mlkem768.der` and `asymmetric/MLKEM/mlkem768.pem`, PKCS8 DER/PEM encoded MLKEM-768 private key using the seed of the first test vector of `kat_MLKEM_768.rsp`.

Key exchange

- `vectors/cryptography_vectors/asymmetric/DH/rfc3526.txt` contains several standardized Diffie-Hellman groups from **RFC 3526**.
- `vectors/cryptography_vectors/asymmetric/DH/RFC5114.txt` contains Diffie-Hellman examples from appendix A.1, A.2 and A.3 of **RFC 5114**.
- `vectors/cryptography_vectors/asymmetric/DH/vec.txt` contains Diffie-Hellman examples from botan.
- `vectors/cryptography_vectors/asymmetric/DH/bad_exchange.txt` contains Diffie-Hellman vector pairs that were generated using OpenSSL `DH_generate_parameters_ex` and `DH_generate_key`.
- `vectors/cryptography_vectors/asymmetric/DH/dhp.pem`, `vectors/cryptography_vectors/asymmetric/DH/dhkey.pem` and `vectors/cryptography_vectors/asymmetric/DH/dhpub.pem` contains Diffie-Hellman parameters and key respectively. The keys were generated using OpenSSL following DHKE guide. `vectors/cryptography_vectors/asymmetric/DH/dhkey.txt` contains all parameter in text. `vectors/cryptography_vectors/asymmetric/DH/dhp.der`, `vectors/cryptography_vectors/asymmetric/DH/dhkey.der` and `vectors/cryptography_vectors/asymmetric/DH/dhpub.der` contains are the above parameters and keys in DER format.
- `vectors/cryptography_vectors/asymmetric/DH/dhp_rfc5114_2.pem`, `vectors/cryptography_vectors/asymmetric/DH/dhkey_rfc5114_2.pem` and `vectors/cryptography_vectors/asymmetric/DH/dhpub_rfc5114_2.pem` contains Diffie-Hellman parameters and key respectively. The keys were generated using OpenSSL following DHKE guide. When creating the parameters we added the *-pkeyopt dh_rfc5114:2* option to use **RFC 5114** 2048 bit DH parameters with 224 bit subgroup. `vectors/cryptography_vectors/asymmetric/DH/dhkey_rfc5114_2.txt` contains all parameter in text. `vectors/cryptography_vectors/asymmetric/DH/dhp_rfc5114_2.der`, `vectors/cryptography_vectors/asymmetric/DH/dhkey_rfc5114_2.der` and `vectors/cryptography_vectors/asymmetric/DH/dhpub_rfc5114_2.der` contains are the above parameters and keys in DER format.
- `vectors/cryptography_vectors/asymmetric/DH/dh_key_256.pem` contains a PEM PKCS8 encoded DH key with a 256-bit key size.

- `vectors/cryptography_vectors/asymmetric/DH/dhp_privatevaluelength.pem` contains PKCS#3 `DH PARAMETERS` that include the optional `privateValueLength` field, which must not be confused with an X9.42 subprime `q`.
- `vectors/cryptoraphy_vectors/asymmetric/ECDH/brainpool.txt` contains Brainpool vectors from **RFC 7027**.
- `vectors/cryptography_vectors/asymmetric/DH/dhpub_cryptography_old.pem` contains a Diffie-Hellman public key generated with a previous version of `cryptography`.

X.509

- PKITS test suite from NIST PKI Testing.
- `v1_cert.pem` from the OpenSSL source tree (testx509.pem).
- `ecdsa_root.pem` - DigiCert Global Root G3, a `secp384r1` ECDSA root certificate.
- `verisign-md2-root.pem` - A legacy Verisign public root signed using the MD2 algorithm.
- `cryptography.io.pem` - A leaf certificate issued by RapidSSL for the cryptography website.
- `cryptography.io.old_header.pem` - A leaf certificate issued by RapidSSL for the cryptography website. This certificate uses the `X509 CERTIFICATE` legacy PEM header format.
- `cryptography.io.chain.pem` - The same as `cryptography.io.pem`, but `rapidssl_sha256_ca_g3.pem` is concatenated to the end.
- `cryptography.io.with_headers.pem` - The same as `cryptography.io.pem`, but with an unrelated (encrypted) private key concatenated to the end.
- `cryptography.io.chain_with_garbage.pem` - The same as `cryptography.io.chain.pem`, but with other sections and text around it.
- `cryptography.io.with_garbage.pem` - The same as `cryptography.io.pem`, but with other sections and text around it.
- `rapidssl_sha256_ca_g3.pem` - The intermediate CA that issued the `cryptography.io.pem` certificate.
- `cryptography.io.precert.pem` - A pre-certificate with the CT poison extension for the cryptography website.
- `cryptography-scts.pem` - A leaf certificate issued by Let's Encrypt for the cryptography website which contains signed certificate timestamps.
- `wildcard_san.pem` - A leaf certificate issued by a public CA for `langui.sh` that contains wildcard entries in the SAN extension.
- `san_edipartyname.der` - A DSA certificate from a Mozilla bug containing a SAN extension with an `ediPartyName` general name.
- `san_x400address.der` - A DSA certificate from a Mozilla bug containing a SAN extension with an `x400Address` general name.
- `department-of-state-root.pem` - The intermediary CA for the Department of State, issued by the United States Federal Government's Common Policy CA. Notably has a `critical` policy constraints extensions.
- `e-trust.ru.der` - A certificate from a Russian CA signed using the GOST cipher and containing numerous unusual encodings such as NUMERICSTRING in the subject DN.
- `alternate-rsa-sha1-oid.der` - A certificate that uses an alternate signature OID for RSA with SHA1. This certificate has an invalid signature.
- `badssl-sct.pem` - A certificate with the certificate transparency signed certificate timestamp extension.

- `badssl-sct-none-hash.der` - The same as `badssl-sct.pem`, but DER-encoded and with the SCT's signature hash manually changed to "none" (`0x00`).
- `badssl-sct-anonymous-sig.der` - The same as `badssl-sct.pem`, but DER-encoded and with the SCT's signature algorithm manually changed to "anonymous" (`0x00`).
- `bigoid.pem` - A certificate with a rather long OID in the Certificate Policies extension. We need to make sure we can parse long OIDs.
- `wosign-bc-invalid.pem` - A certificate issued by WoSign that contains a basic constraints extension with CA set to false and a path length of zero in violation of **RFC 5280**.
- `tls-feature-ocsp-staple.pem` - A certificate issued by Let's Encrypt that contains a TLS Feature extension with the `status_request` feature (commonly known as OCSP Must-Staple).
- `unique-identifier.pem` - A certificate containing a distinguished name with an `x500UniqueIdentifier`.
- `utf8-dnsname.pem` - A certificate containing non-ASCII characters in the DNS name entries of the SAN extension.
- `badasn1time.pem` - A certificate containing an incorrectly specified UTCTime in its validity->not_after.
- `letsencryptx3.pem` - A subordinate certificate used by Let's Encrypt to issue end entity certificates.
- `ed25519-rfc8410.pem` - A certificate containing an X25519 public key with an `ed25519` signature taken from **RFC 8410**.
- `root-ed25519.pem` - An `ed25519` root certificate (`ed25519` signature with `ed25519` public key) from the OpenSSL test suite. (root-ed25519.pem)
- `server-ed25519-cert.pem` - An `ed25519` server certificate (RSA signature with `ed25519` public key) from the OpenSSL test suite. (server-ed25519-cert.pem)
- `server-ed448-cert.pem` - An `ed448` server certificate (RSA signature with `ed448` public key) from the OpenSSL test suite. (server-ed448-cert.pem)
- `accvraiz1.pem` - An RSA root certificate that contains an `explicitText` entry with a `BMPString` type.
- `scottishpower-bitstring-dn.pem` - An ECDSA certificate that contains a subject DN with a bit string type.
- `cryptography-scts-tbs-precert.der` - The "to-be-signed" pre-certificate bytes from `cryptography-scts.pem`, with the SCT list extension removed.
- `belgian-eid-invalid-visiblestring.pem` - A certificate with UTF-8 bytes in a `VisibleString` type.
- `ee-pss-sha1-cert.pem` - An RSA PSS certificate using a SHA1 signature and SHA1 for MGF1 from the OpenSSL test suite.

Custom X.509 Vectors

- `invalid_version.pem` - Contains an RSA 2048 bit certificate with the X.509 version field set to `0x7`.
- `post2000utctime.pem` - Contains an RSA 2048 bit certificate with the `notBefore` and `notAfter` fields encoded as post-2000 `UTCTime`.
- `dsa_selfsigned_ca.pem` - Contains a DSA self-signed CA certificate generated using OpenSSL.
- `ec_no_named_curve.pem` - Contains an ECDSA certificate that does not have an embedded OID defining the curve.
- `all_supported_names.pem` - An RSA 2048 bit certificate generated using OpenSSL that contains a subject and issuer that have two of each supported attribute type from **RFC 5280**.
- `unsupported_subject_name.pem` - An RSA 2048 bit self-signed CA certificate generated using OpenSSL that contains the unsupported "initials" name.

- `utf8_common_name.pem` - An RSA 2048 bit self-signed CA certificate generated using OpenSSL that contains a UTF8String common name with the value "We heart UTF8!™".
- `invalid_utf8_common_name.pem` - A certificate that contains a `UTF8String` common name with an invalid UTF-8 byte sequence.
- `name_attribute_unsupported_tag.pem` - A certificate whose subject and issuer common name values carry an ASN.1 tag (`0x69`) that is not one of the recognized string types. The signature on this certificate is invalid.
- `two_basic_constraints.pem` - An RSA 2048 bit self-signed certificate containing two basic constraints extensions.
- `basic_constraints_not_critical.pem` - An RSA 2048 bit self-signed certificate containing a basic constraints extension that is not marked as critical.
- `bc_path_length_zero.pem` - An RSA 2048 bit self-signed certificate containing a basic constraints extension with a path length of zero.
- `unsupported_extension.pem` - An RSA 2048 bit self-signed certificate containing an unsupported extension type. The OID was encoded as "1.2.3.4" with an `extnValue` of "value".
- `unsupported_extension_2.pem` - A `secp256r1` certificate containing two unsupported extensions. The OIDs are `1.3.6.1.4.1.41482.2` with an `extnValue` of `1.3.6.1.4.1.41482.1.2` and `1.3.6.1.4.1.45724.2.1.1` with an `extnValue` of `\x03\x02\x040`
- `unsupported_extension_critical.pem` - An RSA 2048 bit self-signed certificate containing an unsupported extension type marked critical. The OID was encoded as "1.2.3.4" with an `extnValue` of "value".
- `san_email_dns_ip_dirname_uri.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with the following general names: `rfc822Name`, `dNSName`, `iPAddress`, `directoryName`, and `uniformResourceIdentifier`.
- `san_empty_hostname.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative extension with an empty `dNSName` general name.
- `san_other_name.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with the `otherName` general name.
- `san_registered_id.pem` - An RSA 1024 bit certificate containing a subject alternative name extension with the `registeredID` general name.
- `all_key_usages.pem` - An RSA 2048 bit self-signed certificate containing a key usage extension with all nine purposes set to true.
- `extended_key_usage.pem` - An RSA 2048 bit self-signed certificate containing an extended key usage extension with eight usages.
- `san_idna_names.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with `rfc822Name`, `dNSName`, and `uniformResourceIdentifier` general names with IDNA (**RFC 5895**) encoding.
- `san_wildcard_idna.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with a `dNSName` general name with a wildcard IDNA (**RFC 5895**) domain.
- `san_idna2003_dnsname.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with an IDNA 2003 (**RFC 3490**) `dNSName`.
- `san_rfc822_names.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with various `rfc822Name` values.
- `san_rfc822_idna.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with an IDNA `rfc822Name`.

- `san_uri_with_port.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with various `uniformResourceIdentifier` values.
- `san_ipaddr.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with an `iPAddress` value.
- `san_dirname.pem` - An RSA 2048 bit self-signed certificate containing a subject alternative name extension with a `directoryName` value.
- `inhibit_any_policy_5.pem` - An RSA 2048 bit self-signed certificate containing an inhibit any policy extension with the value 5.
- `inhibit_any_policy_negative.pem` - An RSA 2048 bit self-signed certificate containing an inhibit any policy extension with the value -1.
- `authority_key_identifier.pem` - An RSA 2048 bit self-signed certificate containing an authority key identifier extension with key identifier, authority certificate issuer, and authority certificate serial number fields.
- `authority_key_identifier_no_keyid.pem` - An RSA 2048 bit self-signed certificate containing an authority key identifier extension with authority certificate issuer and authority certificate serial number fields.
- `aia_ocsp_ca_issuers.pem` - An RSA 2048 bit self-signed certificate containing an authority information access extension with two OCSP and one CA issuers entry.
- `aia_ocsp.pem` - An RSA 2048 bit self-signed certificate containing an authority information access extension with an OCSP entry.
- `aia_ca_issuers.pem` - An RSA 2048 bit self-signed certificate containing an authority information access extension with a CA issuers entry.
- `cdp_empty_hostname.pem` - An RSA 2048 bit self-signed certificate containing a CRL distribution point extension with `fullName` URI without a hostname.
- `cdp_fullname_reasons_crl_issuer.pem` - An RSA 1024 bit certificate containing a CRL distribution points extension with `fullName`, `cRLIssuer`, and `reasons` data.
- `cdp_crl_issuer.pem` - An RSA 1024 bit certificate containing a CRL distribution points extension with `cRLIssuer` data.
- `cdp_all_reasons.pem` - An RSA 1024 bit certificate containing a CRL distribution points extension with all `reasons` bits set.
- `cdp_reason_aa_compromise.pem` - An RSA 1024 bit certificate containing a CRL distribution points extension with the `AACompromise` `reasons` bit set.
- `nc_permitted_excluded.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with both permitted and excluded elements. Contains `IPv4` and `IPv6` addresses with network mask as well as `dNSName` with a leading period.
- `nc_permitted_excluded_2.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with both permitted and excluded elements. Unlike `nc_permitted_excluded.pem`, the general names do not contain any name constraints specific values.
- `nc_permitted.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with permitted elements.
- `nc_permitted_2.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with permitted elements that do not contain any name constraints specific values.
- `nc_excluded.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with excluded elements.
- `nc_invalid_ip_netmask.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with a permitted element that has an `IPv6` IP and an invalid network mask.

- `nc_invalid_ip4_netmask.der` - An RSA 2048 bit self-signed certificate containing a name constraints extension with a permitted element that has an `IPv4` IP and an invalid network mask. The signature on this certificate is invalid.
- `nc_single_ip_netmask.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with a permitted element that has two IPs with `/32` and `/128` network masks.
- `nc_ip_invalid_length.pem` - An RSA 2048 bit self-signed certificate containing a name constraints extension with a permitted element that has an invalid length (33 bytes instead of 32) for an `IPv6` address with network mask. The signature on this certificate is invalid.
- `cp_user_notice_with_notice_reference.pem` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with a notice reference in the user notice.
- `cp_user_notice_with_explicit_text.pem` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with explicit text and no notice reference.
- `cp_cps_uri.pem` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with a CPS URI and no user notice.
- `cp_user_notice_no_explicit_text.pem` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with a user notice with no explicit text.
- `cp_invalid.pem` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with invalid data. The `policyQualifierId` is for `id-qt-unotice` but the value is an `id-qt-cps` ASN.1 structure.
- `cp_invalid2.der` - An RSA 2048 bit self-signed certificate containing a certificate policies extension with invalid data. The `policyQualifierId` is for `id-qt-cps` but the value is an `id-qt-unotice` ASN.1 structure. The signature on this certificate is invalid.
- `ian_uri.pem` - An RSA 2048 bit certificate containing an issuer alternative name extension with a `URI` general name.
- `ocsp_nocheck.pem` - An RSA 2048 bit self-signed certificate containing an `OCSPNoCheck` extension.
- `pc_inhibit_require.pem` - An RSA 2048 bit self-signed certificate containing a policy constraints extension with both inhibit policy mapping and require explicit policy elements.
- `pc_inhibit.pem` - An RSA 2048 bit self-signed certificate containing a policy constraints extension with an inhibit policy mapping element.
- `pc_require.pem` - An RSA 2048 bit self-signed certificate containing a policy constraints extension with a require explicit policy element.
- `unsupported_subject_public_key_info.pem` - A certificate whose public key is an unknown OID (`1.3.6.1.4.1.8432.1.1.2`).
- `policy_constraints_explicit.pem` - A self-signed certificate containing a `policyConstraints` extension with a `requireExplicitPolicy` value.
- `freshestcrl.pem` - A self-signed certificate containing a `freshestCRL` extension.
- `sia.pem` - An RSA 2048 bit self-signed certificate containing a subject information access extension with both a CA repository entry and a custom OID entry.
- `ca/ca.pem` - A self-signed certificate with `basicConstraints` set to true. Its private key is `ca/ca_key.pem`.
- `pkcs12/ca/ca.pem` - A self-signed certificate with `basicConstraints` set to true. Its private key is `pkcs12/ca/ca_key.pem`. This key is encoded in several of the PKCS12 custom vectors.
- `negative_serial.pem` - A certificate with a serial number that is a negative number.
- `rsa_pss.pem` - A certificate with an RSA PSS signature.
- `root-ed448.pem` - An `ed448` self-signed CA certificate using `ed448-pkcs8.pem` as key.

- `ca/rsa_ca.pem` - A self-signed RSA certificate with `basicConstraints` set to true. Its private key is `ca/rsa_key.pem`.
- `ca/rsae_ca.pem` - A self-signed RSA certificate using a (non-PSS) RSA public key and a RSA PSS signature. Its private key is `ca/rsa_key.pem`.
- `invalid-sct-version.der` - A certificate with an SCT with an unknown version.
- `invalid-sct-length.der` - A certificate with an SCT with an internal length greater than the amount of data.
- `bad_country.pem` - A certificate with country name and jurisdiction country name values in its subject and issuer distinguished names which are longer than 2 characters.
- `rsa_pss_cert.pem` - A self-signed certificate with an RSA PSS signature with `asymmetric/PKCS8/rsa_pss_2048.pem` as its key.
- `rsa_pss_cert_invalid_mgf.der` - A self-signed certificate with an invalid RSA PSS signature that has a non-MGF1 OID for its mask generation function in the signature algorithm.
- `rsa_pss_cert_no_sig_params.der` - A self-signed certificate with an invalid RSA PSS signature algorithm that is missing signature parameters for PSS.
- `rsa_pss_cert_unsupported_mgf_hash.der` - A self-signed certificate with an unsupported MGF1 hash algorithm in the signature algorithm.
- `long-form-name-attribute.pem` - A certificate with `subject` and `issuer` names containing attributes whose value's tag is encoded in long-form.
- `mismatch_inner_outer_sig_algorithm.der` - A leaf certificate derived from `x509/cryptography.io.pem` but modifying the `tbs_cert.signature_algorithm` OID to not match the outer signature algorithm OID.
- `ms-certificate-template.pem` - A certificate with a `msCertificateTemplate` extension.
- `rsa_pss_sha256_no_null.pem` - A certificate with an RSA PSS signature with no encoded `NULL` for the PSS hash algorithm parameters. This certificate was generated by LibreSSL.
- `ecdsa_null_alg.pem` - A certificate with an ECDSA signature with `NULL` algorithm parameters. This encoding is invalid, but was generated by Java 11.
- `dsa_null_alg_params.pem` - A certificate with a DSA signature with `NULL` algorithm parameters. This encoding is invalid, but was generated by Java 20.
- `ekucrit-testuser-cert.pem` - A leaf certificate containing a critical EKU. This is an invalid certificate per CA/B 7.1.2.7.6.
- `empty-eku.pem` - A leaf certificate containing an empty EKU extension. This is an invalid certificate per **RFC 5280** 4.2.1.12.
- `malformed-san.pem` - A certificate with a malformed SAN.
- `malformed-ian.pem` - A certificate with a malformed IAN.
- `admissions_extension_optional_data_not_provided.pem` - A certificate containing the `Admissions` extension with multiple admissions, signed by `x509/custom/ca/rsa_ca.pem` CA. The admissions in this certificate are prepared using synthetic data to verify the possible corner cases are handled by the parser correctly (an admission missing naming authority or admission authority, a profession info missing naming authority or profession OIDs or the registration number etc).
- `admissions_extension_authority_not_provided.pem` - A certificate containing the `Admissions` extension with no admissions and no admission authority, signed by `x509/custom/ca/rsa_ca.pem` CA.
- `no_sans.pem` - Leaf certificate issued by `x509/custom/ca/rsa_ca.pem` with no SAN extension.

- `private_key_usage_period_both_dates.pem` - A certificate containing PrivateKeyUsagePeriod with both `notBefore` and `notAfter` fields set.
- `private_key_usage_period_only_not_before.pem` - A certificate containing PrivateKeyUsagePeriod with only `notBefore` field set.
- `private_key_usage_period_only_not_after.pem` - A certificate containing PrivateKeyUsagePeriod with only `notAfter` field set.

Custom X.509 Request Vectors

- `dsa_sha1.pem` and `dsa_sha1.der` - Contain a certificate request using 1024-bit DSA parameters and SHA1 generated using OpenSSL.
- `rsa_md4.pem` and `rsa_md4.der` - Contain a certificate request using 2048 bit RSA and MD4 generated using OpenSSL.
- `rsa_sha1.pem` and `rsa_sha1.der` - Contain a certificate request using 2048 bit RSA and SHA1 generated using OpenSSL.
- `rsa_sha256.pem` and `rsa_sha256.der` - Contain a certificate request using 2048 bit RSA and SHA256 generated using OpenSSL.
- `ec_sha256.pem` and `ec_sha256.der` - Contain a certificate request using EC (`secp384r1`) and SHA256 generated using OpenSSL.
- `ec_sha256_old_header.pem` - Identical to `ec_sha256.pem`, but uses the `-----BEGIN NEW CERTIFICATE REQUEST-----` legacy PEM header format.
- `san_rsa_sha1.pem` and `san_rsa_sha1.der` - Contain a certificate request using RSA and SHA1 with a subject alternative name extension generated using OpenSSL.
- `two_basic_constraints.pem` - A certificate signing request for an RSA 2048 bit key containing two basic constraints extensions. The signature on this CSR is invalid.
- `unsupported_extension.pem` - A certificate signing request for an RSA 2048 bit key containing containing an unsupported extension type. The OID was encoded as "1.2.3.4" with an `extnValue` of "value". The signature on this CSR is invalid.
- `unsupported_extension_critical.pem` - A certificate signing request for an RSA 2048 bit key containing containing an unsupported extension type marked critical. The OID was encoded as "1.2.3.4" with an `extnValue` of "value". The signature on this CSR is invalid.
- `basic_constraints.pem` - A certificate signing request for an RSA 2048 bit key containing a basic constraints extension marked as critical. The signature on this CSR is invalid.
- `invalid_signature.pem` - A certificate signing request for an RSA 1024 bit key containing an invalid signature with correct padding.
- `challenge.pem` - A certificate signing request for an RSA 2048 bit key containing a challenge password.
- `challenge-invalid.der` - A certificate signing request for an RSA 2048 bit key containing a challenge password attribute that has been encoded as an ASN.1 integer rather than a string.
- `challenge-unstructured.pem` - A certificate signing request for an RSA 2048 bit key containing a challenge password attribute and an unstructured name attribute.
- `challenge-multi-valued.der` - A certificate signing request for an RSA 2048 bit key containing a challenge password attribute with two values inside the ASN.1 set. The signature on this request is invalid.
- `freeipa-bad-critical.pem` - A certificate signing request where the extensions value has a `critical` value of `False` explicitly encoded.

- `bad-version.pem` - A certificate signing request where the version is invalid.
- `long-form-attribute.pem` - A certificate signing request containing an attribute whose value's tag is encoded in the long form.
- `zero-element-attribute.pem` - A certificate signing request containing an attribute whose value has zero elements.

Custom X.509 Certificate Revocation List Vectors

- `crl_all_reasons.pem` - Contains a CRL with 12 revoked certificates, whose serials match their list position. It includes one revocation without any entry extensions, 10 revocations with every supported reason code and one revocation with an unsupported, non-critical entry extension with the OID value set to "1.2.3.4". The signature on this CRL is invalid.
- `crl_dup_entry_ext.pem` - Contains a CRL with one revocation which has a duplicate entry extension. The signature on this CRL is invalid.
- `crl_md2_unknown_crit_entry_ext.pem` - Contains a CRL with one revocation which contains an unsupported critical entry extension with the OID value set to "1.2.3.4". The CRL uses an unsupported MD2 signature algorithm, and the signature on this CRL is invalid.
- `crl_unsupported_reason.pem` - Contains a CRL with one revocation which has an unsupported reason code. The signature on this CRL is invalid.
- `crl_inval_cert_issuer_entry_ext.pem` - Contains a CRL with one revocation which has one entry extension for certificate issuer with an empty value. The signature on this CRL is invalid.
- `crl_empty.pem` - Contains a CRL with no revoked certificates.
- `crl_empty_no_sequence.der` - Contains a CRL with no revoked certificates and the optional ASN.1 sequence for revoked certificates is omitted.
- `crl_ian_aia_aki.pem` - Contains a CRL with `IssuerAlternativeName`, `AuthorityInformationAccess`, `AuthorityKeyIdentifier` and `CRLNumber` extensions.
- `valid_signature_crl.pem` - Contains a CRL with a valid signature.
- `valid_signature_cert.pem` - Contains a cert whose public key corresponds to the private key that produced the signature for `valid_signature_crl.pem`.
- `invalid_signature_crl.pem` - Contains a CRL with the last signature byte incremented by 1 to produce an invalid signature.
- `invalid_signature_cert.pem` - Contains a cert whose public key corresponds to the private key that produced the signature for `invalid_signature_crl.pem`.
- `crl_delta_crl_indicator.pem` - Contains a CRL with the `DeltaCRLIndicator` extension.
- `crl_idp_fullname_only.pem` - Contains a CRL with an `IssuingDistributionPoints` extension with only a `fullname` for the distribution point.
- `crl_idp_only_ca.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that is only valid for CA certificate revocation.
- `crl_idp_fullname_only_aa.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that sets a `fullname` and is only valid for attribute certificate revocation.
- `crl_idp_fullname_only_user.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that sets a `fullname` and is only valid for user certificate revocation.
- `crl_idp_fullname_indirect_crl.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that sets a `fullname` and the indirect CRL flag.

