Cryptography Documentation

Release 2.7.dev1

Individual Contributors

May 26, 2019

The recipes layer

`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
'...'
>>> f.decrypt(token)
'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 1

Installation

You can install `cryptography` with `pip`:

```
$ pip install cryptography
```

See *Installation* for more information.

CHAPTER 2

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.

```
>>> 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*) – 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 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.

---

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*) – 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`.

extract_timestamp(*token*)
New 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*) – 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`.

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

New 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*) – 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`.

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 such as *PBKDF2HMAC*, bcrypt or *Scrypt*.

```
>>> import base64
>>> import os
>>> from cryptography.fernet import Fernet
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
>>> password = b"password"
>>> salt = os.urandom(16)
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=100000,
...     backend=default_backend()
... )
>>> 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 iteration count used should be adjusted to be as high as your server can tolerate. A good default is at least 100,000 iterations which is what Django recommended in 2014.

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. Unfortunately, this makes it generally unsuitable for very large files at this time.

2.2 X.509

X.509 is an ITU-T standard for a public key infrastructure. X.509v3 is defined in **RFC 5280** (which obsoletes **RFC 2459** and **RFC 3280**). X.509 certificates are commonly used in protocols like TLS.

2.2.1 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.backends import default_backend
>>> 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,
...     backend=default_backend()
... )
>>> # 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, u"US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, u"mysite.com"),
... ])).add_extension(
...     x509.SubjectAlternativeName([
...         # Describe what sites we want this certificate for.
...         x509.DNSName(u"mysite.com"),
...         x509.DNSName(u"www.mysite.com"),
...         x509.DNSName(u"subdomain.mysite.com"),
...     ]),
...     critical=False,
... # Sign the CSR with our private key.
... ).sign(key, hashes.SHA256(), default_backend())
>>> # 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,
...     backend=default_backend()
... )
>>> # 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:

```
>>> # Various details about who we are. For a self-signed certificate the
>>> # subject and issuer are always the same.
>>> subject = issuer = x509.Name([
```

(continues on next page)

(continued from previous page)

```
...     x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
...     x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"California"),
...     x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
...     x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
...     x509.NameAttribute(NameOID.COMMON_NAME, u"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.utcnow()
... ).not_valid_after(
...     # Our certificate will be valid for 10 days
...     datetime.datetime.utcnow() + datetime.timedelta(days=10)
... ).add_extension(
...     x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
...     critical=False,
... # Sign our certificate with our private key
... ).sign(key, hashes.SHA256(), default_backend())
>>> # Write our certificate out to disk.
>>> 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.

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
```

2.2.2 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**

New 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.

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

New in version 2.0.

An enumeration for SignedCertificateTimestamp versions.

v1

For version 1 SignedCertificateTimestamps.

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

New 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.

2.2.3 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*)

New 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**

New 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. This can only be called 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_extension(*extension*, *critical*)

Adds an extension to the request.

Parameters

- **extension** – 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.backends import default_backend
>>> from cryptography.hazmat.primitives import serialization
>>> from cryptography.hazmat.primitives.hashes import SHA1
>>> from cryptography.x509 import load_pem_x509_certificate, ocsp
>>> cert = load_pem_x509_certificate(pem_cert, default_backend())
>>> issuer = load_pem_x509_certificate(pem_issuer, default_backend())
>>> builder = ocsp.OCSPRequestBuilder()
>>> # SHA1 is in this example because RFC 5019 mandates its use.
>>> builder = builder.add_certificate(cert, issuer, SHA1())
>>> req = builder.build()
>>> base64.b64encode(req.public_bytes(serialization.Encoding.DER))
b'MEMwQTA/MD0wOzAJBgUrDgMCGgUABBRAC0Z68eay0wmDug1gfn5ZN0gkxAQUw5zz/NNGCDS7zkZ/
↪oHxb8+IIy1kCAj8g'
```

Loading Responses

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

New 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**

New 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.

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 naïve `datetime.datetime` object representing the most recent time in UTC at which the status being indicated is known by the responder to be correct.
- **next_update** – A naïve `datetime.datetime` object or `None`. The time in UTC at or before which newer information will be available about the status of the certificate.
- **revocation_time** – A naïve `datetime.datetime` object or `None` if the `cert` is not revoked. The time in UTC at which the certificate was revoked.
- **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(*extension*, *critical*)

Adds an extension to the response.

Parameters

- **extension** – 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* or *EllipticCurvePrivateKey* that will be used to sign the certificate.
- **algorithm** – The *HashAlgorithm* that will be used to generate the signature.

Returns A new *OCSPResponse*.

```
>>> import datetime
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes, serialization
>>> from cryptography.x509 import load_pem_x509_certificate, ocsp
>>> cert = load_pem_x509_certificate(pem_cert, default_backend())
>>> issuer = load_pem_x509_certificate(pem_issuer, default_backend())
>>> responder_cert = load_pem_x509_certificate(pem_responder_cert, default_
↪backend())
>>> responder_key = serialization.load_pem_private_key(pem_responder_key, None,␣
↪default_backend())
>>> builder = ocsp.OCSPResponseBuilder()
>>> # SHA1 is in this example because RFC 5019 mandates its use.
>>> builder = builder.add_response(
...     cert=cert, issuer=issuer, algorithm=hashes.SHA1(),
...     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.backends import default_backend
>>> 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**
New 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**
New 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
New in version 2.5.

Type *HashAlgorithm*

Returns the *HashAlgorithm* which was used in signing this response.

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`

A naïve 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*.

revocation_time

Type `datetime.datetime` or None

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*.

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*.

this_update

Type `datetime.datetime`

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*.

next_update

Type `datetime.datetime`

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

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

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*.

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*.

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*.

serial_number

Type int

The serial number of the certificate that was checked.

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

extensions

Type *Extensions*

The extensions encoded in the response.

public_bytes(*encoding*)

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

Return bytes The serialized OCSP response.

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

2.2.4 X.509 Reference

Loading Certificates

cryptography.x509.**load_pem_x509_certificate**(*data*, *backend*)
New 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.

- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *Certificate*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> cert = x509.load_pem_x509_certificate(pem_data, default_backend())
>>> cert.serial_number
2
```

cryptography.x509.**load_der_x509_certificate**(*data*, *backend*)

New 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.
- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *Certificate*.

Loading Certificate Revocation Lists

cryptography.x509.**load_pem_x509_crl**(*data*, *backend*)

New 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.
- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *CertificateRevocationList*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> crl = x509.load_pem_x509_crl(pem_crl_data, default_backend())
>>> isinstance(crl.signature_hash_algorithm, hashes.SHA256)
True
```

cryptography.x509.**load_der_x509_crl**(*data*, *backend*)

New 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.
- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *CertificateRevocationList*.

Loading Certificate Signing Requests

cryptography.x509.**load_pem_x509_csr**(*data*, *backend*)

New 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.
- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *CertificateSigningRequest*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> csr = x509.load_pem_x509_csr(pem_req_data, default_backend())
>>> isinstance(csr.signature_hash_algorithm, hashes.SHA1)
True
```

cryptography.x509.**load_der_x509_csr**(*data*, *backend*)

New 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.
- **backend** – A backend supporting the *X509Backend* interface.

Returns An instance of *CertificateSigningRequest*.

X.509 Certificate Object

class cryptography.x509.**Certificate**

New 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 *RSAPublicKey* or *DSAPublicKey* or *EllipticCurvePublicKey*

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

not_valid_before

Type datetime.datetime

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_after

Type datetime.datetime

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)
```

issuer

New in version 0.8.

Type *Name*

The *Name* of the issuer.

subject

New 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.

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

signature_algorithm_oid

New 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)>
```

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.
- **UnicodeError** – If an extension contains IDNA encoding that is invalid or not compliant with IDNA 2008.

```
>>> 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=None,␣
↪decipher_only=None)>)>
<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

New in version 1.2.

Type bytes

The bytes of the certificate's signature.

tbs_certificate_bytes
New 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
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> issuer_public_key = load_pem_public_key(pem_issuer_public_key, default_
↪backend())
>>> cert_to_check = x509.load_pem_x509_certificate(pem_data_to_check, default_
↪backend())
>>> 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
:class:`~cryptography.exceptions.InvalidSignature`
exception will be raised if the signature fails to verify.
```

public_bytes(*encoding*)
New 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**
New 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 'cryptography.hazmat.backends.openssl.x509._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'e\xcf.\xc4:\x83?1\xdc\xf3\xfc\x95\xd7\xb3\x87\xb3\x8e\xf8\xb93!
↪\x87\x07\x9d\x1b\xb4!\xb9\xe4W\xf4\x1f'
```

get_revoked_certificate_by_serial_number(*serial_number*)

New 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.