- `crl_idp_reasons_only.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that is only valid for revocations with the `keyCompromise` reason.
- `crl_idp_relative_user_all_reasons.pem` - Contains a CRL with an `IssuingDistributionPoints` extension that sets all revocation reasons as allowed.
- `crl_idp_relativename_only.pem` - Contains a CRL with an `IssuingDistributionPoints` extension with only a `relativename` for the distribution point.
- `crl_unrecognized_extension.der` - Contains a CRL containing an unsupported extension type. The OID was encoded as "1.2.3.4.5" with an `extnValue` of `abcdef`.
- `crl_invalid_time.der` - Contains a CRL with an invalid `UTCTime` value in `thisUpdate`. The signature on this CRL is invalid.
- `crl_no_next_time.pem` - Contains a CRL with no `nextUpdate` value. The signature on this CRL is invalid.
- `crl_bad_version.pem` - Contains a CRL with an invalid version.
- `crl_almost_10k.pem` - Contains a CRL with 9,999 entries.
- `crl_inner_outer_mismatch.der` - A CRL created from `valid_signature_crl.pem` but with a mismatched inner and outer signature algorithm. The signature on this CRL is invalid.
- `crl_issuer_invalid_printable_string.der` - A CRL where the `issuer` field contains an invalid `PRINTABLE STRING` value.
- `crl_inval_date_fractional_seconds.der` - A CRL with one revocation whose `InvalidityDate` entry extension carries a `GeneralizedTime` with fractional seconds, which is not valid DER.

X.509 OCSP Test Vectors

- `x509/ocsp/resp-sha256.der` - An OCSP response for `cryptography.io` with a SHA256 signature.
- `x509/ocsp/resp-unauthorized.der` - An OCSP response with an unauthorized status.
- `x509/ocsp/resp-revoked.der` - An OCSP response for `revoked.badssl.com` with a revoked status.
- `x509/ocsp/resp-delegate-unknown-cert.der` - An OCSP response for an unknown cert from `AC Camerafirma`. This response also contains a delegate certificate.
- `x509/ocsp/resp-responder-key-hash.der` - An OCSP response from the `DigiCert` OCSP responder that uses a key hash for the responder ID.
- `x509/ocsp/resp-revoked-reason.der` - An OCSP response from the `QuoVadis` OCSP responder that contains a revoked certificate with a revocation reason.
- `x509/ocsp/resp-revoked-no-next-update.der` - An OCSP response that contains a revoked certificate and no `nextUpdate` value.
- `x509/ocsp/resp-invalid-signature-oid.der` - An OCSP response that was modified to contain an MD2 signature algorithm object identifier.
- `x509/ocsp/resp-single-extension-reason.der` - An OCSP response that contains a `CRLReason` single extension.
- `x509/ocsp/resp-sct-extension.der` - An OCSP response containing a `CT Certificate SCTs` single extension, from the SwissSign OCSP responder.
- `x509/ocsp/ocsp-army.deps.mil-resp.der` - An OCSP response containing multiple `SINGLERESP` values.
- `x509/ocsp/resp-response-type-unknown-oid.der` - An OCSP response with an unknown OID for response type. The signature on this response is invalid.

- `x509/ocsp/resp-successful-no-response-bytes.der` - An OCSP request with a successful response type but the response bytes are missing.
- `x509/ocsp/resp-unknown-response-status.der` - An OCSP response with an unknown response status.

Custom X.509 OCSP Test Vectors

- `x509/ocsp/req-sha1.der` - An OCSP request containing a single request and using SHA1 as the hash algorithm.
- `x509/ocsp/req-multi-sha1.der` - An OCSP request containing multiple requests.
- `x509/ocsp/req-invalid-hash-alg.der` - An OCSP request containing an invalid hash algorithm OID.
- `x509/ocsp/req-ext-nonce.der` - An OCSP request containing a nonce extension.
- `x509/ocsp/req-ext-unknown-oid.der` - An OCSP request containing an extension with an unknown OID.
- `x509/ocsp/req-duplicate-ext.der` - An OCSP request with duplicate extensions.
- `x509/ocsp/resp-unknown-extension.der` - An OCSP response containing an extension with an unknown OID.
- `x509/ocsp/resp-unknown-hash-alg.der` - An OCSP response containing an invalid hash algorithm OID.
- `x509/ocsp/req-acceptable-responses.der` - An OCSP request containing an acceptable responses extension.
- `x509/ocsp/req-invalid-version.der` - `req-sha1.der` modified so the `tbsRequest` carries an explicit version of 1, which is not defined by RFC 6960.
- `x509/ocsp/resp-invalid-version.der` - `resp-sha256.der` modified so the `tbsResponseData` carries an explicit version of 1, which is not defined by RFC 6960.

Custom PKCS12 Test Vectors

- `pkcs12/cert-key-aes256cbc.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) and key (`pkcs12/ca/ca_key.pem`) both encrypted with AES 256 CBC with the password `cryptography`.
- `pkcs12/cert-none-key-none.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) and key (`pkcs12/ca/ca_key.pem`) with no encryption. The password (used for integrity checking only) is `cryptography`.
- `pkcs12/cert-rc2-key-3des.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) encrypted with RC2 and key (`pkcs12/ca/ca_key.pem`) encrypted via 3DES with the password `cryptography`.
- `pkcs12/no-password.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) and key (`pkcs12/ca/ca_key.pem`) with no encryption and no password.
- `pkcs12/no-cert-key-aes256cbc.p12` - A PKCS12 file containing a key (`pkcs12/ca/ca_key.pem`) encrypted via AES 256 CBC with the password `cryptography` and no certificate.
- `pkcs12/cert-aes256cbc-no-key.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) encrypted via AES 256 CBC with the password `cryptography` and no private key.
- `pkcs12/no-name-no-pwd.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) and key (`pkcs12/ca/ca_key.pem`), as well as two additional certificates (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`).
- `pkcs12/name-all-no-pwd.p12` - A PKCS12 file containing a cert (`pkcs12/ca/ca.pem`) and key (`pkcs12/ca/ca_key.pem`) with friendly name `name`, as well as two additional certificates (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`) with friendly names `name2` and `name3`, respectively.

- pkcs12/name-1-no-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem) with friendly name name, as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem).
- pkcs12/name-2-3-no-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names name2 and name3, respectively.
- pkcs12/name-2-no-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), the first having friendly name name2.
- pkcs12/name-3-no-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), the latter having friendly name name3.
- pkcs12/name-unicode-no-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem) with friendly name , as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names ä and ç, respectively.
- pkcs12/no-name-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-all-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem) with friendly name name, as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names name2 and name3 respectively, encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-1-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem) with friendly name name, as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-2-3-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names name2 and name3 respectively, encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-2-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), the first having friendly name name2, encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-3-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem), as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem), the latter having friendly name name2, encrypted via AES 256 CBC with the password cryptography.
- pkcs12/name-unicode-pwd.p12 - A PKCS12 file containing a cert (pkcs12/ca/ca.pem) and key (pkcs12/ca/ca_key.pem) with friendly name , as well as two additional certificates (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names ä and ç respectively, encrypted via AES 256 CBC with the password cryptography.
- pkcs12/no-cert-no-name-no-pwd.p12 - A PKCS12 file containing two certs (x509/cryptography.io.pem and x509/letsencryptx3.pem).
- pkcs12/no-cert-name-all-no-pwd.p12 - A PKCS12 file containing two certs (x509/cryptography.io.pem and x509/letsencryptx3.pem) with friendly names name2 and name3, respectively.
- pkcs12/no-cert-name-2-no-pwd.p12 - A PKCS12 file containing two certs (x509/cryptography.io.pem and x509/letsencryptx3.pem), the first having friendly name name2.

- `pkcs12/no-cert-name-3-no-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`), the second having friendly name `name3`.
- `pkcs12/no-cert-name-unicode-no-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`) with friendly names and `ï`, respectively.
- `pkcs12/no-cert-no-name-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`), encrypted via AES 256 CBC with the password `cryptography`.
- `pkcs12/no-cert-name-all-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`) with friendly names `name2` and `name3`, respectively, encrypted via AES 256 CBC with the password `cryptography`.
- `pkcs12/no-cert-name-2-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`), the first with friendly name `name2`, encrypted via AES 256 CBC with the password `cryptography`.
- `pkcs12/no-cert-name-3-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`), the second with friendly name `name3`, encrypted via AES 256 CBC with the password `cryptography`.
- `pkcs12/no-cert-name-unicode-pwd.p12` - A PKCS12 file containing two certs (`x509/cryptography.io.pem` and `x509/letsencryptx3.pem`) with friendly names and `ï`, respectively, encrypted via AES 256 CBC with the password `cryptography`.
- `pkcs12/java-truststore.p12` - A PKCS12 file containing two certs (`x509/custom/dsa_selfsigned_ca.pem` and `x509/letsencryptx3.pem`) with the first having a friendly name of *cert1*. Both have Java truststore attributes with ANY_EXTENDED_KEY_USAGE.

Custom PKCS7 Test Vectors

- `pkcs7/isrg.pem` - A PEM encoded PKCS7 file containing the ISRG X1 root CA.
- `pkcs7/amazon-roots.p7b` - A BER encoded PCKS7 file containing Amazon Root CA 2 and 3 generated by Apple Keychain.
- `pkcs7/amazon-roots.der` - A DER encoded PCKS7 file containing Amazon Root CA 2 and 3 generated by OpenSSL.
- `pkcs7/enveloped.pem` - A PEM encoded PKCS7 file with enveloped data.
- `pkcs7/enveloped-triple-des.pem` - A PEM encoded PKCS7 file with enveloped data, with content encrypted using DES EDE3 CBC (also called Triple DES), under the public key of `x509/custom/ca/rsa_ca.pem`.
- `pkcs7/enveloped-rsa-oaep.pem`- A PEM encoded PKCS7 file with enveloped data, with key encrypted using RSA-OAEP, under the public key of `x509/custom/ca/rsa_ca.pem`.
- `pkcs7/enveloped-no-content.der`- A DER encoded PKCS7 file with enveloped data, without encrypted content, with key encrypted under the public key of `x509/custom/ca/rsa_ca.pem`.
- `pkcs7/ascii-san.pem` - An invalid certificate adapted for S/MIME signature & verification. It has an ASCII subject alternative name stored as *otherName*.
- `pkcs7/non-ascii-san.pem` - An invalid certificate adapted for S/MIME signature & verification. It has an non-ASCII subject alternative name stored as *rfc822Name*.
- `pkcs7/authenticode.der` - A DER encoded PKCS7 file containing an Authenticode signature (with a `SpcIndirectDataContent` inner content type), extracted from Fedora 44's grubx64.efi.

Custom OpenSSH Test Vectors

- `ed25519-aesgcm-psw.key` and `ed25519-aesgcm-psw.key.pub` generated by exporting an Ed25519 key from `1password 8` with the password "password". This key is encrypted using the `aes256-gcm@openssh.com` algorithm.

Generated by `asymmetric/OpenSSH/gen.sh` using command-line tools from OpenSSH_7.6p1 package.

- `dsa-nopsw.key`, `dsa-nopsw.key.pub`, `dsa-nopsw.key-cert.pub` - DSA-1024 private key; and corresponding public key in plain format and with self-signed certificate.
- `dsa-psw.key`, `dsa-psw.key.pub` - Password-protected DSA-1024 private key and corresponding public key. Password is "password".
- `ecdsa-nopsw.key`, `ecdsa-nopsw.key.pub`, `ecdsa-nopsw.key-cert.pub` - SECP256R1 private key; and corresponding public key in plain format and with self-signed certificate.
- `ecdsa-psw.key`, `ecdsa-psw.key.pub` - Password-protected SECP384R1 private key and corresponding public key. Password is "password".
- `ed25519-nopsw.key`, `ed25519-nopsw.key.pub`, `ed25519-nopsw.key-cert.pub` - Ed25519 private key; and corresponding public key in plain format and with self-signed certificate.
- `ed25519-psw.key`, `ed25519-psw.key.pub` - Password-protected Ed25519 private key and corresponding public key. Password is "password".
- `rsa-nopsw.key`, `rsa-nopsw.key.pub`, `rsa-nopsw.key-cert.pub` - RSA-2048 private key; and corresponding public key in plain format and with self-signed certificate.
- `rsa-psw.key`, `rsa-psw.key.pub` - Password-protected RSA-2048 private key and corresponding public key. Password is "password".

Custom OpenSSH Certificate Test Vectors

- `p256-p256-duplicate-extension.pub` - A certificate with a duplicate extension.
- `p256-p256-non-lexical-extensions.pub` - A certificate with extensions in non-lexical order.
- `p256-p256-duplicate-crit-opts.pub` - A certificate with a duplicate critical option.
- `p256-p256-non-lexical-crit-opts.pub` - A certificate with critical options in non-lexical order.
- `p256-ed25519-non-singular-crit-opt-val.pub` - A certificate with a critical option that contains more than one value.
- `p256-ed25519-non-singular-ext-val.pub` - A certificate with an extension that contains more than one value.
- `dsa-p256.pub` - A certificate with a DSA public key signed by a P256 CA.
- `p256-dsa.pub` - A certificate with a P256 public key signed by a DSA CA.
- `p256-p256-broken-signature-key-type.pub` - A certificate with a P256 public key signed by a P256 CA, but the signature key type is set to `rsa-sha2-512`.
- `p256-p256-empty-principals.pub` - A certificate with a P256 public key signed by a P256 CA with an empty valid principals list.
- `p256-p256-invalid-cert-type.pub` - A certificate with a P256 public key signed by a P256 CA with an invalid certificate type.
- `p256-p384.pub` - A certificate with a P256 public key signed by a P384 CA.
- `p256-p521.pub` - A certificate with a P256 public key signed by a P521 CA.

- `p256-rsa-sha1.pub` - A certificate with a P256 public key signed by a RSA CA using SHA1.
- `p256-rsa-sha256.pub` - A certificate with a P256 public key signed by a RSA CA using SHA256.
- `p256-rsa-sha512.pub` - A certificate with a P256 public key signed by a RSA CA using SHA512.

Hashes

- MD5 from **RFC 1321**.
- RIPEMD160 from the RIPEMD website.
- SHA1 from NIST CAVP.
- SHA2 (224, 256, 384, 512, 512/224, 512/256) from NIST CAVP.
- SHA3 (224, 256, 384, 512) from NIST CAVP.
- SHAKE (128, 256) from NIST CAVP.
- Blake2s and Blake2b from OpenSSL test/evptests.txt.

HMAC

- HMAC-MD5 from **RFC 2202**.
- HMAC-SHA1 from **RFC 2202**.
- HMAC-RIPEMD160 from **RFC 2286**.
- HMAC-SHA2 (224, 256, 384, 512) from **RFC 4231**.

Key derivation functions

- HKDF (SHA1, SHA256) from **RFC 5869**.
- PBKDF2 (HMAC-SHA1) from **RFC 6070**.
- scrypt from the draft RFC.
- X9.63 KDF from NIST CAVP.
- SP 800-108 Counter Mode KDF (HMAC-SHA1, HMAC-SHA224, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512) from NIST CAVP.
- argon2 variants (argon2d, argon2i, and argon2id) from **RFC 9106**, OpenSSL's evpkdf_argon2.txt, and the argon2 command line application.

HPKE (Hybrid Public Key Encryption)

- HPKE test vectors from **RFC 9180**, obtained from the IETF CFRG HPKE repository. These vectors cover all four modes (Base, PSK, Auth, AuthPSK), multiple KEMs (X25519, X448, P-256, P-521), KDFs (HKDF-SHA256, HKDF-SHA512), and AEADs (AES-128-GCM, AES-256-GCM, ChaCha20Poly1305).
- HPKE post-quantum and hybrid vectors (including SHAKE and TurboSHAKE KDF IDs) from hpkewg/hpke-pq test vectors.

Key wrapping

- AES key wrap (AESKW) and 3DES key wrap test vectors from NIST CAVP.
- AES key wrap with padding vectors from Botan's key wrap vectors.

Recipes

- Fernet from its specification repository.
- Chunked encryption from the chunked reference implementation.

Symmetric ciphers

- AES (CBC, CFB, ECB, GCM, OFB, CCM) from NIST CAVP.
- AES CTR from **RFC 3686**.
- AES-GCM-SIV (KEY-LENGTH: 128, 256) from OpenSSL's evpciph_aes_gcm_siv.txt.
- AES-GCM-SIV (KEY-LENGTH: 192) generated by this project. See *AES-GCM-SIV vector creation*
- AES OCB3 from **RFC 7253**, dkg's additional OCB3 vectors, and OpenSSL's OCB vectors.
- AES SIV from OpenSSL's evpciph_aes_siv.txt.
- 3DES (CBC, CFB, ECB, OFB) from NIST CAVP.
- ARC4 (KEY-LENGTH: 40, 56, 64, 80, 128, 192, 256) from **RFC 6229**.
- ARC4 (KEY-LENGTH: 160) generated by this project. See: *ARC4 vector creation*
- Blowfish (CBC, CFB, ECB, OFB) from Bruce Schneier's vectors.
- Camellia (ECB) from NTT's Camellia page as linked by CRYPTREC.
- Camellia (CBC, CFB, OFB) from OpenSSL's test vectors.
- CAST5 (ECB) from **RFC 2144**.
- CAST5 (CBC, CFB, OFB) generated by this project. See: *CAST5 vector creation*
- ChaCha20 from **RFC 7539**.
- ChaCha20Poly1305 from **RFC 7539**, OpenSSL's evpciph.txt, and the BoringSSL ChaCha20Poly1305 tests.
- IDEA (ECB) from the NESSIE IDEA vectors created by NESSIE.
- IDEA (CBC, CFB, OFB) generated by this project. See: *IDEA vector creation*
- RC2-128-CBC generated by this project. See: *RC2 vector creation*
- SEED (ECB) from **RFC 4269**.
- SEED (CBC) from **RFC 4196**.
- SEED (CFB, OFB) generated by this project. See: *SEED vector creation*
- SM4 (CBC, CFB, CTR, ECB, OFB) from draft-ribose-cfrg-sm4-10.
- SM4 (GCM) from **RFC 8998**.

Two factor authentication

- HOTP from **RFC 4226**
- TOTP from **RFC 6238** (Note that an errata for the test vectors in RFC 6238 exists)

CMAC

- AES-128, AES-192, AES-256, 3DES from NIST SP-800-38B

Poly1305

- Test vectors from **RFC 7539**.

Creating test vectors

When official vectors are unavailable `cryptography` may choose to build its own using existing vectors as source material.

Created Vectors

AES-GCM-SIV vector creation

This page documents the code that was used to generate the AES-GCM-SIV test vectors for key lengths not available in the OpenSSL test vectors. All the vectors were generated using OpenSSL and verified with Rust.

Creation

The following Python script was run to generate the vector files. The OpenSSL test vectors were used as a base and modified to have 192-bit key length.

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.

import binascii

from cryptography.hazmat.primitives.ciphers.aead import AESGCMSIV


def convert_key_to_192_bits(key: str) -> str:
    """
    This takes existing 128 and 256-bit keys from test vectors from OpenSSL
    and makes them 192-bit by either appending 0 or truncating the key.
    """
    new_key = binascii.unhexlify(key)
    if len(new_key) == 16:
        new_key += b"\x00" * 8
    elif len(new_key) == 32:
        new_key = new_key[0:24]
    else:
        raise RuntimeError(
            "Unexpected key length. OpenSSL AES-GCM-SIV test vectors only "
            "contain 128-bit and 256-bit keys"
        )

    return binascii.hexlify(new_key).decode("ascii")


def encrypt(key: str, iv: str, plaintext: str, aad: str) -> (str, str):
```

(continues on next page)

(continued from previous page)

```
    aesgcmsiv = AESGCMSIV(binascii.unhexlify(key))
    encrypted_output = aesgcmsiv.encrypt(
        binascii.unhexlify(iv),
        binascii.unhexlify(plaintext),
        binascii.unhexlify(aad) if aad else None,
    )
    ciphertext, tag = encrypted_output[:-16], encrypted_output[-16:]

    return (
        binascii.hexlify(ciphertext).decode("ascii"),
        binascii.hexlify(tag).decode("ascii"),
    )


def build_vectors(filename):
    count = 0
    output = []
    key = None
    iv = None
    aad = None
    plaintext = None

    with open(filename) as vector_file:
        for line in vector_file:
            line = line.strip()
            if line.startswith("Key"):
                if count != 0:
                    ciphertext, tag = encrypt(key, iv, plaintext, aad)
                    output.append(f"Tag = {tag}\nCiphertext = {ciphertext}\n")
                output.append(f"\nCOUNT = {count}")
                count += 1
                aad = None
                _, key = line.split(" = ")
                key = convert_key_to_192_bits(key)
                output.append(f"Key = {key}")
            elif line.startswith("IV"):
                _, iv = line.split(" = ")
                output.append(f"IV = {iv}")
            elif line.startswith("AAD"):
                _, aad = line.split(" = ")
                output.append(f"AAD = {aad}")
            elif line.startswith("Plaintext"):
                _, plaintext = line.split(" = ")
                output.append(f"Plaintext = {plaintext}")

        ciphertext, tag = encrypt(key, iv, plaintext, aad)
        output.append(f"Tag = {tag}\nCiphertext = {ciphertext}\n")
        return "\n".join(output)


def write_file(data, filename):
    with open(filename, "w") as f:
```

(continues on next page)

(continued from previous page)

```
        f.write(data)


path = "vectors/cryptography_vectors/ciphers/AES/GCM-SIV/openssl.txt"
write_file(build_vectors(path), "aes-192-gcm-siv.txt")
```

Download link: `generate_aes192gcmsiv.py`

Verification

The following Rust program was used to verify the vectors.

```
use aes_gcm_siv::{
    aead::{Aead, KeyInit},
    AesGcmSiv, Nonce,
};

use aes::Aes192;
use aes_gcm_siv::aead::generic_array::GenericArray;
use aes_gcm_siv::aead::Payload;
use std::fs::File;
use std::io;
use std::io::BufRead;
use std::path::Path;

pub type Aes192GcmSiv = AesGcmSiv<Aes192>;

struct VectorArgs {
    nonce: String,
    key: String,
    aad: String,
    tag: String,
    plaintext: String,
    ciphertext: String,
}

fn validate(v: &VectorArgs) {
    let key_bytes = hex::decode(&v.key).unwrap();
    let nonce_bytes = hex::decode(&v.nonce).unwrap();
    let aad_bytes = hex::decode(&v.aad).unwrap();
    let plaintext_bytes = hex::decode(&v.plaintext).unwrap();
    let expected_ciphertext_bytes = hex::decode(&v.ciphertext).unwrap();
    let expected_tag_bytes = hex::decode(&v.tag).unwrap();

    let key_array: [u8; 24] = key_bytes.try_into().unwrap();
    let cipher = Aes192GcmSiv::new(&GenericArray::from(key_array));

    let payload = Payload {
        msg: plaintext_bytes.as_slice(),
        aad: aad_bytes.as_slice(),
    };
    let encrypted_bytes = cipher
```

(continues on next page)

(continued from previous page)

```
        .encrypt(Nonce::from_slice(nonce_bytes.as_slice()), payload)
        .unwrap();
    let (ciphertext_bytes, tag_bytes) = encrypted_bytes.split_at(plaintext_bytes.len());
    assert_eq!(ciphertext_bytes, expected_ciphertext_bytes);
    assert_eq!(tag_bytes, expected_tag_bytes);
}

fn validate_vectors(filename: &Path) {
    let file = File::open(filename).expect("Failed to open file");
    let reader = io::BufReader::new(file);

    let mut vector: Option<VectorArgs> = None;

    for line in reader.lines() {
        let line = line.expect("Failed to read line");
        let segments: Vec<&str> = line.splitn(2, " = ").collect();

        match segments.first() {
            Some(&"COUNT") => {
                if let Some(v) = vector.take() {
                    validate(&v);
                }
                vector = Some(VectorArgs {
                    nonce: String::new(),
                    key: String::new(),
                    aad: String::new(),
                    tag: String::new(),
                    plaintext: String::new(),
                    ciphertext: String::new(),
                });
            }
            Some(&"IV") => {
                if let Some(v) = &mut vector {
                    v.nonce = segments[1].parse().expect("Failed to parse IV");
                }
            }
            Some(&"Key") => {
                if let Some(v) = &mut vector {
                    v.key = segments[1].to_string();
                }
            }
            Some(&"AAD") => {
                if let Some(v) = &mut vector {
                    v.aad = segments[1].to_string();
                }
            }
            Some(&"Tag") => {
                if let Some(v) = &mut vector {
                    v.tag = segments[1].to_string();
                }
            }
            Some(&"Plaintext") => {
```

(continues on next page)

(continued from previous page)

```
                if let Some(v) = &mut vector {
                    v.plaintext = segments[1].to_string();
                }
            }
            Some(&"Ciphertext") => {
                if let Some(v) = &mut vector {
                    v.ciphertext = segments[1].to_string();
                }
            }
            _ => {}
        }
    }

    if let Some(v) = vector {
        validate(&v);
    }
}

fn main() {
    validate_vectors(Path::new(
        "vectors/cryptography_vectors/ciphers/AES/GCM-SIV/aes-192-gcm-siv.txt",
    ));
    println!("AES-192-GCM-SIV OK.")
}
```

Download link: `main.rs`

ARC4 vector creation

This page documents the code that was used to generate the ARC4 test vectors for key lengths not available in **RFC 6229**. All the vectors were generated using OpenSSL and verified with Go.

Creation

`cryptography` was modified to support ARC4 key lengths not listed in **RFC 6229**. Then the following Python script was run to generate the vector files.

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.

import binascii

from cryptography.hazmat.primitives import ciphers
from cryptography.hazmat.primitives.ciphers import algorithms

_RFC6229_KEY_MATERIALS = [
    (
        True,
        8 * "0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20",
    ),
    (
```

(continues on next page)

(continued from previous page)

```
        False,
        8 * "1ada31d5cf688221c109163908ebe51debb46227c6cc8b37641910833222772a",
    ),
]


_RFC6229_OFFSETS = [
    0,
    16,
    240,
    256,
    496,
    512,
    752,
    768,
    1008,
    1024,
    1520,
    1536,
    2032,
    2048,
    3056,
    3072,
    4080,
    4096,
]


_SIZES_TO_GENERATE = [160]


def _key_for_size(size, keyinfo):
    msb, key = keyinfo
    if msb:
        return key[: size // 4]
    else:
        return key[-size // 4 :]


def _build_vectors():
    count = 0
    output = []
    key = None
    plaintext = binascii.unhexlify(32 * "0")
    for size in _SIZES_TO_GENERATE:
        for keyinfo in _RFC6229_KEY_MATERIALS:
            key = _key_for_size(size, keyinfo)
            cipher = ciphers.Cipher(
                algorithms.ARC4(binascii.unhexlify(key)),
                None,
            )
            encryptor = cipher.encryptor()
```

(continues on next page)

(continued from previous page)

```
            current_offset = 0
            for offset in _RFC6229_OFFSETS:
                if offset % 16 != 0:
                    raise ValueError(
                        f"Offset {offset} is not evenly divisible by 16"
                    )
                while current_offset < offset:
                    encryptor.update(plaintext)
                    current_offset += len(plaintext)
                output.append(f"\nCOUNT = {count}")
                count += 1
                output.append(f"KEY = {key}")
                output.append(f"OFFSET = {offset}")
                output.append(f"PLAINTEXT = {binascii.hexlify(plaintext)}")
                output.append(
                    f"CIPHERTEXT = "
                    f"{binascii.hexlify(encryptor.update(plaintext))}"
                )
                current_offset += len(plaintext)
            assert not encryptor.finalize()
    return "\n".join(output)


def _write_file(data, filename):
    with open(filename, "w") as f:
        f.write(data)


if __name__ == "__main__":
    _write_file(_build_vectors(), "arc4.txt")
```

Download link: `generate_arc4.py`

Verification

The following Go code was used to verify the vectors.

```
package main

import (
        "bufio"
        "bytes"
        "crypto/rc4"
        "encoding/hex"
        "fmt"
        "os"
        "strconv"
        "strings"
)

func unhexlify(s string) []byte {
        bytes, err := hex.DecodeString(s)
```

(continues on next page)

(continued from previous page)

```
        if err != nil {
                panic(err)
        }
        return bytes
}

type vectorArgs struct {
        count      string
        offset     uint64
        key        string
        plaintext  string
        ciphertext string
}

type vectorVerifier interface {
        validate(count string, offset uint64, key, plaintext, expectedCiphertext []byte)
}

type arc4Verifier struct{}

func (o arc4Verifier) validate(count string, offset uint64, key, plaintext,␣
↪expectedCiphertext []byte) {
        if offset%16 != 0 || len(plaintext) != 16 || len(expectedCiphertext) != 16 {
                panic(fmt.Errorf("Unexpected input value encountered: offset=%v;␣
↪len(plaintext)=%v; len(expectedCiphertext)=%v",
                        offset,
                        len(plaintext),
                        len(expectedCiphertext)))
        }
        stream, err := rc4.NewCipher(key)
        if err != nil {
                panic(err)
        }

        var currentOffset uint64 = 0
        ciphertext := make([]byte, len(plaintext))
        for currentOffset <= offset {
                stream.XORKeyStream(ciphertext, plaintext)
                currentOffset += uint64(len(plaintext))
        }
        if !bytes.Equal(ciphertext, expectedCiphertext) {
                panic(fmt.Errorf("vector mismatch @ COUNT = %s:\n  %s != %s\n",
                        count,
                        hex.EncodeToString(expectedCiphertext),
                        hex.EncodeToString(ciphertext)))
        }
}

func validateVectors(verifier vectorVerifier, filename string) {
        vectors, err := os.Open(filename)
        if err != nil {
                panic(err)
```

(continues on next page)

(continued from previous page)

```
        }
        defer vectors.Close()

        var segments []string
        var vector *vectorArgs

        scanner := bufio.NewScanner(vectors)
        for scanner.Scan() {
                segments = strings.Split(scanner.Text(), " = ")

                switch {
                case strings.ToUpper(segments[0]) == "COUNT":
                        if vector != nil {
                                verifier.validate(vector.count,
                                        vector.offset,
                                        unhexlify(vector.key),
                                        unhexlify(vector.plaintext),
                                        unhexlify(vector.ciphertext))
                        }
                        vector = &vectorArgs{count: segments[1]}
                case strings.ToUpper(segments[0]) == "OFFSET":
                        vector.offset, err = strconv.ParseUint(segments[1], 10, 64)
                        if err != nil {
                                panic(err)
                        }
                case strings.ToUpper(segments[0]) == "KEY":
                        vector.key = segments[1]
                case strings.ToUpper(segments[0]) == "PLAINTEXT":
                        vector.plaintext = segments[1]
                case strings.ToUpper(segments[0]) == "CIPHERTEXT":
                        vector.ciphertext = segments[1]
                }
        }
        if vector != nil {
                verifier.validate(vector.count,
                        vector.offset,
                        unhexlify(vector.key),
                        unhexlify(vector.plaintext),
                        unhexlify(vector.ciphertext))
        }
}

func main() {
        validateVectors(arc4Verifier{}, "vectors/cryptography_vectors/ciphers/ARC4/arc4.
↪txt")
        fmt.Println("ARC4 OK.")
}
```

Download link: `verify_arc4.go`

CAST5 vector creation

This page documents the code that was used to generate the CAST5 CBC, CFB, OFB, and CTR test vectors as well as the code used to verify them against another implementation. The CBC, CFB, and OFB vectors were generated using OpenSSL and the CTR vectors were generated using Apple's CommonCrypto. All the generated vectors were verified with Go.

Creation

`cryptography` was modified to support CAST5 in CBC, CFB, and OFB modes. Then the following Python script was run to generate the vector files.

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.

import binascii

from cryptography.hazmat.primitives.ciphers import algorithms, base, modes


def encrypt(mode, key, iv, plaintext):
    cipher = base.Cipher(
        algorithms.CAST5(binascii.unhexlify(key)),
        mode(binascii.unhexlify(iv)),
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    count = 0
    output = []
    key = None
    iv = None
    plaintext = None

    with open(filename) as vector_file:
        for line in vector_file:
            line = line.strip()
            if line.startswith("KEY"):
                if count != 0:
                    output.append(
                        f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}"
                    )
                output.append(f"\nCOUNT = {count}")
                count += 1
                _, key = line.split(" = ")
                output.append(f"KEY = {key}")
            elif line.startswith("IV"):
                _, iv = line.split(" = ")
                iv = iv[0:16]
```

(continues on next page)

(continued from previous page)

```
                output.append(f"IV = {iv}")
            elif line.startswith("PLAINTEXT"):
                _, plaintext = line.split(" = ")
                output.append(f"PLAINTEXT = {plaintext}")
        output.append(f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}")
    return "\n".join(output)


def write_file(data, filename):
    with open(filename, "w") as f:
        f.write(data)


cbc_path = "tests/hazmat/primitives/vectors/ciphers/AES/CBC/CBCMMT128.rsp"
write_file(build_vectors(modes.CBC, cbc_path), "cast5-cbc.txt")
ofb_path = "tests/hazmat/primitives/vectors/ciphers/AES/OFB/OFBMMT128.rsp"
write_file(build_vectors(modes.OFB, ofb_path), "cast5-ofb.txt")
cfb_path = "tests/hazmat/primitives/vectors/ciphers/AES/CFB/CFB128MMT128.rsp"
write_file(build_vectors(modes.CFB, cfb_path), "cast5-cfb.txt")
ctr_path = "tests/hazmat/primitives/vectors/ciphers/AES/CTR/aes-128-ctr.txt"
write_file(build_vectors(modes.CTR, ctr_path), "cast5-ctr.txt")
```

Download link: `generate_cast5.py`

Verification

The following Go code was used to verify the vectors.

```
package main

import (
        "bufio"
        "bytes"
        "golang.org/x/crypto/cast5"
        "crypto/cipher"
        "encoding/hex"
        "fmt"
        "os"
        "strings"
)

func unhexlify(s string) []byte {
        bytes, err := hex.DecodeString(s)
        if err != nil {
                panic(err)
        }
        return bytes
}

type vectorArgs struct {
        count      string
        key        string
```

(continues on next page)

(continued from previous page)

```
        iv         string
        plaintext  string
        ciphertext string
}

type vectorVerifier interface {
        validate(count string, key, iv, plaintext, expectedCiphertext []byte)
}

type ofbVerifier struct{}

func (o ofbVerifier) validate(count string, key, iv, plaintext, expectedCiphertext␣
↪[]byte) {
        block, err := cast5.NewCipher(key)
        if err != nil {
                panic(err)
        }

        ciphertext := make([]byte, len(plaintext))
        stream := cipher.NewOFB(block, iv)
        stream.XORKeyStream(ciphertext, plaintext)

        if !bytes.Equal(ciphertext, expectedCiphertext) {
                panic(fmt.Errorf("vector mismatch @ COUNT = %s:\n  %s != %s\n",
                        count,
                        hex.EncodeToString(expectedCiphertext),
                        hex.EncodeToString(ciphertext)))
        }
}

type cbcVerifier struct{}

func (o cbcVerifier) validate(count string, key, iv, plaintext, expectedCiphertext␣
↪[]byte) {
        block, err := cast5.NewCipher(key)
        if err != nil {
                panic(err)
        }

        ciphertext := make([]byte, len(plaintext))
        mode := cipher.NewCBCEncrypter(block, iv)
        mode.CryptBlocks(ciphertext, plaintext)

        if !bytes.Equal(ciphertext, expectedCiphertext) {
                panic(fmt.Errorf("vector mismatch @ COUNT = %s:\n  %s != %s\n",
                        count,
                        hex.EncodeToString(expectedCiphertext),
                        hex.EncodeToString(ciphertext)))
        }
}

type cfbVerifier struct{}
```

(continues on next page)

(continued from previous page)

```
func (o cfbVerifier) validate(count string, key, iv, plaintext, expectedCiphertext␣
↪[]byte) {
        block, err := cast5.NewCipher(key)
        if err != nil {
                panic(err)
        }

        ciphertext := make([]byte, len(plaintext))
        stream := cipher.NewCFBEncrypter(block, iv)
        stream.XORKeyStream(ciphertext, plaintext)

        if !bytes.Equal(ciphertext, expectedCiphertext) {
                panic(fmt.Errorf("vector mismatch @ COUNT = %s:\n  %s != %s\n",
                        count,
                        hex.EncodeToString(expectedCiphertext),
                        hex.EncodeToString(ciphertext)))
        }
}

type ctrVerifier struct{}

func (o ctrVerifier) validate(count string, key, iv, plaintext, expectedCiphertext␣
↪[]byte) {
        block, err := cast5.NewCipher(key)
        if err != nil {
                panic(err)
        }

        ciphertext := make([]byte, len(plaintext))
        stream := cipher.NewCTR(block, iv)
        stream.XORKeyStream(ciphertext, plaintext)

        if !bytes.Equal(ciphertext, expectedCiphertext) {
                panic(fmt.Errorf("vector mismatch @ COUNT = %s:\n  %s != %s\n",
                        count,
                        hex.EncodeToString(expectedCiphertext),
                        hex.EncodeToString(ciphertext)))
        }
}

func validateVectors(verifier vectorVerifier, filename string) {
        vectors, err := os.Open(filename)
        if err != nil {
                panic(err)
        }
        defer vectors.Close()

        var segments []string
        var vector *vectorArgs

        scanner := bufio.NewScanner(vectors)
```

(continues on next page)

(continued from previous page)

```
        for scanner.Scan() {
                segments = strings.Split(scanner.Text(), " = ")

                switch {
                case strings.ToUpper(segments[0]) == "COUNT":
                        if vector != nil {
                                verifier.validate(vector.count,
                                        unhexlify(vector.key),
                                        unhexlify(vector.iv),
                                        unhexlify(vector.plaintext),
                                        unhexlify(vector.ciphertext))
                        }
                        vector = &vectorArgs{count: segments[1]}
                case strings.ToUpper(segments[0]) == "IV":
                        vector.iv = segments[1][:16]
                case strings.ToUpper(segments[0]) == "KEY":
                        vector.key = segments[1]
                case strings.ToUpper(segments[0]) == "PLAINTEXT":
                        vector.plaintext = segments[1]
                case strings.ToUpper(segments[0]) == "CIPHERTEXT":
                        vector.ciphertext = segments[1]
                }
        }

}

func main() {
        validateVectors(ofbVerifier{},
                "vectors/cryptography_vectors/ciphers/CAST5/cast5-ofb.txt")
        fmt.Println("OFB OK.")
        validateVectors(cfbVerifier{},
                "vectors/cryptography_vectors/ciphers/CAST5/cast5-cfb.txt")
        fmt.Println("CFB OK.")
        validateVectors(cbcVerifier{},
                "vectors/cryptography_vectors/ciphers/CAST5/cast5-cbc.txt")
        fmt.Println("CBC OK.")
        validateVectors(ctrVerifier{},
                "vectors/cryptography_vectors/ciphers/CAST5/cast5-ctr.txt")
        fmt.Println("CTR OK.")
}
```

Download link: `verify_cast5.go`

IDEA vector creation

This page documents the code that was used to generate the IDEA CBC, CFB, and OFB test vectors as well as the code used to verify them against another implementation. The vectors were generated using OpenSSL and verified with Botan.

Creation

cryptography was modified to support IDEA in CBC, CFB, and OFB modes. Then the following python script was run to generate the vector files.

```
import binascii

from cryptography.hazmat.primitives.ciphers import algorithms, base, modes


def encrypt(mode, key, iv, plaintext):
    cipher = base.Cipher(
        algorithms.IDEA(binascii.unhexlify(key)),
        mode(binascii.unhexlify(iv)),
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    with open(filename) as f:
        vector_file = f.read().splitlines()

    count = 0
    output = []
    key = None
    iv = None
    plaintext = None
    for line in vector_file:
        line = line.strip()
        if line.startswith("KEY"):
            if count != 0:
                output.append(
                    f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}"
                )
            output.append(f"\nCOUNT = {count}")
            count += 1
            _, key = line.split(" = ")
            output.append(f"KEY = {key}")
        elif line.startswith("IV"):
            _, iv = line.split(" = ")
            iv = iv[0:16]
            output.append(f"IV = {iv}")
        elif line.startswith("PLAINTEXT"):
            _, plaintext = line.split(" = ")
            output.append(f"PLAINTEXT = {plaintext}")

    output.append(f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}")
    return "\n".join(output)


def write_file(data, filename):
```

(continues on next page)

(continued from previous page)

```
    with open(filename, "w") as f:
        f.write(data)


CBC_PATH = "tests/hazmat/primitives/vectors/ciphers/AES/CBC/CBCMMT128.rsp"
write_file(build_vectors(modes.CBC, CBC_PATH), "idea-cbc.txt")
OFB_PATH = "tests/hazmat/primitives/vectors/ciphers/AES/OFB/OFBMMT128.rsp"
write_file(build_vectors(modes.OFB, OFB_PATH), "idea-ofb.txt")
CFB_PATH = "tests/hazmat/primitives/vectors/ciphers/AES/CFB/CFB128MMT128.rsp"
write_file(build_vectors(modes.CFB, CFB_PATH), "idea-cfb.txt")
```

Download link: `generate_idea.py`

Verification

The following Python code was used to verify the vectors using the Botan project's Python bindings.

```
import binascii

import botan

from tests.utils import load_nist_vectors

BLOCK_SIZE = 64


def encrypt(mode, key, iv, plaintext):
    encryptor = botan.Cipher(
        f"IDEA/{mode}/NoPadding", "encrypt", binascii.unhexlify(key)
    )

    cipher_text = encryptor.cipher(
        binascii.unhexlify(plaintext), binascii.unhexlify(iv)
    )
    return binascii.hexlify(cipher_text)


def verify_vectors(mode, filename):
    with open(filename) as f:
        vector_file = f.read().splitlines()

    vectors = load_nist_vectors(vector_file)
    for vector in vectors:
        ct = encrypt(mode, vector["key"], vector["iv"], vector["plaintext"])
        assert ct == vector["ciphertext"]


cbc_path = "tests/hazmat/primitives/vectors/ciphers/IDEA/idea-cbc.txt"
verify_vectors("CBC", cbc_path)
ofb_path = "tests/hazmat/primitives/vectors/ciphers/IDEA/idea-ofb.txt"
verify_vectors("OFB", ofb_path)
cfb_path = "tests/hazmat/primitives/vectors/ciphers/IDEA/idea-cfb.txt"
```

(continues on next page)

(continued from previous page)

```
verify_vectors("CFB", cfb_path)
```

Download link: `verify_idea.py`

SEED vector creation

This page documents the code that was used to generate the SEED CFB and OFB test vectors as well as the code used to verify them against another implementation. The vectors were generated using OpenSSL and verified with Botan.

Creation

`cryptography` was modified to support SEED in CFB and OFB modes. Then the following python script was run to generate the vector files.

```
import binascii

from cryptography.hazmat.primitives.ciphers import algorithms, base, modes


def encrypt(mode, key, iv, plaintext):
    cipher = base.Cipher(
        algorithms.SEED(binascii.unhexlify(key)),
        mode(binascii.unhexlify(iv)),
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    with open(filename) as f:
        vector_file = f.read().splitlines()

    count = 0
    output = []
    key = None
    iv = None
    plaintext = None
    for line in vector_file:
        line = line.strip()
        if line.startswith("KEY"):
            if count != 0:
                output.append(
                    f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}"
                )
            output.append(f"\nCOUNT = {count}")
            count += 1
            _, key = line.split(" = ")
            output.append(f"KEY = {key}")
        elif line.startswith("IV"):
            _, iv = line.split(" = ")
```

(continues on next page)

(continued from previous page)

```
            output.append(f"IV = {iv}")
        elif line.startswith("PLAINTEXT"):
            _, plaintext = line.split(" = ")
            output.append(f"PLAINTEXT = {plaintext}")

    output.append(f"CIPHERTEXT = {encrypt(mode, key, iv, plaintext)}")
    return "\n".join(output)


def write_file(data, filename):
    with open(filename, "w") as f:
        f.write(data)


OFB_PATH = "vectors/cryptography_vectors/ciphers/AES/OFB/OFBMMT128.rsp"
write_file(build_vectors(modes.OFB, OFB_PATH), "seed-ofb.txt")
CFB_PATH = "vectors/cryptography_vectors/ciphers/AES/CFB/CFB128MMT128.rsp"
write_file(build_vectors(modes.CFB, CFB_PATH), "seed-cfb.txt")
```

Download link: `generate_seed.py`

Verification

The following Python code was used to verify the vectors using the Botan project's Python bindings.

```
import binascii

import botan

from tests.utils import load_nist_vectors


def encrypt(mode, key, iv, plaintext):
    encryptor = botan.Cipher(
        f"SEED/{mode}/NoPadding", "encrypt", binascii.unhexlify(key)
    )

    cipher_text = encryptor.cipher(
        binascii.unhexlify(plaintext), binascii.unhexlify(iv)
    )
    return binascii.hexlify(cipher_text)


def verify_vectors(mode, filename):
    with open(filename) as f:
        vector_file = f.read().splitlines()

    vectors = load_nist_vectors(vector_file)
    for vector in vectors:
        ct = encrypt(mode, vector["key"], vector["iv"], vector["plaintext"])
        assert ct == vector["ciphertext"]
```

(continues on next page)

(continued from previous page)

```
ofb_path = "vectors/cryptography_vectors/ciphers/SEED/seed-ofb.txt"
verify_vectors("OFB", ofb_path)
cfb_path = "vectors/cryptography_vectors/ciphers/SEED/seed-cfb.txt"
verify_vectors("CFB", cfb_path)
```

Download link: `verify_seed.py`

HKDF vector creation

This page documents the code that was used to generate a longer HKDF test vector (1200 bytes) than is available in **RFC 5869**. All the vectors were generated using OpenSSL and verified with Go.

Creation

The following Python script was run to generate the vector files.

```
# This file is dual licensed under the terms of the Apache License, Version
# 2.0, and the BSD License. See the LICENSE file in the root of this repository
# for complete details.

import binascii

from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

IKM = binascii.unhexlify(b"0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b")
L = 1200
OKM = HKDF(
    algorithm=hashes.SHA256(),
    length=L,
    salt=None,
    info=None,
).derive(IKM)


def _build_vectors():
    output = [
        "COUNT = 0",
        "Hash = SHA-256",
        "IKM = " + binascii.hexlify(IKM).decode("ascii"),
        "salt = ",
        "info = ",
        f"L = {L}",
        "OKM = " + binascii.hexlify(OKM).decode("ascii"),
    ]
    return "\n".join(output)


def _write_file(data, filename):
    with open(filename, "w") as f:
        f.write(data)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    _write_file(_build_vectors(), "hkdf.txt")
```

Download link: `generate_hkdf.py`

Verification

The following Go code was used to verify the vectors.

```
package main

import (
        "bufio"
        "bytes"
        "crypto/sha256"
        "encoding/hex"
        "fmt"
        "golang.org/x/crypto/hkdf"
        "io"
        "os"
        "strconv"
        "strings"
)

func unhexlify(s string) []byte {
        bytes, err := hex.DecodeString(s)
        if err != nil {
                panic(err)
        }
        return bytes
}

func verifier(l uint64, ikm, okm []byte) bool {
        hash := sha256.New
        hkdf := hkdf.New(hash, ikm, nil, nil)
        okmComputed := make([]byte, l)
        io.ReadFull(hkdf, okmComputed)
        return bytes.Equal(okmComputed, okm)
}

func validateVectors(filename string) bool {
        vectors, err := os.Open(filename)
        if err != nil {
                panic(err)
        }
        defer vectors.Close()

        var segments []string
        var l uint64
        var ikm, okm string
```

(continues on next page)

(continued from previous page)

```
        scanner := bufio.NewScanner(vectors)
        for scanner.Scan() {
                segments = strings.Split(scanner.Text(), " = ")

                switch {
                case strings.ToUpper(segments[0]) == "L":
                        l, err = strconv.ParseUint(segments[1], 10, 64)
                        if err != nil {
                                panic(err)
                        }
                case strings.ToUpper(segments[0]) == "IKM":
                        ikm = segments[1]
                case strings.ToUpper(segments[0]) == "OKM":
                        okm = segments[1]
                }
        }
        return verifier(l, unhexlify(ikm), unhexlify(okm))
}

func main() {
        if validateVectors("vectors/cryptography_vectors/KDF/hkdf-generated.txt") {
                fmt.Println("HKDF OK.")
        } else {
                fmt.Println("HKDF failed.")
                os.Exit(1)
        }
}
```

Download link: `verify_hkdf.go`

RC2 vector creation

This page documents the code that was used to generate the RC2 CBC test vector. The CBC vector was generated using Go's internal RC2 implementation and verified using Go and OpenSSL.

Creation/Verification

The program below outputs a test vector in the standard format we use and also verifies that the encrypted value round trips as expected. The output was also checked against OpenSSL by modifying `cryptography` to support the algorithm. If you wish to run this program we recommend cloning the repository, which also contains the requisite `go.mod` file.