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

signature_algorithm_oid

New 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)>
```

issuer

Type *Name*

The *Name* of the issuer.

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

next_update

Type `datetime.datetime`

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

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

last_update

Type `datetime.datetime`

A naïve datetime representing when this CRL was last updated.

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

extensions

Type *Extensions*

The extensions encoded in the CRL.

signature

New in version 1.2.

Type bytes

The bytes of the CRL's signature.

tbs_certlist_bytes

New 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*)

New 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*)

New 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**

New in version 1.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> 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,
...     backend=default_backend()
```

(continues on next page)

(continued from previous page)

```
... )
>>> public_key = private_key.public_key()
>>> builder = x509.CertificateBuilder()
>>> builder = builder.subject_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, u'cryptography.io'),
... ]))
>>> builder = builder.issuer_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, u'cryptography.io'),
... ]))
>>> 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(public_key)
>>> builder = builder.add_extension(
...     x509.SubjectAlternativeName(
...         [x509.DNSName(u'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(),
...     backend=default_backend()
... )
>>> isinstance(certificate, x509.Certificate)
True
```

issuer_name(*name*)

Sets the issuer's distinguished name.

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

subject_name(*name*)

Sets the subject's distinguished name.

Parameters **name** – The *Name* that describes the subject.

public_key(*public_key*)

Sets the subject's public key.

Parameters **public_key** – The subject's public key. This can be one of *RSAPublicKey*, *DSAPublicKey* or *EllipticCurvePublicKey*

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.

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.

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.

add_extension(*extension*, *critical*)
Adds an X.509 extension to the certificate.

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.

sign(*private_key*, *algorithm*, *backend*)
Sign the certificate using the CA's private key.

Parameters

- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* that will be used to sign the certificate.
- **algorithm** – The *HashAlgorithm* that will be used to generate the signature.
- **backend** – Backend that will be used to build the certificate. Must support the *X509Backend* interface.

Returns *Certificate*

X.509 CSR (Certificate Signing Request) Object

class cryptography.x509.**CertificateSigningRequest**
New in version 0.9.

public_key()
The public key associated with the request.

Returns *RSAPublicKey* or *DSAPublicKey* or *EllipticCurvePublicKey*

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

subject

Type *Name*

The *Name* of the subject.

signature_hash_algorithm

Type *HashAlgorithm*

Returns the *HashAlgorithm* which was used in signing this request.

```
>>> from cryptography.hazmat.primitives import hashes
>>> isinstance(csr.signature_hash_algorithm, hashes.SHA1)
True
```

signature_algorithm_oid
New 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=1.2.840.113549.1.1.5, name=sha1WithRSAEncryption)>
```

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.
- **UnicodeError** – If an extension contains IDNA encoding that is invalid or not compliant with IDNA 2008.

public_bytes(*encoding*)
New 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
New in version 1.2.

Type bytes

The bytes of the certificate signing request's signature.

tbs_certrequest_bytes
New 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
New in version 1.3.

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

X.509 Certificate Revocation List Builder

class cryptography.x509.**CertificateRevocationListBuilder**

New in version 1.2.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> 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,
...     backend=default_backend()
... )
>>> builder = x509.CertificateRevocationListBuilder()
>>> builder = builder.issuer_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, u'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(
...     datetime.datetime.today()
... ).build(default_backend())
>>> builder = builder.add_revoked_certificate(revoked_cert)
>>> crl = builder.sign(
...     private_key=private_key, algorithm=hashes.SHA256(),
...     backend=default_backend()
... )
>>> 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(*extension*, *critical*)

Adds an X.509 extension to this CRL.

Parameters

- **extension** – 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*, *backend*)
Sign this CRL using the CA's private key.

Parameters

- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* that will be used to sign the certificate.
- **algorithm** – The *HashAlgorithm* that will be used to generate the signature.
- **backend** – Backend that will be used to build the CRL. Must support the *X509Backend* interface.

Returns *CertificateRevocationList*

X.509 Revoked Certificate Object

class cryptography.x509.**RevokedCertificate**
New 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`

A naïve datetime representing the date this certificates was revoked.

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

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*.

New in version 1.2.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> import datetime
>>> builder = x509.RevokedCertificateBuilder()
>>> builder = builder.revocation_date(datetime.datetime.today())
>>> builder = builder.serial_number(3333)
>>> revoked_certificate = builder.build(default_backend())
>>> 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(*extension*, *critical*)

Adds an X.509 extension to this revoked certificate.

Parameters

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

build(*backend*)

Create a revoked certificate object using the provided backend.

Parameters **backend** – Backend that will be used to build the revoked certificate. Must support the *X509Backend* interface.

Returns *RevokedCertificate*

X.509 CSR (Certificate Signing Request) Builder Object

class cryptography.x509.**CertificateSigningRequestBuilder**

New in version 1.0.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.x509.oid import NameOID
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
...     backend=default_backend()
```

(continues on next page)

(continued from previous page)

```
... )
>>> builder = x509.CertificateSigningRequestBuilder()
>>> builder = builder.subject_name(x509.Name([
...     x509.NameAttribute(NameOID.COMMON_NAME, u'cryptography.io'),
... ]))
>>> builder = builder.add_extension(
...     x509.BasicConstraints(ca=False, path_length=None), critical=True,
... )
>>> request = builder.sign(
...     private_key, hashes.SHA256(), default_backend()
... )
>>> 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*.

sign(*private_key*, *algorithm*, *backend*)

Parameters

- **backend** – Backend that will be used to sign the request. Must support the *X509Backend* interface.
- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* 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.
- **algorithm** – The *HashAlgorithm* that will be used to generate the request signature.

Returns A new *CertificateSigningRequest*.

class cryptography.x509.**Name**

New 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')>
<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
New in version 1.6.

Type list of *RelativeDistinguishedName*

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(*backend*)
New in version 1.6.

Parameters **backend** – A backend supporting the *X509Backend* interface.

Return bytes The DER encoded name.

rfc4514_string()
New in version 2.5.

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

class cryptography.x509.**Version**
New 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**
New 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 *text*

The value of the attribute.

rfc4514_string()
New in version 2.5.

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

class cryptography.x509.**RelativeDistinguishedName**(*attributes*)
New 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()
New in version 2.5.

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

class cryptography.x509.**ObjectIdentifier**
New 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**
New in version 0.9.

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

class cryptography.x509.**RFC822Name**(*value*)
New in version 0.9.

Changed in version 2.1.

> **Warning:** Starting with version 2.1 *U-label* input is deprecated. If passing an internationalized domain name (IDN) you should first IDNA encode the value and then pass the result as a string. Accessing `value` will return the *A-label* encoded form even if you pass a U-label. This breaks backwards compatibility, but only for internationalized domain names.

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.

value

Type *text*

class cryptography.x509.**DNSName**(*value*)
New in version 0.9.

Changed in version 2.1.

> **Warning:** Starting with version 2.1 *U-label* input is deprecated. If passing an internationalized domain name (IDN) you should first IDNA encode the value and then pass the result as a string. Accessing `value` will return the *A-label* encoded form even if you pass a U-label. This breaks backwards compatibility, but only for internationalized domain names.

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.

type *text*

value

Type *text*

class cryptography.x509.**DirectoryName**(*value*)
New in version 0.9.

This corresponds to a directory name.

value

Type *Name*

class cryptography.x509.**UniformResourceIdentifier**(*value*)
New in version 0.9.

Changed in version 2.1.

> **Warning:** Starting with version 2.1 *U-label* input is deprecated. If passing an internationalized domain name (IDN) you should first IDNA encode the value and then pass the result as a string. Accessing `value` will return the *A-label* encoded form even if you pass a U-label. This breaks backwards compatibility, but only for internationalized domain names.

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.

value

Type *text*

class cryptography.x509.**IPAddress**(*value*)
New in version 0.9.

This corresponds to an IP address.

value

Type IPv4Address, IPv6Address, IPv4Network, or IPv6Network.

class cryptography.x509.**RegisteredID**(*value*)
New in version 0.9.

This corresponds to a registered ID.

value

Type *ObjectIdentifier*

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

Parameters **oid** – An *ObjectIdentifier* instance.

Returns An instance of the extension class.

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*)
New in version 1.1.

Parameters **extclass** – An extension class.

Returns An instance of the extension class.

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**
New 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**
New in version 1.0.

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

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

New 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

New 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*)

New 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

New in version 1.0.

Type *ObjectIdentifier*

Returns *EXTENDED_KEY_USAGE*.

class cryptography.x509.**OCSPNoCheck**

New 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

New in version 1.0.

Type *ObjectIdentifier*

Returns *OCSP_NO_CHECK*.

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

New 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**

New 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*)

New 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

New 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*)

New 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

New 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 *Name* 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*)

New 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 *RSAPublicKey*, *DSAPublicKey*, or *EllipticCurvePublicKey*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> issuer_cert = x509.load_pem_x509_certificate(pem_data, default_backend())
>>> 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*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> issuer_cert = x509.load_pem_x509_certificate(pem_data, default_backend())
>>> ski = issuer_cert.extensions.get_extension_for_class(x509.
↪SubjectKeyIdentifier)
>>> x509.AuthorityKeyIdentifier.from_issuer_subject_key_identifier(ski)
<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*)

New in version 0.9.

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

oid

New in version 1.0.

Type *ObjectIdentifier*

Returns *SUBJECT_KEY_IDENTIFIER*.

digest

Type bytes

The binary value of the identifier.

classmethod from_public_key(*public_key*)

New 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 *RSAPublicKey*, *DSAPublicKey*, or *EllipticCurvePublicKey*.

```
>>> from cryptography import x509
>>> from cryptography.hazmat.backends import default_backend
>>> csr = x509.load_pem_x509_csr(pem_req_data, default_backend())
>>> x509.SubjectKeyIdentifier.from_public_key(csr.public_key())
<SubjectKeyIdentifier(digest=b
↪'\xdb\xaa\xf0\x06\x11\xdbD\xfe\xbf\x93\x03\x8av\x88WP7\xa6\x91\xf7')>
```

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

New 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

New 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.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> cert = x509.load_pem_x509_certificate(cryptography_cert_pem, default_
↪backend())
>>> # 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*)

New 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

New 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*)

New 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**

New 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.**DeltaCRLIndicator**(*crl_number*)

New 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*)

New 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

New in version 1.0.

Type *ObjectIdentifier*

Returns *AUTHORITY_INFORMATION_ACCESS*.

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

New in version 0.9.

access_method

Type *ObjectIdentifier*

The access method defines what the `access_location` means. It must be either *OCSP* or *CA_ISSUERS*. 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.

access_location

Type *GeneralName*

Where to access the information defined by the access method.

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

New 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*)

New 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

New in version 1.0.

Type *ObjectIdentifier*

Returns *CRL_DISTRIBUTION_POINTS*.

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

New 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**

New 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*)

New 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

New in version 1.0.

Type *ObjectIdentifier*

Returns *INHIBIT_ANY_POLICY*.

skip_certs

Type int

class cryptography.x509.**PolicyConstraints**

New 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*)

New 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*)

New 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**

New in version 1.2.

A generic extension class used to hold the raw value of extensions that `cryptography` does not know how to parse.

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.**CertificatePolicies**(*policies*)

New 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.

oid

New in version 1.0.

Type *ObjectIdentifier*

Returns *CERTIFICATE_POLICIES*.

Certificate Policies Classes

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

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

New 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 *text* and/or *UserNotice* objects. If the value is text 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*)

New 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 *text*

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.

New in version 0.9.

organization

Type *text*

notice_numbers

Type list

A list of integers.

CRL Entry Extensions

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

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

New 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*)

New 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*)

New 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

OCSP Extensions

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

New 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

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.

New 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
New in version 1.6.

Corresponds to the dotted string `"2.5.4.9"`.

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"`.

GENERATION_QUALIFIER
Corresponds to the dotted string `"2.5.4.44"`.

X500_UNIQUE_IDENTIFIER
New 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
New 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
New in version 1.6.

Corresponds to the dotted string `"2.5.4.16"`.

POSTAL_CODE
New in version 1.6.

Corresponds to the dotted string `"2.5.4.17"`.

class cryptography.x509.oid.**SignatureAlgorithmOID**
New 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.

RSASSA_PSS
New 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.

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.

class cryptography.x509.oid.**ExtendedKeyUsageOID**
New 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
New 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.

class cryptography.x509.oid.**AuthorityInformationAccessOID**
New 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.**CertificatePoliciesOID**
New 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**
New 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.

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
New 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
New in version 1.9.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.2"`.

PRECERT_POISON
New in version 2.4.

Corresponds to the dotted string `"1.3.6.1.4.1.11129.2.4.3"`.

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
New in version 2.4.

Corresponds to the dotted string `"2.5.29.28"`.

class cryptography.x509.oid.**CRLEntryExtensionOID**
New 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**
New in version 2.4.

NONCE
Corresponds to the dotted string `"1.3.6.1.5.5.7.48.1.2"`.

Helper Functions

cryptography.x509.**random_serial_number**()
New 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.**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.

2.3 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.

2.3.1 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*)

New 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*) – The data to encrypt.
- **associated_data** (*bytes*) – 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^{32} 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*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes*) – 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.

class cryptography.hazmat.primitives.ciphers.aead.**AESGCM**(*key*)

New in version 2.0.

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*) – The data to encrypt.
- **associated_data** (*bytes*) – 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} 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*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes*) – 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.

class cryptography.hazmat.primitives.ciphers.aead.**AESCCM**(*key*, *tag_length=16*)

New 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*) – The data to encrypt.
- **associated_data** (*bytes*) – 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^{32} 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 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*) – The data to decrypt (with tag appended).
- **associated_data** (*bytes*) – 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.

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.

2.3.2 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.

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**

New 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*

```
>>> 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*) – 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* 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.

class `cryptography.hazmat.primitives.asymmetric.ed25519.`**Ed25519PublicKey**

New in version 2.6.

classmethod from_public_bytes(*data*)

Parameters **data** (*bytes*) – 32 byte public key.

Returns *Ed25519PublicKey*

```
>>> 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.

verify(*signature*, *data*)

Parameters

- **signature** (*bytes*) – The signature to verify.
- **data** (*bytes*) – The data to verify.

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.backends import default_backend
>>> 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.
>>> 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',
...     backend=default_backend()
... ).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',
...     backend=default_backend()
... ).derive(shared_key_2)
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.x25519.**X25519PrivateKey**

New in version 2.0.

classmethod generate()

Generate an X25519 private key.

Returns *X25519PrivateKey*

classmethod from_private_bytes(*data*)

New 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()
... )
>>> 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*)

New 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.

class cryptography.hazmat.primitives.asymmetric.x25519.**X25519PublicKey**

New 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(
...     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.

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**

New 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*) – 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.

class cryptography.hazmat.primitives.asymmetric.ed448.**Ed448PublicKey**

New 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.

verify(*signature*, *data*)

Parameters

- **signature** (*bytes*) – The signature to verify.
- **data** (*bytes*) – The data to verify.

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.backends import default_backend
>>> 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()
>>> # In a real handshake the peer_public_key will be received from the
```

(continues on next page)

(continued from previous page)

```
>>> # 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',
...     backend=default_backend()
... ).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',
...     backend=default_backend()
... ).derive(shared_key_2)
```

Key interfaces

class cryptography.hazmat.primitives.asymmetric.x448.**X448PrivateKey**

New 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.

class cryptography.hazmat.primitives.asymmetric.x448.**X448PublicKey**

New 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.

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*, *backend*)

New in version 0.5.

Generate a new private key on `curve` for use with `backend`.

Parameters

- **curve** – An instance of *EllipticCurve*.
- **backend** – An instance of *EllipticCurveBackend*.

Returns A new instance of *EllipticCurvePrivateKey*.

cryptography.hazmat.primitives.asymmetric.ec.**derive_private_key**(*private_value*, *curve*, *backend*)

New in version 1.6.

Derive a private key from `private_value` on `curve` for use with `backend`.

Parameters

- **private_value** (*int*) – The secret scalar value.
- **curve** – An instance of *EllipticCurve*.
- **backend** – An instance of *EllipticCurveBackend*.

Returns A new instance of *EllipticCurvePrivateKey*.

Elliptic Curve Signature Algorithms

class cryptography.hazmat.primitives.asymmetric.ec.**ECDSA**(*algorithm*)

New in version 0.5.

The ECDSA signature algorithm first standardized in NIST publication FIPS 186-3, and later in FIPS 186-4.

Parameters **algorithm** – An instance of *HashAlgorithm*.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> private_key = ec.generate_private_key(
...     ec.SECP384R1(), default_backend()
... )
>>> 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 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, default_backend())
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
```

(continues on next page)

(continued from previous page)

```
...     ec.ECDSA(utils.Prehashed(chosen_hash))
... )
```

Verification requires the public key, the 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()))
```

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, default_backend())
>>> 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*)

New 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(*backend*)

Convert a collection of numbers into a private key suitable for doing actual cryptographic operations.

Parameters **backend** – An instance of *EllipticCurveBackend*.

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.

New 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(*backend*)
Convert a collection of numbers into a public key suitable for doing actual cryptographic operations.

Parameters **backend** – An instance of *EllipticCurveBackend*.

Raises **ValueError** – Raised if the point is invalid for the curve.

Returns A new instance of *EllipticCurvePublicKey*.

encode_point()

Warning: This method is deprecated as of version 2.5. Callers should migrate to using *public_bytes()*.

New in version 1.1.

Encodes an elliptic curve point to a byte string as described in SEC 1 v2.0 section 2.3.3. This method only supports uncompressed points.

Return bytes The encoded point.

classmethod from_encoded_point(*curve*, *data*)
New in version 1.1.

Note: This has been deprecated in favor of *from_encoded_point()*

Decodes a byte string as described in SEC 1 v2.0 section 2.3.3 and returns an *EllipticCurvePublicNumbers*. This method only supports uncompressed points.

Parameters

- **curve** – An *EllipticCurve* instance.
- **data** (*bytes*) – The serialized point byte string.

Returns An *EllipticCurvePublicNumbers* instance.

Raises

- **ValueError** – Raised on invalid point type or data length.
- **TypeError** – Raised when curve is not an *EllipticCurve*.

Elliptic Curve Key Exchange algorithm

class cryptography.hazmat.primitives.asymmetric.ec.**ECDH**

New in version 1.1.

The Elliptic Curve Diffie-Hellman Key Exchange algorithm first standardized in NIST publication 800-56A, and later in 800-56Ar2.