```
package main

import (
        "bytes"
        "crypto/cipher"
        "encoding/hex"
        "fmt"
        "rc2sucks/rc2"
)
```

(continues on next page)

(continued from previous page)

```
func main() {
        // Generate
        count := 1
        key := []byte("0000000000000000")
        iv := []byte("00000000")
        plaintext := []byte("the quick brown fox jumped over the lazy dog!!!!")
        ciphertext := make([]byte, len(plaintext))
        block, _ := rc2.New(key, 128)
        mode := cipher.NewCBCEncrypter(block, iv)
        mode.CryptBlocks(ciphertext, plaintext)
        fmt.Printf("COUNT = %v\n", count)
        fmt.Printf("Key = %s\n", hex.EncodeToString(key))
        fmt.Printf("IV = %s\n", hex.EncodeToString(iv))
        fmt.Printf("Plaintext = %s\n", hex.EncodeToString(plaintext))
        fmt.Printf("Ciphertext = %s\n", hex.EncodeToString(ciphertext))
        // Verify
        decrypted := make([]byte, len(plaintext))
        decmode := cipher.NewCBCDecrypter(block, iv)
        decmode.CryptBlocks(decrypted, ciphertext)
        if bytes.Equal(decrypted, plaintext) {
                fmt.Println("Success")
        } else {
                fmt.Println("Failed")
        }
}
```

Download link: `genrc2.go`

Download link: `rc2.go`

If official test vectors appear in the future the custom generated vectors should be discarded.

Any vectors generated by this method must also be prefixed with the following header format (substituting the correct information):

```
# CAST5 CBC vectors built for https://github.com/pyca/cryptography
# Derived from the AESVS MMT test data for CBC
# Verified against the CommonCrypto and Go crypto packages
# Key Length : 128
```

2.12.5 C bindings

C bindings are bindings to C libraries, using cffi whenever possible.

Bindings live in `cryptography.hazmat.bindings`.

When modifying the bindings you will need to recompile the C extensions to test the changes. This can be accomplished with `pip install -e .` in the project root. If you do not do this a `RuntimeError` will be raised.

Style guide

Don't name parameters:

```
/* Good */
long f(long);
/* Bad */
long f(long x);
```

...unless they're inside a struct:

```
struct my_struct {
    char *name;
    int number;
    ...;
};
```

Include `void` if the function takes no arguments:

```
/* Good */
long f(void);
/* Bad */
long f();
```

Wrap lines at 80 characters like so:

```
/* Pretend this went to 80 characters */
long f(long, long,
       int *)
```

Include a space after commas between parameters:

```
/* Good */
long f(int, char *)
/* Bad */
long f(int,char *)
```

Use C-style `/* */` comments instead of C++-style `//`:

```
// Bad
/* Good */
```

Values set by `#define` should be assigned the appropriate type. If you see this:

```
#define SOME_INTEGER_LITERAL 0x0;
#define SOME_UNSIGNED_INTEGER_LITERAL 0x0001U;
#define SOME_STRING_LITERAL "hello";
```

...it should be added to the bindings like so:

```
static const int SOME_INTEGER_LITERAL;
static const unsigned int SOME_UNSIGNED_INTEGER_LITERAL;
static const char *const SOME_STRING_LITERAL;
```

Adding constant, types, functions…

You can create bindings for any name that exists in some version of the library you're binding against. However, the project also has to keep supporting older versions of the library. In order to achieve this, binding modules have a `CUSTOMIZATIONS` constant, and there is a `CONDITIONAL_NAMES` constants in `src/cryptography/hazmat/bindings/openssl/_conditional.py`.

Let's say you want to enable quantum transmogrification. The upstream library implements this as the following API:

```
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_LEFT;
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_RIGHT;
typedef ... QM_TRANSMOGRIFICATION_CTX;
int QM_transmogrify(QM_TRANSMOGRIFICATION_CTX *, int);
```

To start, create a new constant that defines if the *actual* library has the feature you want, and add it to `TYPES`:

```
static const long Cryptography_HAS_QUANTUM_TRANSMOGRIFICATION;
```

This should start with `Cryptography_`, since we're adding it in this library. This prevents namespace collisions.

Then, define the actual features (constants, types, functions…) you want to expose. If it's a constant, just add it to `TYPES`:

```
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_LEFT;
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_RIGHT;
```

If it's a struct, add it to `TYPES` as well. The following is an opaque struct:

```
typedef ... QM_TRANSMOGRIFICATION_CTX;
```

… but you can also make some or all items in the struct accessible:

```
typedef struct {
    /* Fundamental constant k for your particular universe */
    BIGNUM *k;
    ...;
} QM_TRANSMOGRIFICATION_CTX;
```

For functions just add the signature to `FUNCTIONS`:

```
int QM_transmogrify(QM_TRANSMOGRIFICATION_CTX *, int);
```

Then, we define the `CUSTOMIZATIONS` entry. To do that, we have to come up with a C preprocessor expression that decides whether or not a feature exists in the library. For example:

```
#ifdef QM_transmogrify
```

Then, we set the flag that signifies the feature exists:

```
static const long Cryptography_HAS_QUANTUM_TRANSMOGRIFICATION = 1;
```

Otherwise, we set that flag to 0:

```
#else
static const long Cryptography_HAS_QUANTUM_TRANSMOGRIFICATION = 0;
```

Then, in that `#else` block, we define the names that aren't available as dummy values. For an integer constant, use 0:

```
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_LEFT = 0;
static const int QM_TRANSMOGRIFICATION_ALIGNMENT_RIGHT = 0;
```

For a function, it's a bit trickier. You have to define a function pointer of the appropriate type to be NULL:

```
int (*QM_transmogrify)(QM_TRANSMOGRIFICATION_CTX *, int) = NULL;
```

(To do that, copy the signature, put a `*` in front of the function name and wrap it in parentheses, and then put `= NULL` at the end).

Note how types don't need to be conditionally defined, as long as all the necessarily type definitions are in place.

Finally, add an entry to `CONDITIONAL_NAMES` with all of the things you want to conditionally export:

```
def cryptography_has_quantum_transmogrification():
    return [
        "QM_TRANSMOGRIFICATION_ALIGNMENT_LEFT",
        "QM_TRANSMOGRIFICATION_ALIGNMENT_RIGHT",
        "QM_transmogrify",
    ]


CONDITIONAL_NAMES = {
    ...
    "Cryptography_HAS_QUANTUM_TRANSMOGRIFICATION": (
        cryptography_has_quantum_transmogrification
    ),
}
```

Caveats

Sometimes, a set of loosely related features are added in the same version, and it's impractical to create `#ifdef` statements for each one. In that case, it may make sense to either check for a particular version. For example, to check for OpenSSL 3.2.0 or newer:

```
#if CRYPTOGRAPHY_OPENSSL_320_OR_GREATER
```

Sometimes, the version of a library on a particular platform will have features that you thought it wouldn't, based on its version. Occasionally, packagers appear to ship arbitrary VCS checkouts. As a result, sometimes you may have to add separate `#ifdef` statements for particular features. This kind of issue is typically only caught by running the tests on a wide variety of systems, which is the job of our continuous integration infrastructure.

2.13 Use of OpenSSL

`cryptography` depends on the OpenSSL C library for all cryptographic operation. OpenSSL is the de facto standard for cryptographic libraries and provides high performance along with various certifications that may be relevant to developers.

A list of supported versions can be found in our *Installation* documentation.

In general the backend should be considered an internal implementation detail of the project, but there are some public methods available for debugging purposes.

`cryptography.hazmat.backends.openssl.`**`backend`**

`openssl_version_text()`

Return text
The friendly string name of the loaded OpenSSL library. This is not necessarily the same version as it was compiled against.

`openssl_version_number()`

Added in version 1.8.

Return int
The integer version of the loaded OpenSSL library. This is defined in `opensslv.h` as `OPENSSL_VERSION_NUMBER` and is typically shown in hexadecimal (e.g. `0x1010003f`). This is not necessarily the same version as it was compiled against.

2.13.1 Legacy provider in OpenSSL 3.x

Added in version 39.0.0.

Users can set `CRYPTOGRAPHY_OPENSSL_NO_LEGACY` environment variable to disable the legacy provider in OpenSSL 3.x. This will disable legacy cryptographic algorithms, including `Blowfish`, `CAST5`, `SEED`, `ARC4`, and `RC2` (which is used by some encrypted serialization formats).

Additionally, the `CRYPTOGRAPHY_BUILD_OPENSSL_NO_LEGACY` environment variable can be set during the build process to prevent the library from ever attempting to load the legacy provider.

If loading the legacy provider is not disabled and the legacy provider fails to load, a warning is emitted.

2.14 Security

We take the security of `cryptography` seriously. The following are a set of policies we have adopted to ensure that security issues are addressed in a timely fashion.

2.14.1 Known vulnerabilities

A list of all known vulnerabilities in `cryptography` can be found on osv.dev, as well as other ecosystem vulnerability databases. They can automatically be scanned for using tools such as pip-audit or osv-scan.

2.14.2 Infrastructure

In addition to `cryptography`'s code, we're also concerned with the security of the infrastructure we run (primarily `cryptography.io`). If you discover a security vulnerability in our infrastructure, we ask you to report it using the same procedure.

2.14.3 What is a security issue?

Anytime it's possible to write code using `cryptography`'s public API which does not provide the guarantees that a reasonable developer would expect it to based on our documentation.

That's a bit academic, but basically it means the scope of what we consider a vulnerability is broad, and we do not require a proof of concept or even a specific exploit, merely a reasonable threat model under which `cryptography` could be attacked.

To give a few examples of things we would consider security issues:

- If a recipe, such as Fernet, made it easy for a user to bypass confidentiality or integrity with the public API (e.g. if the API let a user reuse nonces).
- If, under any circumstances, we used a CSPRNG which wasn't fork-safe.

- If `cryptography` used an API in an underlying C library and failed to handle error conditions safely.

Examples of things we wouldn't consider security issues:

- Offering ECB mode for symmetric encryption in the *Hazmat* layer. Though ECB is critically weak, it is documented as being weak in our documentation.
- Using a variable time comparison somewhere, if it's not possible to articulate any particular program in which this would result in problematic information disclosure.
- An uncaught `PanicException` from `pyo3`. In general, these represent bugs that should be filed as regular issues, but they are not security vulnerabilities because they do not cause memory unsafety or undefined behavior – the process remains in a well-defined state.

In general, if you're unsure, we request that you to default to treating things as security issues and handling them sensitively, the worst thing that can happen is that we'll ask you to file a public issue.

2.14.4 Reporting a security issue

We ask that you do not report security issues to our normal GitHub issue tracker.

If you believe you've identified a security issue with `cryptography`, please report it via our security advisory page.

Once you've submitted an issue, you should receive an acknowledgment within 48 hours, and depending on the action to be taken, you may receive further follow-up.

2.14.5 Supported Versions

At any given time, we will provide security support for the main branch as well as the most recent release.

2.14.6 New releases for OpenSSL updates

As of versions 0.5, 1.0.1, and 2.0.0, `cryptography` statically links OpenSSL in binary distributions for Windows, macOS, and Linux respectively, to ease installation. Due to this, `cryptography` will release a new version whenever OpenSSL has a security or bug fix release to avoid shipping insecure software.

Like all our other releases, this will be announced on the mailing list and we strongly recommend that you upgrade as soon as possible.

2.14.7 Disclosure Process

When we become aware of a security bug in `cryptography`, we will endeavor to fix it and issue a release as quickly as possible. We will generally issue a new release for any security issue.

The steps for issuing a security release are described in our *Doing a release* documentation.

2.15 Known security limitations

2.15.1 Secure memory wiping

Memory wiping is used to protect secret data or key material from attackers with access to deallocated memory. This is a defense-in-depth measure against vulnerabilities that leak application memory.

Many `cryptography` APIs which accept `bytes` also accept types which implement the buffer interface. Thus, users wishing to do so can pass `memoryview` or another mutable type to `cryptography` APIs, and overwrite the contents once the data is no longer needed.

However, `cryptography` does not clear memory by default, as there is no way to clear immutable structures such as `bytes`. As a result, `cryptography`, like almost all software in Python is potentially vulnerable to this attack. The

CERT secure coding guidelines assesses this issue as "Severity: medium, Likelihood: unlikely, Remediation Cost: expensive to repair" and we do not consider this a high risk for most users.

2.15.2 RSA PKCS1 v1.5 constant time decryption

RSA decryption has several different modes, one of which is PKCS1 v1.5. When used in **online contexts**, a secure protocol implementation requires that peers not be able to tell whether RSA PKCS1 v1.5 decryption failed or succeeded, even by timing variability.

`cryptography` does not provide an API that makes this possible, due to the fact that RSA decryption raises an exception on failure, which takes a different amount of time than returning a value in the success case.

In OpenSSL 3.2.0 and newer, this is automatically mitigated by OpenSSL (by returning a random value and never raising an exception). If you are using cryptography with an older version of OpenSSL, such attacks are still possible.

Regardless of OpenSSL version, we recommend not implementing or using online protocols that use RSA PKCS1 v1.5 decryption, as such protocols generally have poor security properties due to their lack of forward security.

2.16 API stability

From its first release, `cryptography` has had a strong API stability policy.

2.16.1 What does this policy cover?

This policy includes any API or behavior that is documented in this documentation.

2.16.2 What does "stable" mean?

- Public APIs will not be removed or renamed without providing a compatibility alias.
- The behavior of existing APIs will not change.

2.16.3 What doesn't this policy cover?

- We may add new features, things like the result of `dir(obj))` or the contents of `obj.__dict__` may change.
- Objects are not guaranteed to be pickleable, and pickled objects from one version of `cryptography` may not be loadable in future versions.
- Unless otherwise documented, types in `cryptography` are not intended to be sub-classed, and we do not guarantee that behavior with respect to sub-classes will be stable.
- Development versions of `cryptography`. Before a feature is in a release, it is not covered by this policy and may change.

Security

One exception to our API stability policy is for security. We will violate this policy as necessary in order to resolve a security issue or harden `cryptography` against a possible attack.

2.16.4 Versioning

Version 35.0.0+

Beginning with release 35.0.0 `cryptography` uses a Firefox-inspired version scheme.

Given a version `cryptography X.Y.Z`,

- `X` indicates the major version number. This is incremented on any feature release.

- `Y` is always `0`.
- `Z` is an integer that is incremented for minor backward-compatible releases (such as fixing security issues or correcting regressions in a major release).

This scheme is compatible with SemVer, though many major releases will **not** include any backwards-incompatible changes.

Deprecation

From time to time we will want to change the behavior of an API or remove it entirely. In that case, here's how the process will work:

- In `cryptography X.0.0` the feature exists.
- In `cryptography (X + 1).0.0` using that feature will emit a `CryptographyDeprecationWarning` (base class `UserWarning`).
- In `cryptography (X + 2).0.0` using that feature will emit a `CryptographyDeprecationWarning`.
- In `cryptography (X + 3).0.0` the feature will be removed or changed.

In short, code that runs without warnings will always continue to work for a period of two major releases.

From time to time, we may decide to deprecate an API that is particularly widely used. In these cases, we may decide to provide an extended deprecation period, at our discretion.

Previous Scheme

Before version 35.0.0 this project uses a custom versioning scheme as described below.

Given a version `cryptography X.Y.Z`,

- `X.Y` is a decimal number that is incremented for potentially-backwards-incompatible releases.
  - This increases like a standard decimal. In other words, 0.9 is the ninth release, and 1.0 is the tenth (not 0.10). The dividing decimal point can effectively be ignored.
- `Z` is an integer that is incremented for backward-compatible releases.

2.17 Doing a release

Doing a release of `cryptography` requires a few steps.

2.17.1 Security Releases

In addition to the other steps described below, for a release which fixes a security vulnerability, you should also include the following steps:

- Request a CVE from MITRE. Once you have received the CVE, it should be included in the *Changelog*. Ideally you should request the CVE before starting the release process so that the CVE is available at the time of the release.
- Document the CVE in the git commit that fixes the issue.
- Ensure that the *Changelog* entry credits whoever reported the issue and contains the assigned CVE.
- Publish a GitHub Security Advisory on the repository with all relevant information.
- The release should be announced on the oss-security mailing list, in addition to the regular announcement lists.

2.17.2 Verifying OpenSSL version

The release process creates wheels bundling OpenSSL for Windows, macOS, and Linux. Check that the Windows, macOS, and Linux builders (the `manylinux` containers) have the latest OpenSSL. If anything is out of date follow the instructions for upgrading OpenSSL.

2.17.3 Upgrading OpenSSL

Use the upgrading OpenSSL issue template.

2.17.4 Bumping the version number

The next step in doing a release is bumping the version number in the software.

- Run `python release.py bump-version {new_version}`
- Set the release date in the *Changelog*.
- Do a commit indicating this.
- Send a pull request with this.
- Wait for it to be merged.

2.17.5 Performing the release

The commit that merged the version number bump is now the official release commit for this release. You will need to have `git` configured to perform signed tags. Once this has happened:

- Run `python release.py release`.

The release should now be available on PyPI and a tag should be available in the repository.

2.17.6 Verifying the release

You should verify that `pip install cryptography` works correctly:

```
>>> import cryptography
>>> cryptography.__version__
'...'
>>> import cryptography_vectors
>>> cryptography_vectors.__version__
'...'
```

Verify that this is the version you just released.

For the Windows wheels check the builds for the `cryptography-wheel-builder` job and verify that the final output for each build shows it loaded and linked the expected OpenSSL version.

2.17.7 Post-release tasks

- Send an email to the mailing list and python-announce announcing the release.
- Close the milestone for the previous release on GitHub.
- For major version releases, send a pull request to pyOpenSSL increasing the maximum `cryptography` version pin and perform a pyOpenSSL release.
- Update the version number to the next major (e.g. `0.5.dev1`) with `python release.py bump-version {new_version}`.

- Add new *Changelog* entry with next version and note that it is under active development
- Send a pull request with these items
- Check for any outstanding code undergoing a deprecation cycle by looking in `cryptography.utils` for `DeprecatedIn**` definitions. If any exist open a ticket to increment them for the next release.

2.18 Community

You can find `cryptography` all over the web:

- Mailing list
- Source code
- Issue tracker
- Documentation
- IRC: `#pyca` on `irc.libera.chat`

Wherever we interact, we adhere to the Python Community Code of Conduct.

2.19 Glossary

A-label
The ASCII compatible encoded (ACE) representation of an internationalized (unicode) domain name. A-labels begin with the prefix `xn--`. To create an A-label from a unicode domain string use a library like idna.

authentication
The process of verifying that a message was created by a specific individual (or program). Like encryption, authentication can be either symmetric or asymmetric. Authentication is necessary for effective encryption.

bits
A bit is binary value – a value that has only two possible states. Typically binary values are represented visually as 0 or 1, but remember that their actual value is not a printable character. A byte on modern computers is 8 bits and represents 256 possible values. In cryptographic applications when you see something say it requires a 128 bit key, you can calculate the number of bytes by dividing by 8. 128 divided by 8 is 16, so a 128 bit key is a 16 byte key.

bytes-like
A bytes-like object contains binary data and supports the buffer protocol. This includes `bytes`, `bytearray`, and `memoryview` objects. It is *unsafe* to pass a mutable object (e.g., a `bytearray` or other implementer of the buffer protocol) and to mutate it concurrently with the operation it has been provided for.

ciphertext
The encoded data, it's not user readable. Potential attackers are able to see this.

ciphertext indistinguishability
This is a property of encryption systems whereby two encrypted messages aren't distinguishable without knowing the encryption key. This is considered a basic, necessary property for a working encryption system.

decryption
The process of converting ciphertext to plaintext.

encryption
The process of converting plaintext to ciphertext.

key
Secret data is encoded with a function using this key. Sometimes multiple keys are used. These **must** be kept secret, if a key is exposed to an attacker, any data encrypted with it will be exposed.

nonce
A nonce is a **n**umber used **once**. Nonces are used in many cryptographic protocols. Generally, a nonce does not have to be secret or unpredictable, but it must be unique. A nonce is often a random or pseudo-random number (see *Random number generation*). Since a nonce does not have to be unpredictable, it can also take a form of a counter.

opaque key
An opaque key is a type of key that allows you to perform cryptographic operations such as encryption, decryption, signing, and verification, but does not allow access to the key itself. Typically an opaque key is loaded from a hardware security module (HSM).

plaintext
User-readable data you care about.

private key
This is one of two keys involved in *public-key cryptography*. It can be used to decrypt messages which were encrypted with the corresponding *public key*, as well as to create signatures, which can be verified with the corresponding *public key*. These **must** be kept secret, if they are exposed, all encrypted messages are compromised, and an attacker will be able to forge signatures.

public key
This is one of two keys involved in *public-key cryptography*. It can be used to encrypt messages for someone possessing the corresponding *private key* and to verify signatures created with the corresponding *private key*. This can be distributed publicly, hence the name.

public-key cryptography
asymmetric cryptography
Cryptographic operations where encryption and decryption use different keys. There are separate encryption and decryption keys. Typically encryption is performed using a *public key*, and it can then be decrypted using a *private key*. Asymmetric cryptography can also be used to create signatures, which can be generated with a *private key* and verified with a *public key*.

symmetric cryptography
Cryptographic operations where encryption and decryption use the same key.

thread safety
All immutable objects in `cryptography` are safe to use in multi-threaded environments. This means they can be shared across threads without requiring additional synchronization. Mutable objects, such as hash contexts, can also be shared, but concurrent modification may lead to exceptions or incorrect results. When working with cryptographic operations in a multi-threaded application, ensure that any mutable objects are used in a thread-safe manner.

U-label
The presentational unicode form of an internationalized domain name. U-labels use unicode characters outside the ASCII range and are encoded as A-labels when stored in certificates.

unsafe
This is a term used to describe an operation where the user must ensure that the input is correct. Failure to do so can result in crashes, hangs, and other security issues.

2.20 Statements

Occasional statements from the Python Cryptographic Authority about the project.

2.20.1 The State of OpenSSL for `pyca/cryptography`

Published: January 14, 2026

For the past 12 years, we (Paul Kehrer and Alex Gaynor) have maintained the Python `cryptography` library (also known as `pyca/cryptography` or cryptography.io). For that entire period, we've relied on OpenSSL to provide core cryptographic algorithms. This past October, we gave a talk at the OpenSSL Conference describing our experiences. This talk focuses on the growing problems we have with OpenSSL's direction. The mistakes we see in OpenSSL's development have become so significant that we believe substantial changes are required — either to OpenSSL, or to our reliance on it.

Fundamentally, OpenSSL's trajectory can be understood as a play in three acts:

- In the pre-Heartbleed era (pre-2014), OpenSSL was under-maintained and languishing, substantially lagging behind expectations.
- In the immediate post-Heartbleed era, OpenSSL's maintenance was reinvigorated and it made substantial progress and improvements. It grew a real code review process, began running tests in CI, adopted fuzz testing, and matured its release process.
- Finally, in 2021 OpenSSL 3 was released. OpenSSL 3 introduced new APIs and had large internal refactors. Relative to previous OpenSSL versions, OpenSSL 3 had significant regressions in performance, complexity, API ergonomics, and didn't make needed improvements in areas like testing, verification, and memory safety. Over the same period, OpenSSL's forks have all made progress in these areas. Many of our concerns about OpenSSL's direction in this time have substantial overlap with those highlighted by HAProxy.

The remainder of this post describes the problems we have with OpenSSL in more detail, and concludes with the changes we are making to our own policies in response. To avoid burying the lede, we intend to pursue several approaches to reducing our reliance on OpenSSL.

Performance

Compared to OpenSSL 1.1.1, OpenSSL 3 has significant performance regressions in areas such as parsing and key loading.

Several years ago, we filed a bug reporting that elliptic curve public key loading had regressed 5-8x between OpenSSL 1.1.1 and 3.0.7. The reason we had noticed this is that performance had gotten so bad that we'd seen it in our test suite runtimes. Since then, OpenSSL has improved performance such that it's *only* 3x slower than it used to be. But more significantly, the response to the issue was that, 'regression was expected with OpenSSL 3, and while there might be some optimizations, we shouldn't expect it to ever get back to 1.1.1 levels'. Performance regressions can be acceptable, and even appropriate, when they improve other areas of the library, however as we'll describe, the cause of these regressions has been *other* mistakes, and not offsetting improvements.

As a result of these sorts of regressions, when `pyca/cryptography` migrated X.509 certificate parsing from OpenSSL to our own Rust code, we got a 10x performance improvement relative to OpenSSL 3 (n.b., some of this improvement is attributable to advantages in our own code, but much is explainable by the OpenSSL 3 regressions). Later, moving public key parsing to our own Rust code made end-to-end X.509 path validation 60% faster — just improving key loading led to a 60% end-to-end improvement, that's how extreme the overhead of key parsing in OpenSSL was.

The fact that we are able to achieve better performance doing our own parsing makes clear that doing better is practical. And indeed, our performance is not a result of clever SIMD micro-optimizations, it's the result of doing simple things that work: we avoid copies, allocations, hash tables, indirect calls, and locks — none of which should be required for parsing basic DER structures.

Complexity and APIs

OpenSSL 3 started the process of substantially changing its APIs — it introduced `OSSL_PARAM` and has been using those for all new API surfaces (including those for post-quantum cryptographic algorithms). In short, `OSSL_PARAM` works by passing arrays of key-value pairs to functions, instead of normal argument passing. This reduces performance, reduces compile-time verification, increases verbosity, and makes code less readable. To the extent there is an argument in favor of it, we infer that the benefit is that it allows OpenSSL to use the same API (and ABI) for different algorithms with different parameters, allowing things like reading algorithm parameters from configuration files with generic configuration parsing code that doesn't need to be updated when new algorithms are added to OpenSSL.

For a concrete comparison of the verbosity, performing an ML-KEM encapsulation with OpenSSL takes 37 lines with 6 fallible function calls. Doing so with BoringSSL takes 19 lines with 3 fallible function calls.

In addition to making public APIs more frustrating and error prone to use, OpenSSL internals have also become more complex. For example, in order to make managing arrays of `OSSL_PARAM` palatable, many OpenSSL source files are no longer simply C files, they now have a custom Perl preprocessor for their C code.

OpenSSL 3 also introduced the notion of "providers" (obsoleting, but not replacing, the previous ENGINE APIs), which allow for external implementations of algorithms (including algorithms provided by OpenSSL itself). This was the source of innumerable performance regressions, due to poorly designed APIs. In particular, OpenSSL allowed replacing any algorithm at any point in program execution, which necessitated adding innumerable allocations and locks to nearly every operation. To mitigate this, OpenSSL then added more caches, and ultimately RCU (Read-Copy-Update) — a complex memory management strategy which had difficult to diagnose bugs.

From our perspective, this is a cycle of compounding bad decisions: the providers API was incorrectly designed (there is no need to be able to redefine SHA-256 at arbitrary points in program execution) leading to performance regressions. This led to additional complexity to mitigate those regressions in the form of caching and RCU, which in term led to more bugs. And after all that, performance was *still* worse than it had been at the beginning.

Finally, taking an OpenSSL public API and attempting to trace the implementation to see how it is implemented has become an exercise in self-flagellation. Being able to read the source to understand how something works is important both as part of self-improvement in software engineering, but also because as sophisticated consumers there are inevitably things about how an implementation works that aren't documented, and reading the source gives you ground truth. The number of indirect calls, optional paths, `#ifdef`, and other obstacles to comprehension is astounding. We cannot overstate the extent to which just reading the OpenSSL source code has become miserable — in a way that both wasn't true previously, and isn't true in LibreSSL, BoringSSL, or AWS-LC.

Testing and Verification

We joke that the Python Cryptographic Authority is a CI engineering project that incidentally produces a cryptography library. The joke reflects our real belief that investment in testing and automation enables Pareto improvements in development speed and correctness — to the point that it can make other work look trivial.

The OpenSSL project does not sufficiently prioritize testing. While OpenSSL's testing has improved substantially since the pre-Heartbleed era there are quite significant gaps. The gaps in OpenSSL's test coverage were acutely visible during the OpenSSL 3.0 development cycle — where the project was extremely reliant on the community to report regressions experienced during the extended alpha and beta period (covering 19 pre-releases over the course of 16 months), because their own tests were insufficient to catch unintended real-world breakages. Despite the known gaps in OpenSSL's test coverage, it's still common for bug fixes to land without an accompanying regression test.

OpenSSL's CI is exceptionally flaky, and the OpenSSL project has grown to tolerate this flakiness, which masks serious bugs. OpenSSL 3.0.4 contained a critical buffer overflow in the RSA implementation on AVX-512-capable CPUs. This bug was actually caught by CI — but because the crash only occurred when the CI runner happened to have an AVX-512 CPU (not all did), the failures were apparently dismissed as flakiness. Three years later, the project still merges code with failing tests: the day we prepared our conference slides, five of ten recent commits had failing CI checks, and the day before we delivered the talk, every single commit had failing cross-compilation builds.

This incident also speaks to the value of adopting tools like Intel SDE, which allows controlled testing against CPUs with different subsets of x86-64 extension instructions. Using Intel SDE to have dedicated test jobs with and without

AVX-512 would have made the nature of the failure immediately legible and reproducible.

OpenSSL is not keeping pace with the state of the art in formal verification. Formal methods have gone from academic novelty to practical reality for meaningful chunks of cryptographic code. BoringSSL and AWS-LC have incorporated formally verified implementations and use automated reasoning to increase assurance.

Memory Safety

At the time OpenSSL was created, there were no programming languages that meaningfully provided performance, embeddability, and memory safety — if you wanted a memory safe language, you were committing to giving up performance and adding a garbage collector.

The world has changed. Nearly 5 years ago, `pyca/cryptography` issued our first release incorporating Rust code, and since then we have migrated nearly all functionality to Rust, using a mix of pure-Rust for all parsing and X.509 operations combined with using OpenSSL for providing cryptographic algorithms — gaining performance wins and avoiding several OpenSSL CVEs. We know these transitions are possible.

A library committed to security needs to make a long-term commitment to a migration to a memory safe programming language. OpenSSL has shown no initiative at all on this issue.

Contributing Causes

Whenever issues with an open source project are raised, many will suggest this is an issue of funding or tragedy of the commons. This is inapposite, in the past decade, post-Heartbleed, OpenSSL has received considerable funding, and at this moment the OpenSSL Corporation and Foundation employ more software engineers than work full time on either BoringSSL or LibreSSL. The problems we have described are not ones caused by underfunding.

We do not fully understand the motivations that led to the public APIs and internal complexity we've described here. We've done our best to reverse engineer them by asking "what would motivate someone to do this" and often we've found ourselves coming up short. The fact that none of the other OpenSSL forks have made these same design choices is informative to the question of "was this necessary".

Future Directions

Our experience with OpenSSL has been on a negative trajectory for several years. As a result of these issues, we are making the following changes to our (admittedly undocumented) policies.

First, we will no longer require OpenSSL implementations for new functionality. Where we deem it desirable, we will add new APIs that are only on LibreSSL/BoringSSL/AWS-LC. Concretely, we expect to add ML-KEM and ML-DSA APIs that are only available with LibreSSL/BoringSSL/AWS-LC, and not with OpenSSL.

Second, we currently statically link a copy of OpenSSL in our wheels (binary artifacts). We are beginning the process of looking into what would be required to change our wheels to link against one of the OpenSSL forks.

If we are able to successfully switch to one of OpenSSL's forks for our binary wheels, we will begin considering the circumstances under which we would drop support for OpenSSL entirely.

Lastly, in the long term, we are actively tracking non-OpenSSL derived cryptography libraries such as Graviola as potential alternatives.

We recognize that changes in which libraries we use to provide cryptographic implementations have substantial impact on our users — particularly redistributors. We do not contemplate these steps lightly, nor do we anticipate making them hastily. However, due to the gravity of our concerns, we are compelled to act. If you rely on `pyca/cryptography`'s support for OpenSSL, the best way to avoid the most drastic steps contemplated here is to engage with the OpenSSL project and contribute to improvements on these axes.

Note

`cryptography` has not been subjected to an external audit of its code or documentation. If you're interested in discussing an audit please *get in touch*.

PYTHON MODULE INDEX

C

cryptography.hazmat.asn1, 274
cryptography.hazmat.decrepit.ciphers.algorithms, 282
cryptography.hazmat.decrepit.ciphers.modes, 284
cryptography.hazmat.primitives.asymmetric.dsa, 170
cryptography.hazmat.primitives.asymmetric.ec, 141
cryptography.hazmat.primitives.asymmetric.padding, 156
cryptography.hazmat.primitives.asymmetric.rsa, 152
cryptography.hazmat.primitives.ciphers, 248
cryptography.hazmat.primitives.ciphers.aead, 100
cryptography.hazmat.primitives.ciphers.modes, 252
cryptography.hazmat.primitives.hashes, 242
cryptography.hazmat.primitives.hpke, 206
cryptography.hazmat.primitives.kdf, 211
cryptography.hazmat.primitives.keywrap, 235
cryptography.hazmat.primitives.padding, 261
cryptography.hazmat.primitives.serialization, 175
cryptography.x509.verification, 33

INDEX

A

A-label, **399**
aa_compromise (*cryptography.x509.ReasonFlags attribute*), 79
ACCEPTABLE_RESPONSES (*cryptography.x509.oid.OCSPExtensionOID attribute*), 98
access_location (*cryptography.x509.AccessDescription attribute*), 77
access_method (*cryptography.x509.AccessDescription attribute*), 77
AccessDescription (*class in cryptography.x509*), 77
ACP_NODE_NAME (*cryptography.x509.oid.OtherNameFormOID attribute*), 95
add_attribute() (*cryptography.x509.CertificateSigningRequestBuilder method*), 60
add_certificate() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7SignatureBuilder method*), 192
add_certificate() (*cryptography.x509.ocsp.OCSPRequestBuilder method*), 21
add_certificate_by_hash() (*cryptography.x509.ocsp.OCSPRequestBuilder method*), 21
add_critical_option() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 186
add_extension() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 186
add_extension() (*cryptography.x509.CertificateBuilder method*), 53
add_extension() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
add_extension() (*cryptography.x509.CertificateSigningRequestBuilder method*), 60
add_extension() (*cryptography.x509.ocsp.OCSPRequestBuilder method*), 21
add_extension() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 24
add_extension() (*cryptography.x509.RevokedCertificateBuilder method*), 59
add_profession_info (*cryptography.x509.ProfessionInfo attribute*), 85
add_recipient() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7EnvelopeBuilder method*), 193
add_response() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 22
add_response_by_hash() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 23
add_revoked_certificate() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
add_signer() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7SignatureBuilder method*), 191
additional_certs (*cryptography.hazmat.primitives.serialization.pkcs12.PKCS12KeyAndCertificates attribute*), 190
Admission (*class in cryptography.x509*), 84
admission_authority (*cryptography.x509.Admission attribute*), 84
Admissions (*class in cryptography.x509*), 83
ADMISSIONS (*cryptography.x509.oid.ExtensionOID attribute*), 97
AEAD (*class in cryptography.hazmat.primitives.hpke*), 209
AEADCipherContext (*class in cryptography.hazmat.primitives.ciphers*), 258
AEADDecryptionContext (*class in cryptography.hazmat.primitives.ciphers*), 259
AEADEncryptionContext (*class in cryptography.hazmat.primitives.ciphers*), 259
AES (*class in cryptogra-*

phy.hazmat.primitives.ciphers.algorithms), 249
AES128 (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 249
AES256 (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 249
AES_128_GCM (*cryptography.hazmat.primitives.hpke.AEAD attribute*), 209
AES_256_GCM (*cryptography.hazmat.primitives.hpke.AEAD attribute*), 209
aes_key_unwrap() (*in module cryptography.hazmat.primitives.keywrap*), 235
aes_key_unwrap_with_padding() (*in module cryptography.hazmat.primitives.keywrap*), 236
aes_key_wrap() (*in module cryptography.hazmat.primitives.keywrap*), 235
aes_key_wrap_with_padding() (*in module cryptography.hazmat.primitives.keywrap*), 235
AESCCM (*class in cryptography.hazmat.primitives.ciphers.aead*), 112
AESGCM (*class in cryptography.hazmat.primitives.ciphers.aead*), 102
AESGCMSIV (*class in cryptography.hazmat.primitives.ciphers.aead*), 105
AESOCB3 (*class in cryptography.hazmat.primitives.ciphers.aead*), 107
AESSIV (*class in cryptography.hazmat.primitives.ciphers.aead*), 109
affiliation_changed (*cryptography.x509.ReasonFlags attribute*), 79
AfterFixed (*cryptography.hazmat.primitives.kdf.kbkdf.CounterLocation attribute*), 232
AGNOSTIC (*cryptography.x509.verification.Criticality attribute*), 39
algorithm (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveSignatureAlgorithm attribute*), 147
algorithm (*cryptography.hazmat.primitives.asymmetric.padding.OAEP attribute*), 157
algorithm (*cryptography.hazmat.primitives.hashes.HashContext attribute*), 248
AlreadyFinalized (*class in cryptography.exceptions*), 267
AlreadyUpdated (*class in cryptography.exceptions*), 267
ANONYMOUS (*cryptography.x509.certificate_transparency.SignatureAlgorithm attribute*), 20
ANSIX923 (*class in cryptography.hazmat.primitives.padding*), 262
ANY_EXTENDED_KEY_USAGE (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
ANY_POLICY (*cryptography.x509.oid.CertificatePoliciesOID attribute*), 95
ARC4 (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 282
ARC4 (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 251
Argon2d (*class in cryptography.hazmat.primitives.kdf.argon2*), 211
Argon2i (*class in cryptography.hazmat.primitives.kdf.argon2*), 211
Argon2id (*class in cryptography.hazmat.primitives.kdf.argon2*), 211
as_bytes() (*cryptography.hazmat.asn1.BitString method*), 278
as_datetime() (*cryptography.hazmat.asn1.GeneralizedTime method*), 277
as_datetime() (*cryptography.hazmat.asn1.UTCTime method*), 277
as_list() (*cryptography.hazmat.asn1.SetOf method*), 279
as_str() (*cryptography.hazmat.asn1.IA5String method*), 277
as_str() (*cryptography.hazmat.asn1.PrintableString method*), 277
asymmetric cryptography, **400**
AsymmetricPadding (*class in cryptography.hazmat.primitives.asymmetric.padding*), 156
Attribute (*class in cryptography.x509*), 88
AttributeNotFound (*class in cryptography.x509*), 99
AttributeOID (*class in cryptography.x509.oid*), 98
Attributes (*class in cryptography.x509*), 88
attributes (*cryptography.x509.CertificateSigningRequest attribute*), 55
authenticate_additional_data() (*cryptography.hazmat.primitives.ciphers.AEADCipherContext method*), 259
authentication, **399**
authority (*cryptography.x509.Admissions attribute*), 83
authority_cert_issuer (*cryptography.x509.AuthorityKeyIdentifier attribute*), 72
authority_cert_serial_number (*cryptogra-*

phy.x509.AuthorityKeyIdentifier attribute), 72
AUTHORITY_INFORMATION_ACCESS (*cryptography.x509.oid.ExtensionOID attribute*), 96
AUTHORITY_KEY_IDENTIFIER (*cryptography.x509.oid.ExtensionOID attribute*), 96
AuthorityInformationAccess (*class in cryptography.x509*), 77
AuthorityInformationAccessOID (*class in cryptography.x509.oid*), 95
AuthorityKeyIdentifier (*class in cryptography.x509*), 72
AUTO (*cryptography.hazmat.primitives.asymmetric.padding.PSS attribute*), 156

B

BASIC_CONSTRAINTS (*cryptography.x509.oid.ExtensionOID attribute*), 95
BasicConstraints (*class in cryptography.x509*), 70
BeforeFixed (*cryptography.hazmat.primitives.kdf.kbkdf.CounterLocation attribute*), 232
BestAvailableEncryption (*class in cryptography.hazmat.primitives.serialization*), 200
Binary (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 196
bits, **399**
BitString (*class in cryptography.hazmat.asn1*), 278
BLAKE2b (*class in cryptography.hazmat.primitives.hashes*), 245
BLAKE2s (*class in cryptography.hazmat.primitives.hashes*), 245
block_size (*cryptography.hazmat.primitives.ciphers.BlockCipherAlgorithm attribute*), 260
BlockCipherAlgorithm (*class in cryptography.hazmat.primitives.ciphers*), 260
Blowfish (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 283
BrainpoolP256R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
BRAINPOOLP256R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
BrainpoolP384R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
BRAINPOOLP384R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
BrainpoolP512R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
BRAINPOOLP512R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 152
build() (*cryptography.hazmat.primitives.serialization.KeySerializationEn method*), 201
build() (*cryptography.x509.ocsp.OCSPRequestBuilder method*), 21
build() (*cryptography.x509.RevokedCertificateBuilder method*), 59
build_client_verifier() (*cryptography.x509.verification.PolicyBuilder method*), 37
build_server_verifier() (*cryptography.x509.verification.PolicyBuilder method*), 37
build_unsuccessful() (*cryptography.x509.ocsp.OCSPResponseBuilder class method*), 24
BUNDLE_EID (*cryptography.x509.oid.OtherNameFormOID attribute*), 95
BUNDLE_SECURITY (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 94
BUSINESS_CATEGORY (*cryptography.x509.oid.NameOID attribute*), 90
bytes_eq() (*in module cryptography.hazmat.primitives.constant_time*), 210
bytes-like, **399**

C

ca (*cryptography.x509.BasicConstraints attribute*), 70
ca_compromise (*cryptography.x509.ReasonFlags attribute*), 79
CA_ISSUERS (*cryptography.x509.oid.AuthorityInformationAccessOID attribute*), 95
CA_REPOSITORY (*cryptography.x509.oid.SubjectInformationAccessOID attribute*), 95
calculate_max_pss_salt_length() (*in module cryptography.hazmat.primitives.asymmetric.padding*), 157
Camellia (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 250
CAST5 (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 283
CBC (*class in cryptography.hazmat.primitives.ciphers.modes*), 252
cert (*cryptography.hazmat.primitives.serialization.pkcs12.PKCS12KeyAna attribute*), 189
Certificate (*class in cryptography.x509*), 42

certificate (*cryptography.hazmat.primitives.serialization.pkcs12.PKCS12Certificate attribute*), 189
certificate_hold (*cryptography.x509.ReasonFlags attribute*), 79
CERTIFICATE_ISSUER (*cryptography.x509.oid.CRLEntryExtensionOID attribute*), 97
CERTIFICATE_POLICIES (*cryptography.x509.oid.ExtensionOID attribute*), 96
certificate_status (*cryptography.x509.ocsp.OCSPResponse attribute*), 27
certificate_status (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 31
CERTIFICATE_TRANSPARENCY (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 94
CertificateBuilder (*class in cryptography.x509*), 51
CertificateIssuer (*class in cryptography.x509*), 86
CertificateIssuerPrivateKeyTypes (*in module cryptography.hazmat.primitives.asymmetric.types*), 206