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.backends import default_backend
>>> 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(), default_backend()
... )
>>> # In a real handshake the peer is a remote client. For this
>>> # example we'll generate another local private key though.
>>> peer_private_key = ec.generate_private_key(
...     ec.SECP384R1(), default_backend()
... )
>>> 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',
...     backend=default_backend()
... ).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',
```

(continues on next page)

(continued from previous page)

```
...     backend=default_backend()
... ).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.backends import default_backend
>>> 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(), default_backend()
... )
>>> # 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(), default_backend()
... ).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',
...     backend=default_backend()
... ).derive(shared_key)
>>> # For the next handshake we MUST generate another private key.
>>> private_key_2 = ec.generate_private_key(
...     ec.SECP384R1(), default_backend()
... )
>>> peer_public_key_2 = ec.generate_private_key(
...     ec.SECP384R1(), default_backend()
... ).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',
...     backend=default_backend()
... ).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**
New in version 0.5.

SECG curve `secp256r1`. Also called NIST P-256.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP384R1**
New in version 0.5.

SECG curve `secp384r1`. Also called NIST P-384.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP521R1**
New in version 0.5.

SECG curve `secp521r1`. Also called NIST P-521.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP224R1**
New in version 0.5.

SECG curve `secp224r1`. Also called NIST P-224.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP192R1**
New in version 0.5.

SECG curve `secp192r1`. Also called NIST P-192.

class cryptography.hazmat.primitives.asymmetric.ec.**SECP256K1**
New in version 0.9.

SECG curve `secp256k1`.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP256R1**
New in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP384R1**
New in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**BrainpoolP512R1**
New in version 2.2.

Brainpool curve specified in **RFC 5639**. These curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT571K1**
New in version 0.5.

SECG curve `sect571k1`. Also called NIST K-571. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT409K1**
New in version 0.5.

SECG curve `sect409k1`. Also called NIST K-409. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT283K1**
New in version 0.5.

SECG curve `sect283k1`. Also called NIST K-283. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT233K1**
New in version 0.5.

SECG curve `sect233k1`. Also called NIST K-233. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT163K1**
New in version 0.5.

SECG curve `sect163k1`. Also called NIST K-163. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT571R1**
New in version 0.5.

SECG curve `sect571r1`. Also called NIST B-571. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT409R1**
New in version 0.5.

SECG curve `sect409r1`. Also called NIST B-409. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT283R1**
New in version 0.5.

SECG curve `sect283r1`. Also called NIST B-283. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT233R1**
New in version 0.5.

SECG curve `sect233r1`. Also called NIST B-233. These binary curves are discouraged for new systems.

class cryptography.hazmat.primitives.asymmetric.ec.**SECT163R2**
New in version 0.5.

SECG curve `sect163r2`. Also called NIST B-163. These binary curves are discouraged for new systems.

Key Interfaces

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurve**
New 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()`*).

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurveSignatureAlgorithm**
New 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**
New in version 0.5.

An elliptic curve private key for use with an algorithm such as ECDSA or EdDSA. An elliptic curve private key that is not an *opaque key* also implements *EllipticCurvePrivateKeyWithSerialization* to provide serialization methods.

exchange(*algorithm*, *peer_public_key*)
New 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*)
New in version 1.5.

Sign one block of data which can be verified later by others using the public key.

Parameters

- **data** (*bytes*) – The message string to sign.
- **signature_algorithm** – An instance of *EllipticCurveSignatureAlgorithm*, such as *ECDSA*.

Return bytes Signature.

key_size
New in version 1.9.

Type int

Size (in *bits*) of a secret scalar for the curve (as generated by *generate_private_key()*).

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePrivateKeyWithSerializatio**
New in version 0.8.

This interface contains additional methods relating to serialization. Any object with this interface also has all the methods from *EllipticCurvePrivateKey*.

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* 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**
New 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*)
New in version 1.5.

Verify one block of data was signed by the private key associated with this public key.

Parameters

- **signature** (*bytes*) – The signature to verify.
- **data** (*bytes*) – The message string that was signed.
- **signature_algorithm** – An instance of *EllipticCurveSignatureAlgorithm*.

Raises *cryptography.exceptions.InvalidSignature* – If the signature does not validate.

key_size
New 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*)
New 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*.

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurvePublicKeyWithSerialization**
New in version 0.6.

Alias for *EllipticCurvePublicKey*.

Serialization

This sample demonstrates how to generate a private key and serialize it.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives import serialization

>>> private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())

>>> 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,
```

(continues on next page)

(continued from previous page)

```
...     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,
...     backend=default_backend()
... )

>>> loaded_private_key = serialization.load_pem_private_key(
...     serialized_private,
...     # or password=None, if in plain text
...     password=b'testpassword',
...     backend=default_backend()
... )
```

Elliptic Curve Object Identifiers

class cryptography.hazmat.primitives.asymmetric.ec.**EllipticCurveOID**

New 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

New in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.7"`.

BRAINPOOLP384R1

New in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.11"`.

BRAINPOOLP512R1
New in version 2.5.

Corresponds to the dotted string `"1.3.36.3.3.2.8.1.1.13"`.

SECT163K1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.1"`.

SECT163R2
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.15"`.

SECT233K1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.26"`.

SECT233R1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.27"`.

SECT283K1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.16"`.

SECT283R1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.17"`.

SECT409K1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.36"`.

SECT409R1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.37"`.

SECT571K1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.38"`.

SECT571R1
New in version 2.5.

Corresponds to the dotted string `"1.3.132.0.39"`.

cryptography.hazmat.primitives.asymmetric.ec.**get_curve_for_oid**(*oid*)
New 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*, *backend*)

New in version 0.5.

Generates a new RSA private key using the provided `backend`. `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.backends import default_backend
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
...     backend=default_backend()
... )
```

Parameters

- **`public_exponent`** (*int*) – The public exponent of the new key. Usually one of the small Fermat primes 3, 5, 17, 257, 65537. If in doubt you 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. Some backends may have additional limitations.
- **`backend`** – A backend which implements *RSABackend*.

Returns An instance of *RSAPrivateKey*.

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

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.backends import default_backend
>>> 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,
...         backend=default_backend()
...     )
```

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 or generated which implements the *RSAPrivateKeyWithSerialization* interface 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
... )
>>> 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, default_backend())
>>> 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,
...     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, default_backend())
>>> 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**
New in version 0.2.

name

class cryptography.hazmat.primitives.asymmetric.padding.**PSS**(*mgf*, *salt_length*)
New 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.MAX_LENGTH`.

MAX_LENGTH
Pass this attribute to `salt_length` to get the maximum salt length available.

class cryptography.hazmat.primitives.asymmetric.padding.**OAEP**(*mgf*, *algorithm*, *label*)
New 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.

class cryptography.hazmat.primitives.asymmetric.padding.**PKCS1v15**
New 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.

cryptography.hazmat.primitives.asymmetric.padding.**calculate_max_pss_salt_length**(*key*, *hash_algorithm*)
New 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.**MGF1**(*algorithm*)

New 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*)

New 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(*backend*)

Parameters **backend** – An instance of *RSABackend*.

Returns A new instance of *RSAPublicKey*.

class cryptography.hazmat.primitives.asymmetric.rsa.**RSAPrivateNumbers**(*p*, *q*, *d*, *dmp1*, *dmq1*, *iqmp*, *public_numbers*)

New 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(*backend*)

Parameters **backend** – A new instance of *RSABackend*.

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*)

New 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*)

New 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*)
New in version 0.4.

Computes the `dmq1` parameter from the RSA private exponent (`d`) and prime `q`.

cryptography.hazmat.primitives.asymmetric.rsa.**rsa_recover_prime_factors**(*n*, *e*, *d*)
New 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**
New in version 0.2.

An RSA private key. An RSA private key that is not an *opaque key* also implements *RSAPrivateKeyWithSerialization* to provide serialization methods.

decrypt(*ciphertext*, *padding*)
New in version 0.4.

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*)
New 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*) – 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.

class `cryptography.hazmat.primitives.asymmetric.rsa.`**RSAPrivateKeyWithSerialization**

New in version 0.8.

This interface contains additional methods relating to serialization. Any object with this interface also has all the methods from *RSAPrivateKey*.

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* 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**

New in version 0.2.

An RSA public key.

encrypt(*plaintext*, *padding*)

New 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.

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*)
New 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*) – The signature to verify.
- **data** (*bytes*) – 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.

Raises ***cryptography.exceptions.InvalidSignature*** – If the signature does not validate.

class cryptography.hazmat.primitives.asymmetric.rsa.**RSAPublicKeyWithSerialization**
New in version 0.8.

Alias for *RSAPublicKey*.

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

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.backends import default_backend
>>> 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,
...                                     backend=default_backend())
>>> # 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,
...     salt=None,
...     info=b'handshake data',
...     backend=default_backend()
... ).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',
...     backend=default_backend()
... ).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.backends import default_backend
>>> 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,
...                                     backend=default_backend())
>>> # 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(
```

(continues on next page)

(continued from previous page)

```
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=None,
...     info=b'handshake data',
...     backend=default_backend()
... ).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',
...     backend=default_backend()
... ).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(default_backend())
peer_public_numbers = dh.DHPublicNumbers(y, pn)
peer_public_key = peer_public_numbers.public_key(default_backend())
```

See also the *DHBackend* API for additional functionality.

Group parameters

cryptography.hazmat.primitives.asymmetric.dh.**generate_parameters**(*generator*, *key_size*, *backend*)

New in version 1.7.

Generate a new DH parameter group for use with `backend`.

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.
- **backend** – A *DHBackend* instance.

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**

New 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*)
New 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.

class cryptography.hazmat.primitives.asymmetric.dh.**DHParametersWithSerialization**
New in version 1.7.

Alias for *DHParameters*.

Key interfaces

class cryptography.hazmat.primitives.asymmetric.dh.**DHPrivateKey**
New in version 1.7.

A DH private key that is not an *opaque key* also implements *DHPrivateKeyWithSerialization* to provide serialization methods.

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*)
New 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.

class cryptography.hazmat.primitives.asymmetric.dh.**DHPrivateKeyWithSerialization**
New in version 1.7.

This interface contains additional methods relating to serialization. Any object with this interface also has all the methods from *DHPrivateKey*.

private_numbers()
Return the numbers that make up this private key.

Returns A *DHPrivateNumbers*.

private_bytes(*encoding*, *format*, *encryption_algorithm*)
New 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**
New 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*)
New 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.

class cryptography.hazmat.primitives.asymmetric.dh.**DHPublicKeyWithSerialization**
New in version 1.7.

Alias for *DHPublicKey*.

Numbers

class cryptography.hazmat.primitives.asymmetric.dh.**DHParameterNumbers**(*p*, *g*, *q=None*)
New 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

New in version 1.8.

Type int

p subgroup order value.

parameters(*backend*)

New in version 1.7.

Parameters **backend** – An instance of *DHBackend*.

Returns A new instance of *DHParameters*.

class cryptography.hazmat.primitives.asymmetric.dh.**DHPrivateNumbers**(*x*, *public_numbers*)

New 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(*backend*)

New in version 1.7.

Parameters **backend** – An instance of *DHBackend*.

Returns A new instance of *DHPrivateKey*.

class cryptography.hazmat.primitives.asymmetric.dh.**DHPublicNumbers**(*y*, *parameter_numbers*)

New 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(*backend*)

New in version 1.7.

Parameters **backend** – An instance of *DHBackend*.

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

DSA is a public-key algorithm for signing messages.

Generation

cryptography.hazmat.primitives.asymmetric.dsa.**generate_private_key**(*key_size*, *backend*)

New in version 0.5.

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 or 3072. For keys generated in 2015 this should be at least 2048 (See page 41). Note that some applications (such as SSH) have not yet gained support for larger key sizes specified in FIPS 186-3 and are still restricted to only the 1024-bit keys specified in FIPS 186-2.
- **backend** – An instance of *DSABackend*.

Returns An instance of *DSAPrivateKey*.

Raises ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided backend does not implement *DSABackend*

cryptography.hazmat.primitives.asymmetric.dsa.**generate_parameters**(*key_size*, *backend*)

New in version 0.5.

Generate DSA parameters using the provided backend.

Parameters

- **key_size** (*int*) – The length of *q*. It should be either 1024, 2048 or 3072. For keys generated in 2015 this should be at least 2048 (See page 41). Note that some applications (such as SSH) have not yet gained support for larger key sizes specified in FIPS 186-3 and are still restricted to only the 1024-bit keys specified in FIPS 186-2.
- **backend** – An instance of *DSABackend*.

Returns An instance of *DSAParameters*.

Raises ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided backend does not implement *DSABackend*

Signing

Using a *DSAPrivateKey* instance.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import dsa
>>> private_key = dsa.generate_private_key(
...     key_size=1024,
...     backend=default_backend()
... )
>>> 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, default_backend())
>>> 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, default_backend())
>>> 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*)

New 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(*backend*)

Parameters **backend** – An instance of *DSABackend*.

Returns A new instance of *DSAParameters*.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPublicNumbers**(*y*, *parameter_numbers*)

New 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(*backend*)

Parameters **backend** – An instance of *DSABackend*.

Returns A new instance of *DSAPublicKey*.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPrivateNumbers**(*x*, *public_numbers*)

New 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(*backend*)

Parameters **backend** – An instance of *DSABackend*.

Returns A new instance of *DSAPrivateKey*.

Key interfaces

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAParameters**

New in version 0.3.

DSA parameters.

generate_private_key()

New 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*.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAParametersWithNumbers**

New in version 0.5.

Extends *DSAParameters*.

parameter_numbers()

Create a *DSAParameterNumbers* object.

Returns A *DSAParameterNumbers* instance.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPrivateKey**

New in version 0.3.

A DSA private key. A DSA private key that is not an *opaque key* also implements *DSAPrivateKeyWithSerialization* to provide serialization methods.

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 *q*.

sign(*data*, *algorithm*)
New 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*) – 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.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPrivateKeyWithSerialization**
New in version 0.8.

This interface contains additional methods relating to serialization. Any object with this interface also has all the methods from *DSAPrivateKey*.

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**
New in version 0.3.

A DSA public key.

key_size

Type int

The bit length of *q*.

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*)
New 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*) – The signature to verify.
- **data** (*bytes*) – The message string that was signed.
- **algorithm** – An instance of *HashAlgorithm* or *Prehashed* if the `data` you want to sign has already been hashed.

Raises ***cryptography.exceptions.InvalidSignature*** – If the signature does not validate.

class cryptography.hazmat.primitives.asymmetric.dsa.**DSAPublicKeyWithSerialization**
New in version 0.8.

Alias for *DSAPublicKey*.

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.backends import default_backend
>>> 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, backend=default_
↪backend())
>>> if isinstance(key, rsa.RSAPrivateKey):
...     signature = sign_with_rsa_key(key, message)
... elif isinstance(key, dsa.DSAPrivateKey):
```

(continues on next page)

(continued from previous page)

```
...     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 in 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*, *backend*)

New in version 0.6.

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** – The password to use to decrypt the data. Should be `None` if the private key is not encrypted.
- **backend** – An instance of *PEMSerializationBackend*.

Returns One of *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 is of a type that is not supported by the backend or if the key is encrypted with a symmetric cipher that is not supported by the backend.

cryptography.hazmat.primitives.serialization.**load_pem_public_key**(*data*, *backend*)

New 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, backend=default_backend())
>>> isinstance(key, rsa.RSAPublicKey)
True
```

Parameters

- **data** (*bytes*) – The PEM encoded key data.
- **backend** – An instance of *PEMSerializationBackend*.

Returns One of *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 is of a type that is not supported by the backend.

`cryptography.hazmat.primitives.serialization.`**load_pem_parameters**(*data*, *backend*)

New in version 2.0.

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, backend=default_
↪backend())
>>> isinstance(parameters, dh.DHParameters)
True
```

Parameters

- **data** (*bytes*) – The PEM encoded parameters data.
- **backend** – An instance of *PEMSerializationBackend*.

Returns Currently only *DHParameters* supported.

Raises

- **ValueError** – If the PEM data's structure could not be decoded successfully.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the serialized parameters is of a type that is not supported by the backend.

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*, *backend*)

New 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.
- **backend** – An instance of *DERSerializationBackend*.

Returns One of *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 is of a type that is not supported by the backend or if the key is encrypted with a symmetric cipher that is not supported by the backend.

```
>>> from cryptography.hazmat.backends import default_backend
>>> 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, backend=default_backend())
>>> isinstance(key, rsa.RSAPrivateKey)
True
```

cryptography.hazmat.primitives.serialization.**load_der_public_key**(*data*, *backend*)

New 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.
- **backend** – An instance of *DERSerializationBackend*.

Returns One of *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 is of a type that is not supported by the backend.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> from cryptography.hazmat.primitives.serialization import load_der_public_key
```

(continues on next page)

(continued from previous page)