CertificateIssuerPublicKeyTypes (*in module cryptography.hazmat.primitives.asymmetric.types*), 205
CertificatePolicies (*class in cryptography.x509*), 82
CertificatePoliciesOID (*class in cryptography.x509.oid*), 95
CertificatePublicKeyTypes (*in module cryptography.hazmat.primitives.asymmetric.types*), 205
CertificateRevocationList (*class in cryptography.x509*), 48
CertificateRevocationListBuilder (*class in cryptography.x509*), 56
certificates (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
certificates() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 23
CertificateSigningRequest (*class in cryptography.x509*), 54
CertificateSigningRequestBuilder (*class in cryptography.x509*), 60
cessation_of_operation (*cryptography.x509.ReasonFlags attribute*), 79
CFB (*class in cryptography.hazmat.decrepit.ciphers.modes*), 284
CFB (*class in cryptography.hazmat.primitives.ciphers.modes*), 253
CFB8 (*class in cryptography.hazmat.decrepit.ciphers.modes*), 284
CFB8 (*class in cryptography.hazmat.primitives.ciphers.modes*), 253
ChaCha20 (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 250
CHACHA20_POLY1305 (*cryptography.hazmat.primitives.hpke.AEAD attribute*), 210
ChaCha20Poly1305 (*class in cryptography.hazmat.primitives.ciphers.aead*), 100
chain (*cryptography.x509.verification.VerifiedClient attribute*), 34
CHALLENGE_PASSWORD (*cryptography.x509.oid.AttributeOID attribute*), 98
Cipher (*class in cryptography.hazmat.primitives.ciphers*), 248
CipherAlgorithm (*class in cryptography.hazmat.primitives.ciphers*), 259
CipherContext (*class in cryptography.hazmat.primitives.ciphers*), 257
ciphertext, **399**
ciphertext indistinguishability, **399**
CLIENT_AUTH (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
ClientVerifier (*class in cryptography.x509.verification*), 34
CMAC (*class in cryptography.hazmat.primitives.cmac*), 236
Cobblestone128Decryptor (*class in cryptography.cobblestone*), 11
Cobblestone128Encryptor (*class in cryptography.cobblestone*), 10
Cobblestone256Decryptor (*class in cryptography.cobblestone*), 12
Cobblestone256Encryptor (*class in cryptography.cobblestone*), 12
CODE_SIGNING (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
COMMON_NAME (*cryptography.x509.oid.NameOID attribute*), 89
CompressedPoint (*cryptography.hazmat.primitives.serialization.PublicFormat attribute*), 199
ConcatKDFHash (*class in cryptography.hazmat.primitives.kdf.concatkdf*), 219
ConcatKDFHMAC (*class in cryptography.hazmat.primitives.kdf.concatkdf*), 220
content_commitment (*cryptography.x509.KeyUsage attribute*), 68
copy() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSAMuHas method*), 125
copy() (*cryptography.hazmat.primitives.cmac.CMAC

method), 237
copy() (*cryptography.hazmat.primitives.hashes.Hash method*), 243
copy() (*cryptography.hazmat.primitives.hashes.HashContext method*), 248
copy() (*cryptography.hazmat.primitives.hashes.XOFHash method*), 244
copy() (*cryptography.hazmat.primitives.hmac.HMAC method*), 239
CounterLocation (*class in cryptography.hazmat.primitives.kdf.kbkdf*), 232
CounterMode (*cryptography.hazmat.primitives.kdf.kbkdf.Mode attribute*), 232
COUNTRY_NAME (*cryptography.x509.oid.NameOID attribute*), 89
CPS_QUALIFIER (*cryptography.x509.oid.CertificatePoliciesOID attribute*), 95
CPS_USER_NOTICE (*cryptography.x509.oid.CertificatePoliciesOID attribute*), 95
create_unsigned() (*cryptography.x509.CertificateBuilder method*), 54
critical (*cryptography.x509.Extension attribute*), 68
CRITICAL (*cryptography.x509.verification.Criticality attribute*), 39
critical_options (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
Criticality (*class in cryptography.x509.verification*), 39
CRL_DISTRIBUTION_POINTS (*cryptography.x509.oid.ExtensionOID attribute*), 96
crl_issuer (*cryptography.x509.DistributionPoint attribute*), 78
crl_number (*cryptography.x509.CRLNumber attribute*), 80
crl_number (*cryptography.x509.DeltaCRLIndicator attribute*), 77
CRL_NUMBER (*cryptography.x509.oid.ExtensionOID attribute*), 96
CRL_REASON (*cryptography.x509.oid.CRLEntryExtensionOID attribute*), 97
crl_sign (*cryptography.x509.KeyUsage attribute*), 69
CRLDistributionPoints (*class in cryptography.x509*), 78
CRLEntryExtensionOID (*class in cryptography.x509.oid*), 97
CRLNumber (*class in cryptography.x509*), 80
CRLReason (*class in cryptography.x509*), 86
cryptography.hazmat.asn1
module, 274
cryptography.hazmat.backends.openssl.backend (*built-in variable*), 393
cryptography.hazmat.decrepit.ciphers.algorithms
module, 282
cryptography.hazmat.decrepit.ciphers.modes
module, 284
cryptography.hazmat.primitives.asymmetric.dsa
module, 170
cryptography.hazmat.primitives.asymmetric.ec
module, 141
cryptography.hazmat.primitives.asymmetric.padding
module, 156
cryptography.hazmat.primitives.asymmetric.rsa
module, 152
cryptography.hazmat.primitives.ciphers
module, 248
cryptography.hazmat.primitives.ciphers.aead
module, 100
cryptography.hazmat.primitives.ciphers.modes
module, 252
cryptography.hazmat.primitives.hashes
module, 242
cryptography.hazmat.primitives.hpke
module, 206
cryptography.hazmat.primitives.kdf
module, 211
cryptography.hazmat.primitives.keywrap
module, 235
cryptography.hazmat.primitives.padding
module, 261
cryptography.hazmat.primitives.serialization
module, 175
cryptography.x509.verification
module, 33
CTR (*class in cryptography.hazmat.primitives.ciphers.modes*), 252
curve (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivate attribute*), 148
curve (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublic attribute*), 149
curve (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublic attribute*), 144

D

d (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159
data (*cryptography.hazmat.asn1.TLV attribute*), 278
data_encipherment (*cryptography.x509.KeyUsage attribute*), 69
decapsulate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKe method*), 129
decapsulate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey

method), 127
decipher_only (*cryptography.x509.KeyUsage attribute*), 69
decode_der() (*in module cryptography.hazmat.asn1*), 274
decode_dss_signature() (*in module cryptography.hazmat.primitives.asymmetric.utils*), 201
decrypt() (*cryptography.fernet.Fernet method*), 6
decrypt() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey method*), 161
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESCCM method*), 113
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESGCM method*), 104
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESGCMSIV method*), 106
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESOCB3 method*), 108
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESSIV method*), 111
decrypt() (*cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305 method*), 101
decrypt() (*cryptography.hazmat.primitives.hpke.Suite method*), 207
decrypt_at_time() (*cryptography.fernet.Fernet method*), 6
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESCCM method*), 114
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESGCM method*), 104
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESGCMSIV method*), 106
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESOCB3 method*), 109
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESSIV method*), 111
decrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305 method*), 102
decryption, **399**
decryptor() (*cryptography.hazmat.primitives.ciphers.Cipher method*), 249
Default (*class in cryptography.hazmat.asn1*), 281
DELTA_CRL_INDICATOR (*cryptography.x509.oid.ExtensionOID attribute*), 96
DeltaCRLIndicator (*class in cryptography.x509*), 76
DER (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 199
derive() (*cryptography.hazmat.primitives.kdf.argon2.Argon2id method*), 212
derive() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHash method*), 219
derive() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHMA method*), 221
derive() (*cryptography.hazmat.primitives.kdf.hkdf.HKDF method*), 224
derive() (*cryptography.hazmat.primitives.kdf.hkdf.HKDFExpand method*), 226
derive() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFCMAC method*), 230
derive() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFHMAC method*), 228
derive() (*cryptography.hazmat.primitives.kdf.KeyDerivationFunction method*), 234
derive() (*cryptography.hazmat.primitives.kdf.pbkdf2.PBKDF2HMAC method*), 215
derive() (*cryptography.hazmat.primitives.kdf.scrypt.Scrypt method*), 217
derive() (*cryptography.hazmat.primitives.kdf.x963kdf.X963KDF method*), 233
derive_into() (*cryptography.hazmat.primitives.kdf.argon2.Argon2id method*), 213
derive_into() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHash method*), 220
derive_into() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHMAC method*), 222
derive_into() (*cryptography.hazmat.primitives.kdf.hkdf.HKDF method*), 224
derive_into() (*cryptography.hazmat.primitives.kdf.hkdf.HKDFExpand method*), 226
derive_into() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFCMAC method*), 231
derive_into() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFHMAC method*), 228
derive_into() (*cryptography.hazmat.primitives.kdf.KeyDerivationFunction method*), 234
derive_into() (*cryptogra-*

phy.hazmat.primitives.kdf.pbkdf2.PBKDF2HMAC method), 216
derive_into() (*cryptography.hazmat.primitives.kdf.scrypt.Scrypt method*), 218
derive_into() (*cryptography.hazmat.primitives.kdf.x963kdf.X963KDF method*), 233
derive_phc_encoded() (*cryptography.hazmat.primitives.kdf.argon2.Argon2id method*), 213
derive_private_key() (*in module cryptography.hazmat.primitives.asymmetric.ec*), 141
DetachedSignature (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 196
DHParameterNumbers (*class in cryptography.hazmat.primitives.asymmetric.dh*), 168
DHParameters (*class in cryptography.hazmat.primitives.asymmetric.dh*), 166
DHPrivateKey (*class in cryptography.hazmat.primitives.asymmetric.dh*), 167
DHPrivateNumbers (*class in cryptography.hazmat.primitives.asymmetric.dh*), 169
DHPublicKey (*class in cryptography.hazmat.primitives.asymmetric.dh*), 167
DHPublicNumbers (*class in cryptography.hazmat.primitives.asymmetric.dh*), 169
digest (*cryptography.x509.SubjectKeyIdentifier attribute*), 73
DIGEST_LENGTH (*cryptography.hazmat.primitives.asymmetric.padding.PSS attribute*), 156
digest_size (*cryptography.hazmat.primitives.hashes.HashAlgorithm attribute*), 248
digital_signature (*cryptography.x509.KeyUsage attribute*), 68
DirectoryName (*class in cryptography.x509*), 65
DistributionPoint (*class in cryptography.x509*), 78
dmp1 (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159
dmq1 (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159
DN_QUALIFIER (*cryptography.x509.oid.NameOID attribute*), 90
DNS_SRV (*cryptography.x509.oid.OtherNameFormOID attribute*), 94
DNSName (*class in cryptography.x509*), 65
DOMAIN_COMPONENT (*cryptography.x509.oid.NameOID attribute*), 90
dotted_string (*cryptography.x509.ObjectIdentifier attribute*), 64
DSA (*cryptography.x509.certificate_transparency.SignatureAlgorithm attribute*), 20
DSA (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
DSA_WITH_SHA1 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
DSA_WITH_SHA224 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
DSA_WITH_SHA256 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
DSA_WITH_SHA384 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
DSA_WITH_SHA512 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
DSAParameterNumbers (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 171
DSAParameters (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 173
DSAPrivateKey (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 173
DSAPrivateNumbers (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 172
DSAPublicKey (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 174
DSAPublicNumbers (*class in cryptography.hazmat.primitives.asymmetric.dsa*), 172
DuplicateExtension (*class in cryptography.x509*), 99

E

e (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicNumbers attribute*), 158
EC_PUBLIC_KEY (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
ECB (*class in cryptography.hazmat.primitives.ciphers.modes*), 256
ECDH (*class in cryptography.hazmat.primitives.asymmetric.ec*), 144
ECDSA (*class in cryptography.hazmat.primitives.asymmetric.ec*), 141
ECDSA (*cryptography.x509.certificate_transparency.SignatureAlgorithm attribute*), 20
ECDSA_WITH_SHA1 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
ECDSA_WITH_SHA224 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
ECDSA_WITH_SHA256 (*cryptography.x509.oid.SignatureAlgorithmOID at-*

tribute), 91
ECDSA_WITH_SHA384 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
ECDSA_WITH_SHA3_224 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ECDSA_WITH_SHA3_256 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ECDSA_WITH_SHA3_384 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ECDSA_WITH_SHA3_512 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ECDSA_WITH_SHA512 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ED25519 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
ED25519 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
Ed25519PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.ed25519*), 131
Ed25519PublicKey (*class in cryptography.hazmat.primitives.asymmetric.ed25519*), 132
ED448 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
ED448 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
Ed448PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.ed448*), 136
Ed448PublicKey (*class in cryptography.hazmat.primitives.asymmetric.ed448*), 137
EllipticCurve (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
EllipticCurveOID (*class in cryptography.hazmat.primitives.asymmetric.ec*), 151
EllipticCurvePrivateKey (*class in cryptography.hazmat.primitives.asymmetric.ec*), 148
EllipticCurvePrivateNumbers (*class in cryptography.hazmat.primitives.asymmetric.ec*), 143
EllipticCurvePublicKey (*class in cryptography.hazmat.primitives.asymmetric.ec*), 149
EllipticCurvePublicNumbers (*class in cryptography.hazmat.primitives.asymmetric.ec*), 143
EllipticCurveSignatureAlgorithm (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
EMAIL_ADDRESS (*cryptography.x509.oid.NameOID attribute*), 90
EMAIL_PROTECTION (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
enc_length() (*cryptography.hazmat.primitives.hpke.KEM method*), 207
encapsulate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PublicKey method*), 130
encapsulate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PublicKey method*), 128
encipher_only (*cryptography.x509.KeyUsage attribute*), 69
encode_der() (*in module cryptography.hazmat.asn1*), 274
encode_dss_signature() (*in module cryptography.hazmat.primitives.asymmetric.utils*), 201
Encoding (*class in cryptography.hazmat.primitives.serialization*), 199
encrypt() (*cryptography.fernet.Fernet method*), 5
encrypt() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey method*), 162
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESCCM method*), 112
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESGCM method*), 103
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESGCMSIV method*), 105
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESOCB3 method*), 107
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.AESSIV method*), 110
encrypt() (*cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305 method*), 101
encrypt() (*cryptography.hazmat.primitives.hpke.Suite method*), 207
encrypt() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7EnvelopeBuilder method*), 193
encrypt_at_time() (*cryptography.fernet.Fernet method*), 6
encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESCCM method*), 113

encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESGCM method*), 103
encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESGCMSIV method*), 106
encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESOCB3 method*), 108
encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.AESSIV method*), 110
encrypt_into() (*cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305 method*), 101
encryption, **399**
encryption_builder() (*cryptography.hazmat.primitives.serialization.PrivateFormat method*), 198
encryptor() (*cryptography.hazmat.primitives.ciphers.Cipher method*), 249
entry_type (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
exact() (*cryptography.hazmat.asn1.Size static method*), 281
exchange() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey method*), 167
exchange() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey method*), 148
exchange() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey method*), 135
exchange() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey method*), 140
excluded_subtrees (*cryptography.x509.NameConstraints attribute*), 71
Explicit (*class in cryptography.hazmat.asn1*), 280
explicit_text (*cryptography.x509.UserNotice attribute*), 84
ExtendableOutputFunction (*class in cryptography.hazmat.primitives.hashes*), 248
EXTENDED_KEY_USAGE (*cryptography.x509.oid.ExtensionOID attribute*), 96
extended_key_usage (*cryptography.x509.verification.Policy attribute*), 39
ExtendedKeyUsage (*class in cryptography.x509*), 70
ExtendedKeyUsageOID (*class in cryptography.x509.oid*), 93
Extension (*class in cryptography.x509*), 67
extension_bytes (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
extension_policies() (*cryptography.x509.verification.PolicyBuilder method*), 36
ExtensionNotFound (*class in cryptography.x509*), 99
ExtensionOID (*class in cryptography.x509.oid*), 95
ExtensionPolicy (*class in cryptography.x509.verification*), 37
Extensions (*class in cryptography.x509*), 67
extensions (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
extensions (*cryptography.x509.Certificate attribute*), 45
extensions (*cryptography.x509.CertificateRevocationList attribute*), 50
extensions (*cryptography.x509.CertificateSigningRequest attribute*), 55
extensions (*cryptography.x509.ocsp.OCSPRequest attribute*), 25
extensions (*cryptography.x509.ocsp.OCSPResponse attribute*), 30
extensions (*cryptography.x509.RevokedCertificate attribute*), 59
ExtensionType (*class in cryptography.x509*), 68
extract() (*cryptography.hazmat.primitives.kdf.hkdf.HKDF static method*), 223
extract_timestamp() (*cryptography.fernet.Fernet method*), 7

F

Fernet (*class in cryptography.fernet*), 5
finalize() (*cryptography.cobblestone.Cobblestone128Decryptor method*), 12
finalize() (*cryptography.cobblestone.Cobblestone128Encryptor method*), 10
finalize() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSAMuHasher method*), 125
finalize() (*cryptography.hazmat.primitives.ciphers.CipherContext method*), 258
finalize() (*cryptography.hazmat.primitives.cmac.CMAC method*), 238
finalize() (*cryptography.hazmat.primitives.hashes.Hash method*),

243
finalize() (*cryptography.hazmat.primitives.hashes.HashContext method*), 248
finalize() (*cryptography.hazmat.primitives.hmac.HMAC method*), 239
finalize() (*cryptography.hazmat.primitives.padding.PaddingContext method*), 263
finalize() (*cryptography.hazmat.primitives.poly1305.Poly1305 method*), 241
finalize_with_tag() (*cryptography.hazmat.primitives.ciphers.AEADDecryptionContext method*), 259
fingerprint() (*cryptography.x509.Certificate method*), 43
fingerprint() (*cryptography.x509.CertificateRevocationList method*), 48
FRESHEST_CRL (*cryptography.x509.oid.ExtensionOID attribute*), 97
FreshestCRL (*class in cryptography.x509*), 78
friendly_name (*cryptography.hazmat.primitives.serialization.pkcs12.PKCS12Certificate attribute*), 189
from_bytes() (*cryptography.x509.Name class method*), 62
from_encoded_point() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicKey class method*), 150
from_issuer_public_key() (*cryptography.x509.AuthorityKeyIdentifier class method*), 72
from_issuer_subject_key_identifier() (*cryptography.x509.AuthorityKeyIdentifier class method*), 73
from_private_bytes() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey class method*), 131
from_private_bytes() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey class method*), 137
from_private_bytes() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey class method*), 134
from_private_bytes() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey class method*), 139
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PublicKey class method*), 132
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PublicKey class method*), 137
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PublicKey class method*), 117
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PublicKey class method*), 120
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PublicKey class method*), 123
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PublicKey class method*), 129
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PublicKey class method*), 127
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PublicKey class method*), 135
from_public_bytes() (*cryptography.hazmat.primitives.asymmetric.x448.X448PublicKey class method*), 140
from_public_key() (*cryptography.x509.SubjectKeyIdentifier class method*), 74
from_rfc4514_string() (*cryptography.x509.Name class method*), 62
from_seed_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey class method*), 116
from_seed_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey class method*), 119
from_seed_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey class method*), 122
from_seed_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKey class method*), 128
from_seed_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey class method*), 126
full_name (*cryptography.x509.DistributionPoint attribute*), 78
full_name (*cryptography.x509.IssuingDistributionPoint attribute*), 81

G

g (*cryptography.hazmat.primitives.asymmetric.dh.DHParameterNumbers attribute*), 168
g (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameterNumbers attribute*), 172

GCM (*class in cryptography.hazmat.primitives.ciphers.modes*), 254
GeneralizedTime (*class in cryptography.hazmat.asn1*), 277
GeneralName (*class in cryptography.x509*), 65
generate() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey class method*), 131
generate() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey class method*), 136
generate() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey class method*), 116
generate() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey class method*), 119
generate() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey class method*), 122
generate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKey class method*), 128
generate() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey class method*), 126
generate() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey class method*), 134
generate() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey class method*), 139
generate() (*cryptography.hazmat.primitives.twofactor.hotp.HOTP method*), 264
generate() (*cryptography.hazmat.primitives.twofactor.totp.TOTP method*), 266
generate_key() (*cryptography.cobblestone.Cobblestone128Encryptor static method*), 10
generate_key() (*cryptography.fernet.Fernet class method*), 5
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.AESCCM class method*), 112
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.AESGCM class method*), 103
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.AESGCMSIV class method*), 105
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.AESOCB3 class method*), 107
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.AESSIV class method*), 109
generate_key() (*cryptography.hazmat.primitives.ciphers.aead.ChaCha20Poly1305 class method*), 100
generate_parameters() (*in module cryptography.hazmat.primitives.asymmetric.dh*), 166
generate_parameters() (*in module cryptography.hazmat.primitives.asymmetric.dsa*), 170
generate_private_key() (*cryptography.hazmat.primitives.asymmetric.dh.DHParameters method*), 166
generate_private_key() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameters method*), 173
generate_private_key() (*in module cryptography.hazmat.primitives.asymmetric.dsa*), 170
generate_private_key() (*in module cryptography.hazmat.primitives.asymmetric.ec*), 141
generate_private_key() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 152
generate_tag() (*cryptography.hazmat.primitives.poly1305.Poly1305 class method*), 241
GENERATION_QUALIFIER (*cryptography.x509.oid.NameOID attribute*), 89
get_attribute_for_oid() (*cryptography.x509.Attributes method*), 88
get_attributes_for_oid() (*cryptography.x509.Name method*), 62
get_attributes_for_oid() (*cryptography.x509.RelativeDistinguishedName method*), 64
get_curve_for_oid() (*in module cryptography.hazmat.primitives.asymmetric.ec*), 152
get_extension_for_class() (*cryptography.x509.Extensions method*), 67
get_extension_for_oid() (*cryptography.x509.Extensions method*), 67
get_provisioning_uri() (*cryptography.hazmat.primitives.twofactor.hotp.HOTP method*), 265
get_provisioning_uri() (*cryptography.hazmat.primitives.twofactor.totp.TOTP method*), 267
get_revoked_certificate_by_serial_number() (*cryptography.x509.CertificateRevocationList method*), 48
get_values_for_type() (*cryptography.x509.CertificateIssuer method*), 86
get_values_for_type() (*cryptography.x509.IssuerAlternativeName method*),

75
get_values_for_type() (*cryptography.x509.SubjectAlternativeName method*), 74
GIVEN_NAME (*cryptography.x509.oid.NameOID attribute*), 89
GOOD (*cryptography.x509.ocsp.OCSPCertStatus attribute*), 31
group_order (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve attribute*), 147

H

Hash (*class in cryptography.hazmat.primitives.hashes*), 242
HASH (*cryptography.x509.ocsp.OCSPResponderEncoding attribute*), 31
hash() (*cryptography.hazmat.primitives.hashes.Hash static method*), 243
hash_algorithm (*cryptography.x509.ocsp.OCSPRequest attribute*), 25
hash_algorithm (*cryptography.x509.ocsp.OCSPResponse attribute*), 29
hash_algorithm (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 33
HashAlgorithm (*class in cryptography.hazmat.primitives.hashes*), 247
HashContext (*class in cryptography.hazmat.primitives.hashes*), 248
HKDF (*class in cryptography.hazmat.primitives.kdf.hkdf*), 222
HKDF_SHA256 (*cryptography.hazmat.primitives.hpke.KDF attribute*), 209
HKDF_SHA384 (*cryptography.hazmat.primitives.hpke.KDF attribute*), 209
HKDF_SHA512 (*cryptography.hazmat.primitives.hpke.KDF attribute*), 209
HKDFExpand (*class in cryptography.hazmat.primitives.kdf.hkdf*), 225
HMAC (*class in cryptography.hazmat.primitives.hmac*), 238
hmac_hash() (*cryptography.hazmat.primitives.serialization.KeySerializationEncryptionBuilder method*), 200
HOST (*cryptography.hazmat.primitives.serialization.SSHCertificateType attribute*), 184
HOTP (*class in cryptography.hazmat.primitives.twofactor.hotp*), 263
HOTPHashTypes (*in module cryptography.hazmat.primitives.twofactor.hotp*), 263
HW_MODULE_NAME (*cryptography.x509.oid.OtherNameFormOID attribute*), 94

I

IA5String (*class in cryptography.hazmat.asn1*), 277
id (*cryptography.x509.NamingAuthority attribute*), 85
IDEA (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 284
Implicit (*class in cryptography.hazmat.asn1*), 281
indirect_crl (*cryptography.x509.IssuingDistributionPoint attribute*), 81
INHIBIT_ANY_POLICY (*cryptography.x509.oid.ExtensionOID attribute*), 96
inhibit_policy_mapping (*cryptography.x509.PolicyConstraints attribute*), 80
InhibitAnyPolicy (*class in cryptography.x509*), 79
initialization_vector (*cryptography.hazmat.primitives.ciphers.modes.ModeWithInitializationVector attribute*), 260
INITIALS (*cryptography.x509.oid.NameOID attribute*), 89
INTERNAL_ERROR (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 30
invalidity_date (*cryptography.x509.InvalidityDate attribute*), 87
INVALIDITY_DATE (*cryptography.x509.oid.CRLEntryExtensionOID attribute*), 97
invalidity_date_utc (*cryptography.x509.InvalidityDate attribute*), 87
InvalidityDate (*class in cryptography.x509*), 87
InvalidKey (*class in cryptography.exceptions*), 267
InvalidSignature (*class in cryptography.exceptions*), 267
InvalidTag (*class in cryptography.exceptions*), 261
InvalidToken (*class in cryptography.fernet*), 8
InvalidToken (*class in cryptography.hazmat.primitives.twofactor*), 263
InvalidUnwrap (*class in cryptography.hazmat.primitives.keywrap*), 236
InvalidVersion (*class in cryptography.x509*), 99
IPAddress (*class in cryptography.x509*), 66
IPSEC_IKE (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 94
iqmp (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159

is_signature_valid (*cryptography.x509.CertificateSigningRequest attribute*), 56
is_signature_valid() (*cryptography.x509.CertificateRevocationList method*), 51
issuer (*cryptography.x509.Certificate attribute*), 44
issuer (*cryptography.x509.CertificateRevocationList attribute*), 49
ISSUER_ALTERNATIVE_NAME (*cryptography.x509.oid.ExtensionOID attribute*), 95
issuer_key_hash (*cryptography.x509.ocsp.OCSPRequest attribute*), 25
issuer_key_hash (*cryptography.x509.ocsp.OCSPResponse attribute*), 29
issuer_key_hash (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 33
issuer_name() (*cryptography.x509.CertificateBuilder method*), 52
issuer_name() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
issuer_name_hash (*cryptography.x509.ocsp.OCSPRequest attribute*), 25
issuer_name_hash (*cryptography.x509.ocsp.OCSPResponse attribute*), 29