```
>>> key = load_der_public_key(public_der_data, backend=default_backend())
>>> isinstance(key, rsa.RSAPublicKey)
True
```

cryptography.hazmat.primitives.serialization.**load_der_parameters**(*data*, *backend*)

New in version 2.0.

Deserialize parameters from DER encoded data to one of the supported asymmetric parameters types.

Parameters

- **data** (*bytes*) – The DER encoded parameters data.
- **backend** – An instance of *DERSerializationBackend*.

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 is of a type that is not supported by the backend.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.primitives.serialization import load_der_parameters
>>> parameters = load_der_parameters(parameters_der_data, backend=default_
↪backend())
>>> 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.**load_ssh_public_key**(*data*, *backend*)

New in version 0.7.

Deserialize a public key from OpenSSH (**RFC 4253**) encoded data to an instance of the public key type for the specified backend.

Parameters

- **data** (*bytes*) – The OpenSSH encoded key data.

- **backend** – A backend which implements *RSABackend*, *DSABackend*, or *EllipticCurveBackend* depending on the key's type.

Returns One of *RSAPublicKey*, *DSAPublicKey*, *EllipticCurvePublicKey* , or *Ed25519PublicKey*, 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.

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.`**load_key_and_certificates**(*data*, *password*, *backend*)

New 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.
- **backend** – A backend instance.

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.

Serialization Formats

class `cryptography.hazmat.primitives.serialization.`**PrivateFormat**

New in version 0.8.

An enumeration for private key formats. Used with the `private_bytes` method available on *RSAPrivateKeyWithSerialization* , *EllipticCurvePrivateKeyWithSerialization* , *DHPrivateKeyWithSerialization* and *DSAPrivateKeyWithSerialization*.

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
New in version 2.5.

A raw format used by *X448 key exchange*. It is a binary format and is invalid for other key types.

class cryptography.hazmat.primitives.serialization.**PublicFormat**
New in version 0.8.

An enumeration for public key formats. Used with the `public_bytes` method available on *RSAPublicKeyWithSerialization* , *EllipticCurvePublicKeyWithSerialization* , *DHPublicKeyWithSerialization* , and *DSAPublicKeyWithSerialization*.

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
New 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
New in version 2.5.

A raw format used by *X448 key exchange*. It is a binary format and is invalid for other key types.

CompressedPoint
New 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
New 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**
New in version 2.0.

An enumeration for parameters formats. Used with the `parameter_bytes` method available on *DHParametersWithSerialization*.

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 *RSAPrivateKeyWithSerialization* , *EllipticCurvePrivateKeyWithSerialization* , *DHPrivateKeyWithSerialization*, *DSAPrivateKeyWithSerialization*, and *X448PrivateKey* as well as `public_bytes` on *RSAPublicKey*, *DHPublicKey*, *EllipticCurvePublicKey*, and *X448PublicKey*.

PEM
New in version 0.8.

For PEM format. This is a base64 format with delimiters.

DER
New in version 0.9.

For DER format. This is a binary format.

OpenSSH
New in version 1.4.

The format used by OpenSSH public keys. This is a text format.

Raw
New in version 2.5.

A raw format used by *X448 key exchange*. It is a binary format and is invalid for other key types.

X962
New in version 2.5.

The format used by elliptic curve point encodings. This is a binary 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 *RSAPrivateKeyWithSerialization* , *EllipticCurvePrivateKeyWithSerialization* , *DHPrivateKeyWithSerialization* and *DSAPrivateKeyWithSerialization*. All other classes in this section represent the available choices for encryption and have this interface. They are used with *private_bytes*.

class cryptography.hazmat.primitives.serialization.**BestAvailableEncryption**(*password*)
Encrypt using the best available encryption for a given key's backend. This is a curated encryption choice and the algorithm may change over time.

Parameters **password** (*bytes*) – The password to use for encryption.

class cryptography.hazmat.primitives.serialization.**NoEncryption**
Do not encrypt.

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.**Prehashed**(*algorithm*)
New 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.backends import default_backend
>>> 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,
...     backend=default_backend()
... )
>>> prehashed_msg = hashlib.sha256(b"A message I want to sign").digest()
>>> signature = private_key.sign(
```

(continues on next page)

(continued from previous page)

```
...     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.

2.3.3 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, see Coda Hale's blog post about the timing attacks on KeyCzar and Java's `MessageDigest.isEqual()`.

`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.

2.3.4 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.

class cryptography.hazmat.primitives.kdf.pbkdf2.**PBKDF2HMAC**(*algorithm*, *length*, *salt*, *iterations*, *backend*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> # Salts should be randomly generated
>>> salt = os.urandom(16)
>>> # derive
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=100000,
...     backend=backend
... )
>>> key = kdf.derive(b"my great password")
>>> # verify
>>> kdf = PBKDF2HMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     iterations=100000,
...     backend=backend
... )
>>> 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. See OWASP's Password Storage Cheat Sheet for more detailed recommendations if you intend to use this for password storage.
- **backend** – An instance of *PBKDF2HMACBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *PBKDF2HMACBackend*
- **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()* or *verify()* is called more than once.
- **TypeError** – This exception is raised if `key_material` is not `bytes`.

This generates and returns a new key from the supplied password.

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()* 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.

[1] See NIST SP 800-132.

class cryptography.hazmat.primitives.kdf.hkdf.**HKDF**(*algorithm*, *length*, *salt*, *info*, *backend*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> salt = os.urandom(16)
>>> info = b"hkdf-example"
>>> hkdf = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     info=info,
...     backend=backend
... )
>>> key = hkdf.derive(b"input key")
>>> hkdf = HKDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     info=info,
...     backend=backend
... )
>>> 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.
- **info** (*bytes*) – Application specific context information. If `None` is explicitly passed an empty byte string will be used.
- **backend** – An instance of *HMACBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HMACBackend*

- **TypeError** – This exception is raised if `salt` or `info` 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`.

Derives a new key from the input key material by performing both the extract and expand operations.

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()* 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*, *backend*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> info = b"hkdf-example"
>>> key_material = os.urandom(16)
>>> hkdf = HKDFExpand(
...     algorithm=hashes.SHA256(),
...     length=32,
...     info=info,
...     backend=backend
... )
>>> key = hkdf.derive(key_material)
>>> hkdf = HKDFExpand(
...     algorithm=hashes.SHA256(),
...     length=32,
```

(continues on next page)

(continued from previous page)

```
...     info=info,
...     backend=backend
... )
>>> 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.
- **backend** – An instance of *HMACBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HMACBackend*
- **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`.

Derives a new key from the input key material by performing both the extract and expand operations.

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()* 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.

class cryptography.hazmat.primitives.kdf.concatkdf.**ConcatKDFHash**(*algorithm*, *length*, *otherinfo*, *backend*)

New in version 1.0.

ConcatKDFHash (Concatenation Key Derivation Function) is defined by the NIST Special Publication NIST SP 800-56Ar2 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> otherinfo = b"concatkdf-example"
>>> ckdf = ConcatKDFHash(
...     algorithm=hashes.SHA256(),
...     length=32,
...     otherinfo=otherinfo,
...     backend=backend
... )
>>> key = ckdf.derive(b"input key")
>>> ckdf = ConcatKDFHash(
...     algorithm=hashes.SHA256(),
...     length=32,
...     otherinfo=otherinfo,
...     backend=backend
... )
>>> 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.
- **backend** – An instance of *HashBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HashBackend*
- **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`.

Derives a new key from the input key material.

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()* 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*, *backend*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> salt = os.urandom(16)
>>> otherinfo = b"concatkdf-example"
>>> ckdf = ConcatKDFHMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     otherinfo=otherinfo,
...     backend=backend
... )
>>> key = ckdf.derive(b"input key")
>>> ckdf = ConcatKDFHMAC(
...     algorithm=hashes.SHA256(),
...     length=32,
...     salt=salt,
...     otherinfo=otherinfo,
...     backend=backend
... )
>>> 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.
- **backend** – An instance of *HMACBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HMACBackend*
- **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`.

Derives a new key from the input key material.

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()* 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.x963kdf.**X963KDF**(*algorithm*, *length*, *otherinfo*, *backend*)

New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> sharedinfo = b"ANSI X9.63 Example"
>>> xkdf = X963KDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     sharedinfo=sharedinfo,
...     backend=backend
... )
>>> key = xkdf.derive(b"input key")
>>> xkdf = X963KDF(
...     algorithm=hashes.SHA256(),
...     length=32,
...     sharedinfo=sharedinfo,
...     backend=backend
... )
>>> 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.
- **backend** – A cryptography backend *HashBackend* instance.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HashBackend*
- **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`.

Derives a new key from the input key material.

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()* 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.**KBKDFHMAC**(*algorithm*, *mode*, *length*, *rlen*, *llen*, *location*, *label*, *context*, *fixed*, *backend*)

New 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
... )
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> 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,
...     backend=backend
... )
>>> 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,
...     backend=backend
... )
>>> 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.
- **backend** – A cryptography backend *HashBackend* instance.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HashBackend*
- **TypeError** – This exception is raised if `label` or `context` is not `bytes`. Also raised if `rlen` or `llen` 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`.

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`.

Derives a new key from the input key material.

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()* 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.

class cryptography.hazmat.primitives.kdf.scrypt.**Scrypt**(*salt*, *length*, *n*, *r*, *p*, *backend*)
New 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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> salt = os.urandom(16)
>>> # derive
>>> kdf = Scrypt(
...     salt=salt,
...     length=32,
...     n=2**14,
...     r=8,
...     p=1,
...     backend=backend
... )
>>> key = kdf.derive(b"my great password")
>>> # verify
>>> kdf = Scrypt(
...     salt=salt,
...     length=32,
...     n=2**14,
...     r=8,
...     p=1,
...     backend=backend
... )
>>> 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).

Parameters **backend** – An instance of *ScryptBackend*.

Raises

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *ScryptBackend*
- **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()* or *verify()* is called more than once.

This generates and returns a new key from the supplied password.

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()* 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.

Interface

class cryptography.hazmat.primitives.kdf.**KeyDerivationFunction**

New 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()* or *verify()* is called more than once.

This generates and returns a new key from the supplied key material.

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()* 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.

2.3.5 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*, *backend*)

New 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.
- **backend** – A *CipherBackend* instance that supports *AES*.

Return bytes The wrapped key as bytes.

`cryptography.hazmat.primitives.keywrap.`**aes_key_unwrap**(*wrapping_key*, *wrapped_key*, *backend*)

New 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.
- **backend** – A *CipherBackend* instance that supports *AES*.

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*, *backend*)

New 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.
- **backend** – A *CipherBackend* instance that supports *AES*.

Return bytes The wrapped key as bytes.

cryptography.hazmat.primitives.keywrap.**aes_key_unwrap_with_padding**(*wrapping_key*, *wrapped_key*, *backend*)

New 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.
- **backend** – A *CipherBackend* instance that supports *AES*.

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.

2.3.6 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*, *backend*)

New in version 0.4.

CMAC objects take a *BlockCipherAlgorithm* instance.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import cmac
>>> from cryptography.hazmat.primitives.ciphers import algorithms
>>> c = cmac.CMAC(algorithms.AES(key), backend=default_backend())
>>> c.update(b"message to authenticate")
>>> c.finalize()
b'CT\x1d\xc8\x0e\x15\xbe4e\xdb\xb6\x84\xca\xd9Xk'
```

If the backend doesn't support the requested `algorithm` an *UnsupportedAlgorithm* exception will be raised.