issuer_name_hash (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 33
IssuerAlternativeName (*class in cryptography.x509*), 74
ISSUING_DISTRIBUTION_POINT (*cryptography.x509.oid.ExtensionOID attribute*), 97
IssuingDistributionPoint (*class in cryptography.x509*), 81

J

JURISDICTION_COUNTRY_NAME (*cryptography.x509.oid.NameOID attribute*), 90
JURISDICTION_LOCALITY_NAME (*cryptography.x509.oid.NameOID attribute*), 90
JURISDICTION_STATE_OR_PROVINCE_NAME (*cryptography.x509.oid.NameOID attribute*), 90

K

KBKDFCMAC (*class in cryptography.hazmat.primitives.kdf.kbkdf*), 229
KBKDFHMAC (*class in cryptography.hazmat.primitives.kdf.kbkdf*), 227
KDF (*class in cryptography.hazmat.primitives.hpke*), 209
kdf_rounds() (*cryptography.hazmat.primitives.serialization.KeySerializationEncryptionBu method*), 200
KEM (*class in cryptography.hazmat.primitives.hpke*), 207
KERBEROS_PKINIT_KDC (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 94
key, **399**
key (*cryptography.hazmat.primitives.serialization.pkcs12.PKCS12KeyAndC attribute*), 189
key_agreement (*cryptography.x509.KeyUsage attribute*), 69
key_cert_algorithm() (*cryptography.hazmat.primitives.serialization.KeySerializationEncryptionBu method*), 200
key_cert_sign (*cryptography.x509.KeyUsage attribute*), 69
key_compromise (*cryptography.x509.ReasonFlags attribute*), 79
key_encipherment (*cryptography.x509.KeyUsage attribute*), 69
key_id (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
key_id() (*cryptography.hazmat.primitives.serialization.SSHCertificateBui method*), 185
key_identifier (*cryptography.x509.AuthorityKeyIdentifier attribute*), 72
key_identifier (*cryptography.x509.SubjectKeyIdentifier attribute*), 73
key_size (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey attribute*), 167
key_size (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicKey attribute*), 167
key_size (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey attribute*), 173
key_size (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey attribute*), 174
key_size (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve attribute*), 147
key_size (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePri attribute*), 148
key_size (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePu attribute*), 150
key_size (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey attribute*), 161
key_size (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey attribute*), 162
key_size (*cryptography.hazmat.primitives.ciphers.CipherAlgorithm attribute*), 260
KEY_USAGE (*cryptography.x509.oid.ExtensionOID attribute*), 95
KeyDerivationFunction (*class in cryptogra-*

phy.hazmat.primitives.kdf), 234
KeySerializationEncryption (*class in cryptography.hazmat.primitives.serialization*), 200
KeySerializationEncryptionBuilder (*class in cryptography.hazmat.primitives.serialization*), 200
KeyUsage (*class in cryptography.x509*), 68

L

last_update (*cryptography.x509.CertificateRevocationList attribute*), 50
last_update() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
last_update_utc (*cryptography.x509.CertificateRevocationList attribute*), 50
load_der_ocsp_request() (*in module cryptography.x509.ocsp*), 20
load_der_ocsp_response() (*in module cryptography.x509.ocsp*), 22
load_der_parameters() (*in module cryptography.hazmat.primitives.serialization*), 179
load_der_pkcs7_certificates() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 190
load_der_private_key() (*in module cryptography.hazmat.primitives.serialization*), 178
load_der_public_key() (*in module cryptography.hazmat.primitives.serialization*), 178
load_der_x509_certificate() (*in module cryptography.x509*), 41
load_der_x509_crl() (*in module cryptography.x509*), 42
load_der_x509_csr() (*in module cryptography.x509*), 42
load_key_and_certificates() (*in module cryptography.hazmat.primitives.serialization.pkcs12*), 186
load_pem_parameters() (*in module cryptography.hazmat.primitives.serialization*), 177
load_pem_pkcs7_certificates() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 190
load_pem_private_key() (*in module cryptography.hazmat.primitives.serialization*), 176
load_pem_public_key() (*in module cryptography.hazmat.primitives.serialization*), 177
load_pem_x509_certificate() (*in module cryptography.x509*), 40
load_pem_x509_certificates() (*in module cryptography.x509*), 41
load_pem_x509_crl() (*in module cryptography.x509*), 41
load_pem_x509_csr() (*in module cryptography.x509*), 42
load_pkcs12() (*in module cryptography.hazmat.primitives.serialization.pkcs12*), 187
load_ssh_private_key() (*in module cryptography.hazmat.primitives.serialization*), 181
load_ssh_public_identity() (*in module cryptography.hazmat.primitives.serialization*), 182
load_ssh_public_key() (*in module cryptography.hazmat.primitives.serialization*), 180
LOCALITY_NAME (*cryptography.x509.oid.NameOID attribute*), 89
log_id (*cryptography.x509.certificate_transparency.SignedCertificateTime attribute*), 19
LogEntryType (*class in cryptography.x509.certificate_transparency*), 20

M

major_version (*cryptography.x509.MSCertificateTemplate attribute*), 82
MALFORMED_REQUEST (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 30
max_chain_depth (*cryptography.x509.verification.Policy attribute*), 39
max_chain_depth() (*cryptography.x509.verification.PolicyBuilder method*), 36
MAX_LENGTH (*cryptography.hazmat.primitives.asymmetric.padding.PSS attribute*), 156
may_be_present() (*cryptography.x509.verification.ExtensionPolicy method*), 38
MaybeExtensionValidatorCallback (*in module cryptography.x509.verification*), 40
MD5 (*class in cryptography.hazmat.primitives.hashes*), 246
MGF (*class in cryptography.hazmat.primitives.asymmetric.padding*), 158
mgf (*cryptography.hazmat.primitives.asymmetric.padding.OAEP attribute*), 157
mgf (*cryptography.hazmat.primitives.asymmetric.padding.PSS attribute*), 156
MGF1 (*class in cryptography.hazmat.primitives.asymmetric.padding*), 158

MiddleFixed (*cryptography.hazmat.primitives.kdf.kbkdf.CounterLocation attribute*), 232
minimum_rsa_modulus (*cryptography.x509.verification.Policy attribute*), 40
minor_version (*cryptography.x509.MSCertificateTemplate attribute*), 82
ML_DSA_44 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
ML_DSA_44 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 92
ML_DSA_65 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
ML_DSA_65 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 93
ML_DSA_87 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 99
ML_DSA_87 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 93
MLDSA44PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 116
MLDSA44PublicKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 117
MLDSA65PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 119
MLDSA65PublicKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 120
MLDSA87PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 122
MLDSA87PublicKey (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 123
MLDSAMuHasher (*class in cryptography.hazmat.primitives.asymmetric.mldsa*), 125
MLKEM1024 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
MLKEM1024_P384 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
MLKEM1024P384PrivateKey (*class in cryptography.hazmat.primitives.hpke*), 209
MLKEM1024P384PublicKey (*class in cryptography.hazmat.primitives.hpke*), 209
MLKEM1024PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.mlkem*), 128
MLKEM1024PublicKey (*class in cryptography.hazmat.primitives.asymmetric.mlkem*), 129
MLKEM768 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
MLKEM768_X25519 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
MLKEM768PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.mlkem*), 126
MLKEM768PublicKey (*class in cryptography.hazmat.primitives.asymmetric.mlkem*), 127
MLKEM768X25519PrivateKey (*class in cryptography.hazmat.primitives.hpke*), 208
MLKEM768X25519PublicKey (*class in cryptography.hazmat.primitives.hpke*), 208
Mode (*class in cryptography.hazmat.primitives.ciphers.modes*), 260
Mode (*class in cryptography.hazmat.primitives.kdf.kbkdf*), 231
ModeWithAuthenticationTag (*class in cryptography.hazmat.primitives.ciphers.modes*), 261
ModeWithInitializationVector (*class in cryptography.hazmat.primitives.ciphers.modes*), 260
ModeWithNonce (*class in cryptography.hazmat.primitives.ciphers.modes*), 260
ModeWithTweak (*class in cryptography.hazmat.primitives.ciphers.modes*), 261
module
- cryptography.hazmat.asn1, 274
- cryptography.hazmat.decrepit.ciphers.algorithms, 282
- cryptography.hazmat.decrepit.ciphers.modes, 284
- cryptography.hazmat.primitives.asymmetric.dsa, 170
- cryptography.hazmat.primitives.asymmetric.ec, 141
- cryptography.hazmat.primitives.asymmetric.padding, 156
- cryptography.hazmat.primitives.asymmetric.rsa, 152
- cryptography.hazmat.primitives.ciphers, 248
- cryptography.hazmat.primitives.ciphers.aead, 100
- cryptography.hazmat.primitives.ciphers.modes,

252
cryptography.hazmat.primitives.hashes, 242
cryptography.hazmat.primitives.hpke, 206
cryptography.hazmat.primitives.kdf, 211
cryptography.hazmat.primitives.keywrap, 235
cryptography.hazmat.primitives.padding, 261
cryptography.hazmat.primitives.serialization, 175
cryptography.x509.verification, 33
MS_CERTIFICATE_TEMPLATE (*cryptography.x509.oid.ExtensionOID attribute*), 97
MSCertificateTemplate (*class in cryptography.x509*), 82
MultiFernet (*class in cryptography.fernet*), 7

N

n (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicNumbers attribute*), 158
NAI_REALM (*cryptography.x509.oid.OtherNameFormOID attribute*), 94
Name (*class in cryptography.x509*), 61
name (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve attribute*), 147
name (*cryptography.hazmat.primitives.asymmetric.padding.AsymmetricPadding attribute*), 156
name (*cryptography.hazmat.primitives.ciphers.CipherAlgorithm attribute*), 259
name (*cryptography.hazmat.primitives.ciphers.modes.Mode attribute*), 260
name (*cryptography.hazmat.primitives.hashes.HashAlgorithm attribute*), 247
NAME (*cryptography.x509.ocsp.OCSPResponderEncoding attribute*), 31
NAME_CONSTRAINTS (*cryptography.x509.oid.ExtensionOID attribute*), 96
NameAttribute (*class in cryptography.x509*), 63
NameConstraints (*class in cryptography.x509*), 71
NameOID (*class in cryptography.x509.oid*), 89
naming_authority (*cryptography.x509.Admission attribute*), 84
naming_authority (*cryptography.x509.ProfessionInfo attribute*), 85
NamingAuthority (*class in cryptography.x509*), 85
next_update (*cryptography.x509.CertificateRevocationList attribute*), 49
next_update (*cryptography.x509.ocsp.OCSPResponse attribute*), 29
next_update (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 32
next_update() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
next_update_utc (*cryptography.x509.CertificateRevocationList attribute*), 49
next_update_utc (*cryptography.x509.ocsp.OCSPResponse attribute*), 29
next_update_utc (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 33
NoAttributes (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 197
NoCapabilities (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 196
NoCerts (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 197
NoDigestInfo (*class in cryptography.hazmat.primitives.asymmetric.utils*), 201
NoEncryption (*class in cryptography.hazmat.primitives.serialization*), 200
NON_CRITICAL (*cryptography.x509.verification.Criticality attribute*), 39
nonce, **400**
nonce (*cryptography.hazmat.primitives.ciphers.modes.ModeWithNonce attribute*), 260
nonce (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 182
nonce (*cryptography.x509.OCSPAcceptableResponses attribute*), 88
nonce (*cryptography.x509.OCSPNonce attribute*), 87
NONCE (*cryptography.x509.oid.OCSPExtensionOID attribute*), 98
not_after (*cryptography.x509.PrivateKeyUsagePeriod attribute*), 76
not_before (*cryptography.x509.PrivateKeyUsagePeriod attribute*), 76
not_valid_after (*cryptography.x509.Certificate attribute*), 44
not_valid_after() (*cryptography.x509.CertificateBuilder method*), 53
not_valid_after_utc (*cryptography.x509.Certificate attribute*), 44
not_valid_before (*cryptography.x509.Certificate attribute*), 43
not_valid_before() (*cryptography.x509.CertificateBuilder method*), 53

not_valid_before_utc (*cryptography.x509.Certificate attribute*), 44
notice_numbers (*cryptography.x509.NoticeReference attribute*), 84
notice_reference (*cryptography.x509.UserNotice attribute*), 84
NoticeReference (*class in cryptography.x509*), 84
NotYetFinalized (*class in cryptography.exceptions*), 267
Null (*class in cryptography.hazmat.asn1*), 278

O

OAEP (*class in cryptography.hazmat.primitives.asymmetric.padding*), 156
ObjectIdentifier (*class in cryptography.x509*), 64
OCSP (*cryptography.x509.oid.AuthorityInformationAccessOID attribute*), 95
OCSP_NO_CHECK (*cryptography.x509.oid.ExtensionOID attribute*), 96
OCSP_SIGNING (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
OCSPAcceptableResponses (*class in cryptography.x509*), 87
OCSPCertStatus (*class in cryptography.x509.ocsp*), 31
OCSPExtensionOID (*class in cryptography.x509.oid*), 97
OCSPNoCheck (*class in cryptography.x509*), 70
OCSPNonce (*class in cryptography.x509*), 87
OCSPRequest (*class in cryptography.x509.ocsp*), 25
OCSPRequestBuilder (*class in cryptography.x509.ocsp*), 21
OCSPResponderEncoding (*class in cryptography.x509.ocsp*), 31
OCSPResponse (*class in cryptography.x509.ocsp*), 25
OCSPResponseBuilder (*class in cryptography.x509.ocsp*), 22
OCSPResponseStatus (*class in cryptography.x509.ocsp*), 30
OCSPSingleResponse (*class in cryptography.x509.ocsp*), 31
OFB (*class in cryptography.hazmat.decrepit.ciphers.modes*), 284
OFB (*class in cryptography.hazmat.primitives.ciphers.modes*), 253
oid (*cryptography.x509.Admissions attribute*), 83
oid (*cryptography.x509.Attribute attribute*), 88
oid (*cryptography.x509.AttributeNotFound attribute*), 99
oid (*cryptography.x509.AuthorityInformationAccess attribute*), 77
oid (*cryptography.x509.AuthorityKeyIdentifier attribute*), 72
oid (*cryptography.x509.BasicConstraints attribute*), 70
oid (*cryptography.x509.CertificateIssuer attribute*), 86
oid (*cryptography.x509.CertificatePolicies attribute*), 83
oid (*cryptography.x509.CRLDistributionPoints attribute*), 78
oid (*cryptography.x509.CRLNumber attribute*), 80
oid (*cryptography.x509.CRLReason attribute*), 86
oid (*cryptography.x509.DeltaCRLIndicator attribute*), 76
oid (*cryptography.x509.DuplicateExtension attribute*), 99
oid (*cryptography.x509.ExtendedKeyUsage attribute*), 70
oid (*cryptography.x509.Extension attribute*), 67
oid (*cryptography.x509.ExtensionNotFound attribute*), 99
oid (*cryptography.x509.ExtensionType attribute*), 68
oid (*cryptography.x509.FreshestCRL attribute*), 78
oid (*cryptography.x509.InhibitAnyPolicy attribute*), 79
oid (*cryptography.x509.InvalidityDate attribute*), 87
oid (*cryptography.x509.IssuerAlternativeName attribute*), 75
oid (*cryptography.x509.IssuingDistributionPoint attribute*), 81
oid (*cryptography.x509.KeyUsage attribute*), 68
oid (*cryptography.x509.MSCertificateTemplate attribute*), 82
oid (*cryptography.x509.NameAttribute attribute*), 63
oid (*cryptography.x509.NameConstraints attribute*), 71
oid (*cryptography.x509.OCSPAcceptableResponses attribute*), 88
oid (*cryptography.x509.OCSPNoCheck attribute*), 70
oid (*cryptography.x509.OCSPNonce attribute*), 87
oid (*cryptography.x509.PolicyConstraints attribute*), 80
oid (*cryptography.x509.PrecertificateSignedCertificateTimestamps attribute*), 75
oid (*cryptography.x509.PrecertPoison attribute*), 75
oid (*cryptography.x509.PrivateKeyUsagePeriod attribute*), 76
oid (*cryptography.x509.SignedCertificateTimestamps attribute*), 76
oid (*cryptography.x509.SubjectAlternativeName attribute*), 74
oid (*cryptography.x509.SubjectInformationAccess attribute*), 77
oid (*cryptography.x509.SubjectKeyIdentifier attribute*), 73
oid (*cryptography.x509.TLSFeature attribute*), 71
oid (*cryptography.x509.UnrecognizedExtension attribute*), 82
only_contains_attribute_certs (*cryptography.x509.IssuingDistributionPoint attribute*), 81
only_contains_ca_certs (*cryptography.x509.IssuingDistributionPoint attribute*), 81
only_contains_user_certs (*cryptography.x509.IssuingDistributionPoint attribute*),

81
only_some_reasons (*cryptography.x509.IssuingDistributionPoint attribute*), 81
opaque key, **400**
OpenSSH (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 199
OpenSSH (*cryptography.hazmat.primitives.serialization.PrivateFormat attribute*), 197
OpenSSH (*cryptography.hazmat.primitives.serialization.PublicFormat attribute*), 199
openssl_version_number(), 394
openssl_version_text() (*cryptography.hazmat.backends.openssl method*), 393
organization (*cryptography.x509.NoticeReference attribute*), 84
ORGANIZATION_IDENTIFIER (*cryptography.x509.oid.NameOID attribute*), 89
ORGANIZATION_NAME (*cryptography.x509.oid.NameOID attribute*), 89
ORGANIZATIONAL_UNIT_NAME (*cryptography.x509.oid.NameOID attribute*), 89
OtherName (*class in cryptography.x509*), 66
OtherNameFormOID (*class in cryptography.x509.oid*), 94

P

p (*cryptography.hazmat.primitives.asymmetric.dh.DHParameterNumbers attribute*), 168
p (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameterNumbers attribute*), 171
p (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159
P256 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
P384 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
P521 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
padder() (*cryptography.hazmat.primitives.padding.ANSIX923 method*), 262
padder() (*cryptography.hazmat.primitives.padding.PKCS7 method*), 262
padding_bits() (*cryptography.hazmat.asn1.BitString method*), 278
PaddingContext (*class in cryptography.hazmat.primitives.padding*), 262
parameter_bytes() (*cryptography.hazmat.primitives.asymmetric.dh.DHParameters method*), 166
parameter_numbers (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicNumbers attribute*), 169
parameter_numbers (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicNumbers attribute*), 172
parameter_numbers() (*cryptography.hazmat.primitives.asymmetric.dh.DHParameters method*), 166
parameter_numbers() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameters method*), 173
ParameterFormat (*class in cryptography.hazmat.primitives.serialization*), 199
parameters() (*cryptography.hazmat.primitives.asymmetric.dh.DHParameterNumbers method*), 168
parameters() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey method*), 167
parameters() (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicKey method*), 167
parameters() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameterNumbers method*), 172
parameters() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey method*), 173
parameters() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey method*), 174
parse() (*cryptography.hazmat.asn1.TLV method*), 278
parsed_version (*cryptography.x509.InvalidVersion attribute*), 99
path_length (*cryptography.x509.BasicConstraints attribute*), 70
PBES (*class in cryptography.hazmat.primitives.serialization.pkcs12*), 190
PBESv1SHA1And3KeyTripleDESCBC (*cryptography.hazmat.primitives.serialization.pkcs12.PBES attribute*), 190
PBESv2SHA256AndAES256CBC (*cryptography.hazmat.primitives.serialization.pkcs12.PBES attribute*), 190
PBKDF2HMAC (*class in cryptography.hazmat.primitives.kdf.pbkdf2*), 214
PEM (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 199
PERMANENT_IDENTIFIER (*cryptography.x509.oid.OtherNameFormOID attribute*), 94
permit_all() (*cryptography.x509.verification.ExtensionPolicy static method*), 38
permitted_subtrees (*cryptography.x509.NameConstraints attribute*), 71
PKCS1 (*cryptography.hazmat.primitives.serialization.PublicFormat*

attribute), 198
PKCS12 (*cryptography.hazmat.primitives.serialization.PrivateFormat attribute*), 197
PKCS12Certificate (*class in cryptography.hazmat.primitives.serialization.pkcs12*), 189
PKCS12KeyAndCertificates (*class in cryptography.hazmat.primitives.serialization.pkcs12*), 189
PKCS12PrivateKeyTypes (*in module cryptography.hazmat.primitives.serialization.pkcs12*), 186
PKCS1v15 (*class in cryptography.hazmat.primitives.asymmetric.padding*), 157
PKCS3 (*cryptography.hazmat.primitives.serialization.ParameterFormat attribute*), 199
PKCS7 (*class in cryptography.hazmat.primitives.padding*), 261
pkcs7_decrypt_der() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 193
pkcs7_decrypt_pem() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 194
pkcs7_decrypt_smime() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 195
PKCS7EnvelopeBuilder (*class in cryptography.hazmat.primitives.serialization.pkcs7*), 192
PKCS7HashTypes (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 190
PKCS7Options (*class in cryptography.hazmat.primitives.serialization.pkcs7*), 196
PKCS7PrivateKeyTypes (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 190
PKCS7SignatureBuilder (*class in cryptography.hazmat.primitives.serialization.pkcs7*), 191
PKCS8 (*cryptography.hazmat.primitives.serialization.PrivateFormat attribute*), 197
plaintext, **400**
Policy (*class in cryptography.x509.verification*), 39
policy (*cryptography.x509.verification.ClientVerifier attribute*), 34
policy (*cryptography.x509.verification.ServerVerifier attribute*), 35
POLICY_CONSTRAINTS (*cryptography.x509.oid.ExtensionOID attribute*), 97
policy_identifier (*cryptography.x509.PolicyInformation attribute*), 83
POLICY_MAPPINGS (*cryptography.x509.oid.ExtensionOID attribute*), 97
policy_qualifiers (*cryptography.x509.PolicyInformation attribute*), 83
PolicyBuilder (*class in cryptography.x509.verification*), 36
PolicyConstraints (*class in cryptography.x509*), 80
PolicyInformation (*class in cryptography.x509*), 83
Poly1305 (*class in cryptography.hazmat.primitives.poly1305*), 240
POSTAL_ADDRESS (*cryptography.x509.oid.NameOID attribute*), 90
POSTAL_CODE (*cryptography.x509.oid.NameOID attribute*), 90
PRE_CERTIFICATE (*cryptography.x509.certificate_transparency.LogEntryType attribute*), 20
PRECERT_POISON (*cryptography.x509.oid.ExtensionOID attribute*), 97
PRECERT_SIGNED_CERTIFICATE_TIMESTAMPS (*cryptography.x509.oid.ExtensionOID attribute*), 96
PrecertificateSignedCertificateTimestamps (*class in cryptography.x509*), 75
PrecertPoison (*class in cryptography.x509*), 75
Prehashed (*class in cryptography.hazmat.primitives.asymmetric.utils*), 201
PresentExtensionValidatorCallback (*in module cryptography.x509.verification*), 40
PrintableString (*class in cryptography.hazmat.asn1*), 277
private_key, **400**
private_bytes() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey method*), 167
private_bytes() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey method*), 174
private_bytes() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey method*), 149
private_bytes() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey method*), 132
private_bytes() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey method*), 137
private_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey method*), 117
private_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey

method), 120
private_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey method*), 123
private_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKey method*), 129
private_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey method*), 127
private_bytes() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey method*), 162
private_bytes() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey method*), 135
private_bytes() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey method*), 140
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey method*), 132
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey method*), 137
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey method*), 117
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey method*), 120
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey method*), 123
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKey method*), 129
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey method*), 127
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey method*), 135
private_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey method*), 140
private_key() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateNumbers method*), 169
private_key() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateNumbers method*), 173
private_key() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateNumbers method*), 143
private_key() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers method*), 159
PRIVATE_KEY_USAGE_PERIOD (*cryptography.x509.oid.ExtensionOID attribute*), 97
private_numbers() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey method*), 167
private_numbers() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey method*), 174
private_numbers() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey method*), 149
private_numbers() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey method*), 161
private_value (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateNumbers attribute*), 143
PrivateFormat (*class in cryptography.hazmat.primitives.serialization*), 197
PrivateKeyTypes (*in module cryptography.hazmat.primitives.asymmetric.types*), 205
PrivateKeyUsagePeriod (*class in cryptography.x509*), 75
privilege_withdrawn (*cryptography.x509.ReasonFlags attribute*), 79
produced_at (*cryptography.x509.ocsp.OCSPResponse attribute*), 27
produced_at_utc (*cryptography.x509.ocsp.OCSPResponse attribute*), 27
professionInfos (*cryptography.x509.Admission attribute*), 85
profession_items (*cryptography.x509.ProfessionInfo attribute*), 85
profession_oids (*cryptography.x509.ProfessionInfo attribute*), 85
ProfessionInfo (*class in cryptography.x509*), 85
PSEUDONYM (*cryptography.x509.oid.NameOID attribute*), 90
PSS (*class in cryptography.hazmat.primitives.asymmetric.padding*), 156
public key, **400**
public_bytes() (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicKey method*), 168
public_bytes() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey method*), 174

public_bytes() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicKey method*), 149
public_bytes() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PublicKey method*), 132
public_bytes() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PublicKey method*), 137
public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PublicKey method*), 118
public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PublicKey method*), 121
public_bytes() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PublicKey method*), 124
public_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PublicKey method*), 130
public_bytes() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PublicKey method*), 128
public_bytes() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey method*), 162
public_bytes() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PublicKey method*), 136
public_bytes() (*cryptography.hazmat.primitives.asymmetric.x448.X448PublicKey method*), 140
public_bytes() (*cryptography.hazmat.primitives.serialization.SSHCertificate method*), 184
public_bytes() (*cryptography.x509.Certificate method*), 48
public_bytes() (*cryptography.x509.CertificateRevocationList method*), 51
public_bytes() (*cryptography.x509.CertificateSigningRequest method*), 55
public_bytes() (*cryptography.x509.ExtensionType method*), 68
public_bytes() (*cryptography.x509.Name method*), 62
public_bytes() (*cryptography.x509.ocsp.OCSPRequest method*), 25
public_bytes() (*cryptography.x509.ocsp.OCSPResponse method*), 30
public_bytes() (*cryptography.x509.PrivateKeyUsagePeriod method*), 76
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PublicKey method*), 133
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PublicKey method*), 138
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PublicKey method*), 118
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PublicKey method*), 121
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PublicKey method*), 124
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PublicKey method*), 130
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PublicKey method*), 128
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PublicKey method*), 136
public_bytes_raw() (*cryptography.hazmat.primitives.asymmetric.x448.X448PublicKey method*), 141
public_key() (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateKey method*), 167
public_key() (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicNumbers method*), 169
public_key() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey method*), 173
public_key() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicNumbers method*), 172
public_key() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey method*), 148
public_key() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers method*), 144
public_key() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey method*), 131
public_key() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey method*), 137
public_key() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey*

method), 116
public_key() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey method*), 119
public_key() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey method*), 122
public_key() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM1024PrivateKey method*), 129
public_key() (*cryptography.hazmat.primitives.asymmetric.mlkem.MLKEM768PrivateKey method*), 126