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), backend=default_backend())
>>> 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*.
- **backend** – An instance of *CMACBackend*.

Raises

- **TypeError** – This is raised if the provided `algorithm` is not an instance of *BlockCipherAlgorithm*

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *CMACBackend*

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*, *backend*)

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.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes, hmac
>>> h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
>>> h.update(b"message to hash")
>>> h.finalize()
b'#F\xdaI\x8b"e\xc4\xf1\xbb\x9a\x8fc\xff\xf5\xdex.\xbc\xcd/+\x8a\x86\x1d\x84\
↪'\xc3\xa6\x1d\xd8J'
```

If the backend doesn't support the requested `algorithm` an *`UnsupportedAlgorithm`* exception will be raised.

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(), backend=default_backend())
>>> h.update(b"message to hash")
>>> h.verify(b"an incorrect signature")
Traceback (most recent call last):
...
cryptography.exceptions.InvalidSignature: Signature did not match digest.
```

Parameters

- **`key`** (*bytes-like*) – Secret key as `bytes`.
- **`algorithm`** – An *`HashAlgorithm`* instance such as those described in *Cryptographic Hashes*.
- **`backend`** – An *`HMACBackend`* instance.

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

`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*)

New 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
>>> 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*) – Secret key as `bytes`.

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*** –

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.

2.3.7 Message digests (Hashing)

class cryptography.hazmat.primitives.hashes.**Hash**(*algorithm*, *backend*)

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.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
>>> 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'
```

If the backend doesn't support the requested `algorithm` an *UnsupportedAlgorithm* exception will be raised.

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 in *below*.
- **backend** – A *HashBackend* instance.

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

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.

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**

New 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**

New 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 1.1.0.

---

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**

New 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**

New 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**

New 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**

New 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.

class cryptography.hazmat.primitives.hashes.**SHAKE128**(*digest_size*)
New 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.

Parameters **digest_size** (*int*) – The length of output desired. Must be greater than zero.

Raises **ValueError** – If the digest_size is invalid.

class cryptography.hazmat.primitives.hashes.**SHAKE256**(*digest_size*)
New 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.

Parameters **digest_size** (*int*) – The length of output desired. Must be greater than zero.

Raises **ValueError** – If the digest_size is invalid.

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.

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.**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)*.

2.3.8 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.

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

class cryptography.hazmat.primitives.ciphers.**Cipher**(*algorithm*, *mode*, *backend*)

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
>>> from cryptography.hazmat.backends import default_backend
>>> backend = default_backend()
>>> key = os.urandom(32)
>>> iv = os.urandom(16)
>>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
>>> 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

- **algorithms** – A *CipherAlgorithm* instance such as those described *below*.
- **mode** – A *Mode* instance such as those described *below*.
- **backend** – A *CipherBackend* instance.

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

encryptor()

Returns An encrypting *CipherContext* instance.

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

decryptor()

Returns A decrypting *CipherContext* instance.

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

Algorithms

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

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

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

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

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

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

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

New in version 2.1.

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

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

Parameters

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

```
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> from cryptography.hazmat.backends import default_backend
>>> nonce = os.urandom(16)
```

(continues on next page)

(continued from previous page)

```
>>> algorithm = algorithms.ChaCha20(key, nonce)
>>> cipher = Cipher(algorithm, mode=None, backend=default_backend())
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message")
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct)
b'a secret message'
```

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

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

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

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

New in version 0.2.

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

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

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

New in version 0.4.

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

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

Weak ciphers

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

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

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

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

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

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

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

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

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

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

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

Modes

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

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

Padding is required when using this mode.

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

A good construction looks like:

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

While the following is bad and will leak information:

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

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

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

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

This mode does not require padding.

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

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

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

This mode does not require padding.

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

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

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

This mode does not require padding.

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

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

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

This mode does not require padding.

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

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

> **Danger:** If you are encrypting data that can fit into memory you should strongly consider using *`AESGCM`* instead of this.
>
> When using this mode you **must** not use the decrypted data until the appropriate finalization method (*`finalize()`* or *`finalize_with_tag()`*) has been called. GCM provides **no** guarantees of ciphertext integrity until decryption is complete.

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

This mode does not require padding.

Parameters **initialization_vector** (*bytes-like*) – Must be unique, a *nonce*. They do not need to be kept secret and they can be included in a transmitted message. NIST recommends a

96-bit IV length for performance critical situations but it can be up to 2^{64} - 1 *bits*. Do not reuse an `initialization_vector` with a given `key`.

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

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

Parameters

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

Raises

- **ValueError** – This is raised if `len(tag) < min_tag_length` or the `initialization_vector` is too short.
- **NotImplementedError** – This is raised if the version of the OpenSSL backend used is 1.0.1 or earlier.

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

```
import os

from cryptography.hazmat.backends import default_backend
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),
        backend=default_backend()
    ).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()
```

(continues on next page)

(continued from previous page)

```
    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),
        backend=default_backend()
    ).decryptor()

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

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

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

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

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

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

New in version 2.1.

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

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

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

This mode does not require padding.

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

Insecure modes

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

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

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

Padding is required when using this mode.

Interfaces

class cryptography.hazmat.primitives.ciphers.**CipherContext**

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

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

update(*data*)

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

Return bytes Returns the data that was encrypted or decrypted.

Raises ***cryptography.exceptions.AlreadyFinalized*** – See *finalize()*

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

update_into(*data*, *buf*)

New in version 1.8.

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

Parameters

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

Return int Number of bytes written.

Raises

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

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

finalize()

Return bytes Returns the remainder of the data.

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

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

class cryptography.hazmat.primitives.ciphers.**AEADCipherContext**

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

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

---

authenticate_additional_data(*data*)

Parameters **data** (*bytes-like*) – Any data you wish to authenticate but not encrypt.

Raises *AlreadyFinalized*

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

When creating an encryption context using `encryptor` on a `Cipher` object with an AEAD mode such as *GCM*

an object conforming to both the `AEADEncryptionContext` and `AEADCipherContext` interfaces will be returned. This interface provides one additional attribute `tag`. `tag` can only be obtained after `finalize` has been called.

tag

Return bytes Returns the tag value as bytes.

Raises *NotYetFinalized* if called before the context is finalized.

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

New in version 1.9.

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

finalize_with_tag(*tag*)

Note: This method is not supported when compiled against OpenSSL 1.0.1.

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.
- **NotImplementedError** – This is raised if the version of the OpenSSL backend used is 1.0.1 or earlier.

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

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

A named symmetric encryption algorithm.

name

Type str

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

key_size

Type int

The number of *bits* in the key being used.

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

A block cipher algorithm.

block_size

Type int

The number of *bits* in a block.

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

class cryptography.hazmat.primitives.ciphers.modes.**Mode**
A named cipher mode.

name

Type str

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

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

validate_for_algorithm(*algorithm*)

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

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

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

class cryptography.hazmat.primitives.ciphers.modes.**ModeWithInitializationVector**
A cipher mode with an initialization vector.

initialization_vector

Type *bytes-like*

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

class cryptography.hazmat.primitives.ciphers.modes.**ModeWithNonce**
A cipher mode with a nonce.

nonce

Type *bytes-like*

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

class cryptography.hazmat.primitives.ciphers.modes.**ModeWithAuthenticationTag**
A cipher mode with an authentication tag.

tag

Type *bytes-like*

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

class cryptography.hazmat.primitives.ciphers.modes.**ModeWithTweak**
New in version 2.1.

A cipher mode with a tweak.

tweak

Type *bytes-like*

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

Exceptions

class cryptography.exceptions.**InvalidTag**
This is raised if an authenticated encryption tag fails to verify during decryption.

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.

2.3.9 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
b'1111111111111111'
>>> padded_data += padder.finalize()
>>> padded_data
b'11111111111111112222222222\x06\x06\x06\x06\x06\x06'
>>> unpadder = padding.PKCS7(128).unpadder()
>>> data = unpadder.update(padded_data)
>>> data
b'1111111111111111'
>>> data + unpadder.finalize()
b'11111111111111112222222222'
```

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*)
New in version 1.3.

ANSI X.923 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"11111111111111112222222222")
>>> padded_data
b'1111111111111111'
>>> padded_data += padder.finalize()
>>> padded_data
b'11111111111111112222222222\x00\x00\x00\x00\x00\x06'
>>> unpadder = padding.ANSIX923(128).unpadder()
>>> data = unpadder.update(padded_data)
>>> data
b'1111111111111111'
>>> data + unpadder.finalize()
b'11111111111111112222222222'
```

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*) – 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.