public_key() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey method*), 161
public_key() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicNumbers method*), 158
public_key() (*cryptography.hazmat.primitives.asymmetric.x25519.X25519PrivateKey method*), 135
public_key() (*cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey method*), 139
public_key() (*cryptography.hazmat.primitives.hpke.MLKEM1024P384PrivateKey method*), 209
public_key() (*cryptography.hazmat.primitives.hpke.MLKEM768X25519PrivateKey method*), 208
public_key() (*cryptography.hazmat.primitives.serialization.SSHCertificate method*), 182
public_key() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 185
public_key() (*cryptography.x509.Certificate method*), 43
public_key() (*cryptography.x509.CertificateBuilder method*), 52
public_key() (*cryptography.x509.CertificateSigningRequest method*), 54
public_key_algorithm_oid (*cryptography.x509.Certificate attribute*), 43
public_key_algorithm_oid (*cryptography.x509.CertificateSigningRequest attribute*), 54
public_numbers (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateNumbers attribute*), 169
public_numbers (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateNumbers attribute*), 172
public_numbers (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateNumbers attribute*), 143
public_numbers (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 158
public_numbers() (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicKey method*), 168
public_numbers() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey method*), 174
public_numbers() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicKey method*), 149
public_numbers() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey method*), 162
public-key cryptography, **400**
PublicFormat (*class in cryptography.hazmat.primitives.serialization*), 198
PublicKeyAlgorithmOID (*class in cryptography.x509.oid*), 98
PublicKeyTypes (*in module cryptography.hazmat.primitives.asymmetric.types*), 205
Python Enhancement Proposals
PEP 484, 308
PEP 8, 330

Q

q (*cryptography.hazmat.primitives.asymmetric.dh.DHParameterNumbers attribute*), 168
q (*cryptography.hazmat.primitives.asymmetric.dsa.DSAParameterNumbers attribute*), 172
q (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateNumbers attribute*), 159

R

random_serial_number() (*in module cryptography.x509*), 99
Raw (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 200
Raw (*cryptography.hazmat.primitives.serialization.PrivateFormat attribute*), 197
Raw (*cryptography.hazmat.primitives.serialization.PublicFormat attribute*), 199
rdns (*cryptography.x509.Name attribute*), 62
reason (*cryptography.x509.CRLReason attribute*), 86
ReasonFlags (*class in cryptography.x509*), 79
reasons (*cryptography.x509.DistributionPoint attribute*), 79

recover_data_from_signature() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey method*), 163
RegisteredID (*class in cryptography.x509*), 66
registration_number (*cryptography.x509.ProfessionInfo attribute*), 85
relative_name (*cryptography.x509.DistributionPoint attribute*), 78
relative_name (*cryptography.x509.IssuingDistributionPoint attribute*), 82
RelativeDistinguishedName (*class in cryptography.x509*), 64
remove_from_crl (*cryptography.x509.ReasonFlags attribute*), 79
require_explicit_policy (*cryptography.x509.PolicyConstraints attribute*), 80
require_not_present() (*cryptography.x509.verification.ExtensionPolicy method*), 38
require_present() (*cryptography.x509.verification.ExtensionPolicy method*), 38
reset_nonce() (*cryptography.hazmat.primitives.ciphers.CipherContext method*), 258
responder_id() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 23
responder_key_hash (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
responder_name (*cryptography.x509.ocsp.OCSPResponse attribute*), 27
response_status (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
responses (*cryptography.x509.ocsp.OCSPResponse attribute*), 30
revocation_date (*cryptography.x509.RevokedCertificate attribute*), 58
revocation_date() (*cryptography.x509.RevokedCertificateBuilder method*), 59
revocation_date_utc (*cryptography.x509.RevokedCertificate attribute*), 58
revocation_reason (*cryptography.x509.ocsp.OCSPResponse attribute*), 28
revocation_reason (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 32
revocation_time (*cryptography.x509.ocsp.OCSPResponse attribute*), 27
revocation_time (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 31
revocation_time_utc (*cryptography.x509.ocsp.OCSPResponse attribute*), 28
revocation_time_utc (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 32
REVOKED (*cryptography.x509.ocsp.OCSPCertStatus attribute*), 31
RevokedCertificate (*class in cryptography.x509*), 58
RevokedCertificateBuilder (*class in cryptography.x509*), 59
RFC
    RFC 1321, 367
    RFC 1421, 328
    RFC 2104, 238
    RFC 2144, 368
    RFC 2202, 367
    RFC 2256, 90
    RFC 2286, 367
    RFC 2315, 190, 297
    RFC 2409, 163
    RFC 2560, 306
    RFC 2632, 191–195
    RFC 2633, 298
    RFC 2818, 89
    RFC 2986, 56, 303
    RFC 3218, 290
    RFC 3279, 142, 148, 171, 201
    RFC 3370, 298
    RFC 3394, 235
    RFC 3447, 156, 157
    RFC 3490, 356
    RFC 3526, 353
    RFC 3610, 112
    RFC 3686, 368
    RFC 4043, 94
    RFC 4055, 91
    RFC 4108, 94
    RFC 4196, 368
    RFC 4226, 263, 264, 368
    RFC 4231, 367
    RFC 4253, 179, 180, 323
    RFC 4269, 283, 368
    RFC 4346, 163
    RFC 4493, 236
    RFC 4514, 62–64, 303, 305, 310, 311
    RFC 4519, 90
    RFC 4556, 94
    RFC 4945, 94

RFC 4985, 94
RFC 5114, 353
RFC 5246, 20
RFC 5280, 12, 46, 51, 68, 71, 72, 74, 80, 81, 93, 177, 178, 296, 298, 303, 306, 355, 359
RFC 5280 Section 4.2.1.1, 72
RFC 5280 Section 4.2.1.6, 100
RFC 5297, 109
RFC 5639, 147
RFC 5649, 235, 236
RFC 5869, 223, 224, 367, 387
RFC 5895, 356
RFC 6066, 71
RFC 6070, 367
RFC 6229, 368, 373
RFC 6238, 263, 266, 368
RFC 6960, 20, 306
RFC 6961, 71
RFC 6962, 18, 47, 48, 94
RFC 6979, 54, 58, 61, 142, 297, 323, 334
RFC 7027, 354
RFC 7253, 107, 368
RFC 7292, 186
RFC 7539, 100, 240, 250, 291, 368, 369
RFC 7585, 94
RFC 7633, 71
RFC 7693, 245
RFC 7748, 334
RFC 7914, 216, 217
RFC 8032, 334
RFC 8410, 355
RFC 8452, 105
RFC 8994, 95
RFC 8998, 368
RFC 9106, 211, 212, 367
RFC 9171, 95
RFC 9172, 94
RFC 9174, 94, 95
RFC 9180, 206, 293, 367
RFC 9180 Section 8.1, 206
RFC 9598, 95
RFC 9881, 290
RFC 9925, 54, 90, 93, 290
rfc4514_attribute_name (*cryptography.x509.NameAttribute attribute*), 64
rfc4514_string() (*cryptography.x509.Name method*), 63
rfc4514_string() (*cryptography.x509.NameAttribute method*), 64
rfc4514_string() (*cryptography.x509.RelativeDistinguishedName method*), 64
RFC822Name (*class in cryptography.x509*), 65
rotate() (*cryptography.fernet.MultiFernet method*), 7
RSA (*cryptography.x509.certificate_transparency.SignatureAlgorithm attribute*), 20
rsa_crt_dmp1() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 160
rsa_crt_dmq1() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 160
rsa_crt_iqmp() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 160
rsa_recover_prime_factors() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 160
rsa_recover_private_exponent() (*in module cryptography.hazmat.primitives.asymmetric.rsa*), 160
RSA_WITH_MD5 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 90
RSA_WITH_SHA1 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA224 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA256 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA384 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA3_224 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA3_256 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA3_384 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA3_512 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSA_WITH_SHA512 (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91
RSAES_PKCS1_v1_5 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
RSAPrivateKey (*class in cryptography.hazmat.primitives.asymmetric.rsa*), 161
RSAPrivateNumbers (*class in cryptography.hazmat.primitives.asymmetric.rsa*), 158
RSAPublicKey (*class in cryptography.hazmat.primitives.asymmetric.rsa*), 162
RSAPublicNumbers (*class in cryptography.hazmat.primitives.asymmetric.rsa*), 158

RSASSA_PSS (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
RSASSA_PSS (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 91

S

Scrypt (*class in cryptography.hazmat.primitives.kdf.scrypt*), 216
SECP192R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 146
SECP192R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SECP224R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 146
SECP224R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SECP256K1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 147
SECP256K1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SECP256R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 146
SECP256R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SECP384R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 146
SECP384R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SECP521R1 (*class in cryptography.hazmat.primitives.asymmetric.ec*), 146
SECP521R1 (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurveOID attribute*), 151
SEED (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 283
sequence() (*in module cryptography.hazmat.asn1*), 275
serial (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 182
serial() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 185
serial_number (*cryptography.x509.Certificate attribute*), 43
serial_number (*cryptography.x509.ocsp.OCSPRequest attribute*), 25
serial_number (*cryptography.x509.ocsp.OCSPResponse attribute*), 30
serial_number (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 33
SERIAL_NUMBER (*cryptography.x509.oid.NameOID attribute*), 89
serial_number (*cryptography.x509.RevokedCertificate attribute*), 58
serial_number() (*cryptography.x509.CertificateBuilder method*), 52
serial_number() (*cryptography.x509.RevokedCertificateBuilder method*), 59
serialize_certificates() (*in module cryptography.hazmat.primitives.serialization.pkcs7*), 191
serialize_java_truststore() (*in module cryptography.hazmat.primitives.serialization.pkcs12*), 188
serialize_key_and_certificates() (*in module cryptography.hazmat.primitives.serialization.pkcs12*), 187
SERVER_AUTH (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
ServerVerifier (*class in cryptography.x509.verification*), 35
set() (*in module cryptography.hazmat.asn1*), 276
set_content_encryption_algorithm() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7EnvelopeBuilder method*), 192
set_data() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7EnvelopeBuilder method*), 192
set_data() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7SignatureBuilder method*), 191
SetOf (*class in cryptography.hazmat.asn1*), 279
SHA1 (*class in cryptography.hazmat.primitives.hashes*), 246
SHA224 (*class in cryptography.hazmat.primitives.hashes*), 245
SHA256 (*class in cryptography.hazmat.primitives.hashes*), 245
SHA384 (*class in cryptography.hazmat.primitives.hashes*), 245
SHA3_224 (*class in cryptography.hazmat.primitives.hashes*), 246
SHA3_256 (*class in cryptography.hazmat.primitives.hashes*), 246
SHA3_384 (*class in cryptography.hazmat.primitives.hashes*), 246
SHA3_512 (*class in cryptography.hazmat.primitives.hashes*), 246

SHA512 (*class in cryptography.hazmat.primitives.hashes*), 245
SHA512_224 (*class in cryptography.hazmat.primitives.hashes*), 245
SHA512_256 (*class in cryptography.hazmat.primitives.hashes*), 245
SHAKE128 (*class in cryptography.hazmat.primitives.hashes*), 247
SHAKE128 (*cryptography.hazmat.primitives.hpke.KDF attribute*), 209
SHAKE256 (*class in cryptography.hazmat.primitives.hashes*), 247
SHAKE256 (*cryptography.hazmat.primitives.hpke.KDF attribute*), 209
SIG_REQUIRED (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 31
sign() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateKey method*), 173
sign() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePrivateKey method*), 148
sign() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519PrivateKey method*), 131
sign() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448PrivateKey method*), 137
sign() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey method*), 116
sign() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey method*), 119
sign() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey method*), 122
sign() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey method*), 161
sign() (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7SignatureBuilder method*), 192
sign() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 186
sign() (*cryptography.x509.CertificateBuilder method*), 53
sign() (*cryptography.x509.CertificateRevocationListBuilder method*), 57
sign() (*cryptography.x509.CertificateSigningRequestBuilder method*), 61
sign() (*cryptography.x509.ocsp.OCSPResponseBuilder method*), 24
sign_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PrivateKey method*), 116
sign_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PrivateKey method*), 119
sign_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PrivateKey method*), 123
signature (*cryptography.x509.Certificate attribute*), 46
signature (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
signature (*cryptography.x509.CertificateRevocationList attribute*), 50
signature (*cryptography.x509.CertificateSigningRequest attribute*), 56
signature (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
signature_algorithm (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
signature_algorithm_oid (*cryptography.x509.Certificate attribute*), 45
signature_algorithm_oid (*cryptography.x509.CertificateRevocationList attribute*), 49
signature_algorithm_oid (*cryptography.x509.CertificateSigningRequest attribute*), 55
signature_algorithm_oid (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
signature_algorithm_parameters (*cryptography.x509.Certificate attribute*), 45
signature_algorithm_parameters (*cryptography.x509.CertificateRevocationList attribute*), 49
signature_algorithm_parameters (*cryptography.x509.CertificateSigningRequest attribute*), 55
signature_hash_algorithm (*cryptography.x509.Certificate attribute*), 45
signature_hash_algorithm (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
signature_hash_algorithm (*cryptography.x509.CertificateRevocationList attribute*), 48
signature_hash_algorithm (*cryptography.x509.CertificateSigningRequest attribute*), 54
signature_hash_algorithm (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
signature_key() (*cryptography.hazmat.primitives.serialization.SSHCertificate method*), 183
SignatureAlgorithm (*class in cryptography.x509.certificate_transparency*), 20
SignatureAlgorithmOID (*class in cryptogra-*

phy.x509.oid), 90
SIGNED_CERTIFICATE_TIMESTAMPS (*cryptography.x509.oid.ExtensionOID attribute*), 97
SignedCertificateTimestamp (*class in cryptography.x509.certificate_transparency*), 18
SignedCertificateTimestamps (*class in cryptography.x509*), 76
single_extensions (*cryptography.x509.ocsp.OCSPResponse attribute*), 30
Size (*class in cryptography.hazmat.asn1*), 281
skip_certs (*cryptography.x509.InhibitAnyPolicy attribute*), 80
SM3 (*class in cryptography.hazmat.primitives.hashes*), 247
SM4 (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 251
SMARTCARD_LOGON (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
SMIME (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 200
SMTP_UTF8_MAILBOX (*cryptography.x509.oid.OtherNameFormOID attribute*), 94
squeeze() (*cryptography.hazmat.primitives.hashes.XOFHash method*), 244
ssh_key_fingerprint() (*in module cryptography.hazmat.primitives.serialization*), 180
SSHCertificate (*class in cryptography.hazmat.primitives.serialization*), 182
SSHCertificateBuilder (*class in cryptography.hazmat.primitives.serialization*), 184
SSHCertificateType (*class in cryptography.hazmat.primitives.serialization*), 184
SSHCertPrivateKeyTypes (*in module cryptography.hazmat.primitives.serialization*), 182
SSHCertPublicKeyTypes (*in module cryptography.hazmat.primitives.serialization*), 182
SSHPrivateKeyTypes (*in module cryptography.hazmat.primitives.serialization*), 181
SSHPublicKeyTypes (*in module cryptography.hazmat.primitives.serialization*), 179
STATE_OR_PROVINCE_NAME (*cryptography.x509.oid.NameOID attribute*), 89
status_request (*cryptography.x509.TLSFeatureType attribute*), 71
status_request_v2 (*cryptography.x509.TLSFeatureType attribute*), 71
Store (*class in cryptography.x509.verification*), 34
store (*cryptography.x509.verification.ClientVerifier attribute*), 34
store (*cryptography.x509.verification.ServerVerifier attribute*), 35
store() (*cryptography.x509.verification.PolicyBuilder method*), 36
STREET_ADDRESS (*cryptography.x509.oid.NameOID attribute*), 89
Subject (*class in cryptography.x509.verification*), 34
subject (*cryptography.x509.Certificate attribute*), 45
subject (*cryptography.x509.CertificateSigningRequest attribute*), 54
subject (*cryptography.x509.verification.Policy attribute*), 39
SUBJECT_ALTERNATIVE_NAME (*cryptography.x509.oid.ExtensionOID attribute*), 95
SUBJECT_DIRECTORY_ATTRIBUTES (*cryptography.x509.oid.ExtensionOID attribute*), 97
SUBJECT_INFORMATION_ACCESS (*cryptography.x509.oid.ExtensionOID attribute*), 96
SUBJECT_KEY_IDENTIFIER (*cryptography.x509.oid.ExtensionOID attribute*), 96
subject_name() (*cryptography.x509.CertificateBuilder method*), 52
subject_name() (*cryptography.x509.CertificateSigningRequestBuilder method*), 60
SubjectAlternativeName (*class in cryptography.x509*), 74
SubjectInformationAccess (*class in cryptography.x509*), 77
SubjectInformationAccessOID (*class in cryptography.x509.oid*), 95
SubjectKeyIdentifier (*class in cryptography.x509*), 73
SubjectPublicKeyInfo (*cryptography.hazmat.primitives.serialization.PublicFormat attribute*), 198
subjects (*cryptography.x509.verification.VerifiedClient attribute*), 34
SUCCESSFUL (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 30
Suite (*class in cryptography.hazmat.primitives.hpke*), 206
superseded (*cryptography.x509.ReasonFlags attribute*), 79
SURNAME (*cryptography.x509.oid.NameOID attribute*), 89
symmetric cryptography, **400**

T

tag (*cryptography.hazmat.asn1.Variant attribute*), 279
tag (*cryptography.hazmat.primitives.ciphers.AEADEncryptionContext attribute*), 259
tag (*cryptography.hazmat.primitives.ciphers.modes.ModeWithAuthenticati attribute*), 261

tag_bytes (*cryptography.hazmat.asn1.TLV attribute*), 278
tbs_certificate_bytes (*cryptography.x509.Certificate attribute*), 46
tbs_certlist_bytes (*cryptography.x509.CertificateRevocationList attribute*), 50
tbs_certrequest_bytes (*cryptography.x509.CertificateSigningRequest attribute*), 56
tbs_precertificate_bytes (*cryptography.x509.Certificate attribute*), 47
tbs_response_bytes (*cryptography.x509.ocsp.OCSPResponse attribute*), 26
template_id (*cryptography.x509.MSCertificateTemplate attribute*), 82
Text (*cryptography.hazmat.primitives.serialization.pkcs7.PKCS7Options attribute*), 196
text (*cryptography.x509.NamingAuthority attribute*), 86
this_update (*cryptography.x509.ocsp.OCSPResponse attribute*), 28
this_update (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 32
this_update_utc (*cryptography.x509.ocsp.OCSPResponse attribute*), 28
this_update_utc (*cryptography.x509.ocsp.OCSPSingleResponse attribute*), 32
thread safety, **400**
time() (*cryptography.x509.verification.PolicyBuilder method*), 36
TIME_STAMPING (*cryptography.x509.oid.ExtendedKeyUsageOID attribute*), 93
timestamp (*cryptography.x509.certificate_transparency.SignedCertificateTimestamp attribute*), 19
TITLE (*cryptography.x509.oid.NameOID attribute*), 89
TLS_FEATURE (*cryptography.x509.oid.ExtensionOID attribute*), 96
TLSFeature (*class in cryptography.x509*), 71
TLSFeatureType (*class in cryptography.x509*), 71
TLV (*class in cryptography.hazmat.asn1*), 278
TOTP (*class in cryptography.hazmat.primitives.twofactor.totp*), 265
TraditionalOpenSSL (*cryptography.hazmat.primitives.serialization.PrivateFormat attribute*), 197
TripleDES (*class in cryptography.hazmat.decrepit.ciphers.algorithms*), 283
TripleDES (*class in cryptography.hazmat.primitives.ciphers.algorithms*), 251
TRY_LATER (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 31
tweak (*cryptography.hazmat.primitives.ciphers.modes.ModeWithTweak attribute*), 261
type (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
type (*cryptography.x509.UnsupportedGeneralNameType attribute*), 100
type() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilde method*), 185
type_id (*cryptography.x509.OtherName attribute*), 66

U

U-label, **400**
UNAUTHORIZED (*cryptography.x509.ocsp.OCSPResponseStatus attribute*), 31
UncompressedPoint (*cryptography.hazmat.primitives.serialization.PublicFormat attribute*), 199
UniformResourceIdentifier (*class in cryptography.x509*), 66
UNKNOWN (*cryptography.x509.ocsp.OCSPCertStatus attribute*), 31
unpadder() (*cryptography.hazmat.primitives.padding.ANSIX923 method*), 262
unpadder() (*cryptography.hazmat.primitives.padding.PKCS7 method*), 262
UnrecognizedExtension (*class in cryptography.x509*), 82
unsafe, **400**
UNSIGNED (*cryptography.x509.oid.NameOID attribute*), 90
UNSIGNED (*cryptography.x509.oid.SignatureAlgorithmOID attribute*), 93
unspecified (*cryptography.x509.ReasonFlags attribute*), 79
UNSTRUCTURED_NAME (*cryptography.x509.oid.AttributeOID attribute*), 98
UNSTRUCTURED_NAME (*cryptography.x509.oid.NameOID attribute*), 90
UnsupportedAlgorithm (*class in cryptography.exceptions*), 267
UnsupportedGeneralNameType (*class in cryptography.x509*), 99
update() (*cryptography.cobblestone.Cobblestone128Decryptor method*), 11

update() (*cryptography.cobblestone.Cobblestone128Encryptor method*), 10
update() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSAMultiHash method*), 125
update() (*cryptography.hazmat.primitives.ciphers.CipherContext method*), 257
update() (*cryptography.hazmat.primitives.cmac.CMAC method*), 237
update() (*cryptography.hazmat.primitives.hashes.Hash method*), 242
update() (*cryptography.hazmat.primitives.hashes.HashContext method*), 248
update() (*cryptography.hazmat.primitives.hashes.XOFHash method*), 244
update() (*cryptography.hazmat.primitives.hmac.HMAC method*), 239
update() (*cryptography.hazmat.primitives.padding.PaddingContext method*), 263
update() (*cryptography.hazmat.primitives.poly1305.Poly1305 method*), 241
update_into() (*cryptography.cobblestone.Cobblestone128Decryptor method*), 11
update_into() (*cryptography.cobblestone.Cobblestone128Encryptor method*), 10
update_into() (*cryptography.hazmat.primitives.ciphers.CipherContext method*), 257
url (*cryptography.x509.NamingAuthority attribute*), 86
USER (*cryptography.hazmat.primitives.serialization.SSHCertificateType attribute*), 184
USER_ID (*cryptography.x509.oid.NameOID attribute*), 90
UserNotice (*class in cryptography.x509*), 84
UTCTime (*class in cryptography.hazmat.asn1*), 277

V

v1 (*cryptography.x509.certificate_transparency.Version attribute*), 20
v1 (*cryptography.x509.Version attribute*), 63
v3 (*cryptography.x509.Version attribute*), 63
valid_after (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
valid_after() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 185
valid_before (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
valid_before() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 186
valid_for_all_principals() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 185
valid_principals (*cryptography.hazmat.primitives.serialization.SSHCertificate attribute*), 183
valid_principals() (*cryptography.hazmat.primitives.serialization.SSHCertificateBuilder method*), 185
validate_for_algorithm() (*cryptography.hazmat.primitives.ciphers.modes.Mode method*), 260
validation_time (*cryptography.x509.verification.Policy attribute*), 39
value (*cryptography.hazmat.asn1.Variant attribute*), 279
value (*cryptography.x509.Attribute attribute*), 88
value (*cryptography.x509.DirectoryName attribute*), 65
value (*cryptography.x509.DNSName attribute*), 65
value (*cryptography.x509.Extension attribute*), 68
value (*cryptography.x509.IPAddress attribute*), 66
value (*cryptography.x509.NameAttribute attribute*), 63
value (*cryptography.x509.OtherName attribute*), 66
value (*cryptography.x509.RegisteredID attribute*), 66
value (*cryptography.x509.RFC822Name attribute*), 65
value (*cryptography.x509.UniformResourceIdentifier attribute*), 66
value (*cryptography.x509.UnrecognizedExtension attribute*), 82
value_set() (*in module cryptography.hazmat.asn1*), 276
VariantType (*class in cryptography.hazmat.asn1*), 279
VerificationError (*class in cryptography.x509.verification*), 36
VerifiedClient (*class in cryptography.x509.verification*), 34
verify() (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicKey method*), 175
verify() (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePu method*), 149
verify() (*cryptography.hazmat.primitives.asymmetric.ed25519.Ed25519P method*), 133
verify() (*cryptography.hazmat.primitives.asymmetric.ed448.Ed448Public method*), 138
verify() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44Pu method*), 118
verify() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65Pu method*), 121
verify() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87Pu method*), 124
verify() (*cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey method*), 163
verify() (*cryptography.hazmat.primitives.cmac.CMAC method*), 237
verify() (*cryptography.hazmat.primitives.hmac.HMAC

method), 239
verify() (*cryptography.hazmat.primitives.kdf.argon2.Argon2id method*), 213
verify() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHash method*), 220
verify() (*cryptography.hazmat.primitives.kdf.concatkdf.ConcatKDFHMAC method*), 222
verify() (*cryptography.hazmat.primitives.kdf.hkdf.HKDF method*), 224
verify() (*cryptography.hazmat.primitives.kdf.hkdf.HKDFExpand method*), 226
verify() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFCMAC method*), 231
verify() (*cryptography.hazmat.primitives.kdf.kbkdf.KBKDFHMAC method*), 229
verify() (*cryptography.hazmat.primitives.kdf.KeyDerivationFunction method*), 234
verify() (*cryptography.hazmat.primitives.kdf.pbkdf2.PBKDF2HMAC method*), 216
verify() (*cryptography.hazmat.primitives.kdf.scrypt.Scrypt method*), 218
verify() (*cryptography.hazmat.primitives.kdf.x963kdf.X963KDF method*), 233
verify() (*cryptography.hazmat.primitives.poly1305.Poly1305 method*), 241
verify() (*cryptography.hazmat.primitives.twofactor.hotp.HOTP method*), 264
verify() (*cryptography.hazmat.primitives.twofactor.totp.TOTP method*), 266
verify() (*cryptography.x509.verification.ClientVerifier method*), 35
verify() (*cryptography.x509.verification.ServerVerifier method*), 35
verify_cert_signature() (*cryptography.hazmat.primitives.serialization.SSHCertificate method*), 184
verify_directly_issued_by() (*cryptography.x509.Certificate method*), 47
verify_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA44PublicKey method*), 118
verify_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA65PublicKey method*), 121
verify_mu() (*cryptography.hazmat.primitives.asymmetric.mldsa.MLDSA87PublicKey method*), 124
verify_phc_encoded() (*cryptography.hazmat.primitives.kdf.argon2.Argon2id class method*), 214
verify_tag() (*cryptography.hazmat.primitives.poly1305.Poly1305 class method*), 241
Version (*class in cryptography.x509*), 63
Version (*class in cryptography.x509.certificate_transparency*), 19
version (*cryptography.x509.Certificate attribute*), 42
version (*cryptography.x509.certificate_transparency.SignedCertificateTim attribute*), 18

W

webpki_defaults_ca() (*cryptography.x509.verification.ExtensionPolicy static method*), 38
webpki_defaults_ee() (*cryptography.x509.verification.ExtensionPolicy static method*), 38

X

x (*cryptography.hazmat.primitives.asymmetric.dh.DHPrivateNumbers attribute*), 169
x (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPrivateNumbers attribute*), 172
x (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumb attribute*), 144
X25519 (*cryptography.hazmat.primitives.hpke.KEM attribute*), 208
X25519 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
X25519PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.x25519*), 134
X25519PublicKey (*class in cryptography.hazmat.primitives.asymmetric.x25519*), 135
X448 (*cryptography.x509.oid.PublicKeyAlgorithmOID attribute*), 98
X448PrivateKey (*class in cryptography.hazmat.primitives.asymmetric.x448*), 139
X448PublicKey (*class in cryptography.hazmat.primitives.asymmetric.x448*), 140
X500_UNIQUE_IDENTIFIER (*cryptography.x509.oid.NameOID attribute*), 89
X509_CERTIFICATE (*cryptography.x509.certificate_transparency.LogEntryType attribute*), 20
X962 (*cryptography.hazmat.primitives.serialization.Encoding attribute*), 200
X963KDF (*class in cryptography.hazmat.primitives.kdf.x963kdf*), 232
xof() (*cryptography.hazmat.primitives.hashes.SHAKE128 class method*), 247
xof() (*cryptography.hazmat.primitives.hashes.SHAKE256 class method*), 247
XOFHash (*class in cryptography.hazmat.primitives.hashes*), 243

XTS (*class in cryptography.hazmat.primitives.ciphers.modes*), 256

Y

y (*cryptography.hazmat.primitives.asymmetric.dh.DHPublicNumbers attribute*), 169
y (*cryptography.hazmat.primitives.asymmetric.dsa.DSAPublicNumbers attribute*), 172
y (*cryptography.hazmat.primitives.asymmetric.ec.EllipticCurvePublicNumbers attribute*), 144