2.3.10 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).

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.

class cryptography.hazmat.primitives.twofactor.hotp.**HOTP**(*key*, *length*, *algorithm*, *backend*, *enforce_key_length=True*)

New 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.backends import default_backend
>>> from cryptography.hazmat.primitives.twofactor.hotp import HOTP
>>> from cryptography.hazmat.primitives.hashes import SHA1
>>> key = os.urandom(20)
>>> hotp = HOTP(key, 6, SHA1(), backend=default_backend())
>>> 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.
- **backend** – A *HMACBackend* instance.
- **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 *HOTP*.

  New 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.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HMACBackend*

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*)

New in version 1.0.

Parameters

- **account_name** (*text*) – The display name of account, such as `'Alice Smith'` or `'alice@example.com'`.
- **issuer** (*text* 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, default_backend())
```

(continues on next page)

(continued from previous page)

```
    for count in range(counter, counter + look_ahead):
        try:
            otp.verify(hotp, count)
            correct_counter = count
        except InvalidToken:
            pass

    return correct_counter
```

class cryptography.hazmat.primitives.twofactor.totp.**TOTP**(*key*, *length*, *algorithm*, *time_step*, *backend*, *enforce_key_length=True*)

TOTP objects take a `key`, `length`, `algorithm` and `time_step` parameter. The `key` should be *randomly generated 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.backends import default_backend
>>> 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, backend=default_backend())
>>> 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.
- **backend** – A *HMACBackend* instance.
- **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*.

  New 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.

- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised if the provided `backend` does not implement *HMACBackend*

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*)

New in version 1.0.

Parameters

- **account_name** (*text*) – The display name of account, such as `'Alice Smith'` or `'alice@example.com'`.
- **issuer** (*text* 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.

2.4 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.

2.5 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)
```

This will use `/dev/urandom` on UNIX platforms, and `CryptGenRandom` on Windows.

If you need your random number as an integer (for example, for *serial_number()*), you can use `int.from_bytes` to convert the result of `os.urandom`:

```
>>> serial = int.from_bytes(os.urandom(20), byteorder="big")
```

Starting with Python 3.6 the 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.

2.6 Backends

2.6.1 Getting a backend

`cryptography` was originally designed to support multiple backends, but this design has been deprecated.

You can get the default backend by calling *default_backend()*.

cryptography.hazmat.backends.**default_backend**()

Returns An object that provides at least *CipherBackend*, *HashBackend*, and *HMACBackend*.

2.6.2 Individual backends

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.

OpenSSL backend

The OpenSSL C library. Cryptography supports OpenSSL version 1.0.1 and greater.

`cryptography.hazmat.backends.openssl.`**`backend`**

This is the exposed API for the OpenSSL backend.

It implements the following interfaces:

- *CipherBackend*
- *CMACBackend*
- *DERSerializationBackend*
- *DHBackend*
- *DSABackend*
- *EllipticCurveBackend*
- *HashBackend*
- *HMACBackend*
- *PBKDF2HMACBackend*
- *RSABackend*
- *PEMSerializationBackend*
- *X509Backend*

It also implements the following interface for OpenSSL versions `1.1.0` and above.

- *ScryptBackend*

It also exposes the following:

`name`

The string name of this backend: `"openssl"`

`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``()`

New 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.

`activate_osrandom_engine``()`

Activates the OS random engine. This will effectively disable OpenSSL's default CSPRNG.

osrandom_engine_implementation()
New in version 1.7.

Returns the implementation of OS random engine.

activate_builtin_random()
This will activate the default OpenSSL CSPRNG.

OS random engine

By default OpenSSL uses a user-space CSPRNG that is seeded from system random (`/dev/urandom` or `CryptGenRandom`). This CSPRNG is not reseeded automatically when a process calls `fork()`. This can result in situations where two different processes can return similar or identical keys and compromise the security of the system.

The approach this project has chosen to mitigate this vulnerability is to include an engine that replaces the OpenSSL default CSPRNG with one that sources its entropy from `/dev/urandom` on UNIX-like operating systems and uses `CryptGenRandom` on Windows. This method of pulling from the system pool allows us to avoid potential issues with initializing the RNG as well as protecting us from the `fork()` weakness.

This engine is **active** by default when importing the OpenSSL backend. When active this engine will be used to generate all the random data OpenSSL requests.

When importing only the binding it is added to the engine list but **not activated**.

OS random sources

On macOS and FreeBSD `/dev/urandom` is an alias for `/dev/random`. The implementation on macOS uses the Yarrow algorithm. FreeBSD uses the Fortuna algorithm.

On Windows the implementation of `CryptGenRandom` depends on which version of the operation system you are using. See the Microsoft documentation for more details.

Linux uses its own PRNG design. `/dev/urandom` is a non-blocking source seeded from the same pool as `/dev/random`.

Windows	`CryptGenRandom()`
Linux >= 3.17 with working `SYS_getrandom` syscall	`getrandom(GRND_NONBLOCK)`
OpenBSD >= 5.6	`getentropy()`
BSD family (including macOS 10.12+) with `SYS_getentropy` in `sys/syscall.h`	`getentropy()`
fallback	`/dev/urandom` with cached file descriptor

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.

Backend interfaces

Backend implementations may provide a number of interfaces to support operations such as *Symmetric encryption*, *Message digests (Hashing)*, and *Hash-based message authentication codes (HMAC)*.

A specific `backend` may provide one or more of these interfaces.

class cryptography.hazmat.backends.interfaces.**CipherBackend**

A backend that provides methods for using ciphers for encryption and decryption.

The following backends implement this interface:

- *OpenSSL backend*

cipher_supported(*cipher*, *mode*)

Check if a `cipher` and `mode` combination is supported by this backend.

Parameters

- **cipher** – An instance of *CipherAlgorithm*.
- **mode** – An instance of *Mode*.

Returns `True` if the specified `cipher` and `mode` combination is supported by this backend, otherwise `False`

create_symmetric_encryption_ctx(*cipher*, *mode*)

Create a *CipherContext* that can be used for encrypting data with the symmetric `cipher` using the given `mode`.

Parameters

- **cipher** – An instance of *CipherAlgorithm*.
- **mode** – An instance of *Mode*.

Returns *CipherContext*

Raises **ValueError** – When tag is not None in an AEAD mode

create_symmetric_decryption_ctx(*cipher*, *mode*)

Create a *CipherContext* that can be used for decrypting data with the symmetric `cipher` using the given `mode`.

Parameters

- **cipher** – An instance of *CipherAlgorithm*.
- **mode** – An instance of *Mode*.

Returns *CipherContext*

Raises **ValueError** – When tag is None in an AEAD mode

class cryptography.hazmat.backends.interfaces.**HashBackend**

A backend with methods for using cryptographic hash functions.

The following backends implement this interface:

- *OpenSSL backend*

hash_supported(*algorithm*)

Check if the specified `algorithm` is supported by this backend.

Parameters **algorithm** – An instance of *HashAlgorithm*.

Returns `True` if the specified `algorithm` is supported by this backend, otherwise `False`.

create_hash_ctx(*algorithm*)

Create a *HashContext* that uses the specified `algorithm` to calculate a message digest.

Parameters **algorithm** – An instance of *HashAlgorithm*.

Returns *HashContext*

class cryptography.hazmat.backends.interfaces.**HMACBackend**

A backend with methods for using cryptographic hash functions as message authentication codes.

The following backends implement this interface:

- *OpenSSL backend*

hmac_supported(*algorithm*)

Check if the specified `algorithm` is supported by this backend.

Parameters **algorithm** – An instance of *HashAlgorithm*.

Returns `True` if the specified `algorithm` is supported for HMAC by this backend, otherwise `False`.

create_hmac_ctx(*key*, *algorithm*)

Create a *HashContext* that uses the specified `algorithm` to calculate a hash-based message authentication code.

Parameters

- **key** (*bytes*) – Secret key as `bytes`.
- **algorithm** – An instance of *HashAlgorithm*.

Returns *HashContext*

class cryptography.hazmat.backends.interfaces.**CMACBackend**

New in version 0.4.

A backend with methods for using CMAC

cmac_algorithm_supported(*algorithm*)

Parameters **algorithm** – An instance of *BlockCipherAlgorithm*.

Returns Returns True if the block cipher is supported for CMAC by this backend

create_cmac_ctx(*algorithm*)

Create a context that uses the specified `algorithm` to calculate a message authentication code.

Parameters **algorithm** – An instance of *BlockCipherAlgorithm*.

Returns CMAC object.

class cryptography.hazmat.backends.interfaces.**PBKDF2HMACBackend**

New in version 0.2.

A backend with methods for using PBKDF2 using HMAC as a PRF.

The following backends implement this interface:

- *OpenSSL backend*

pbkdf2_hmac_supported(*algorithm*)

Check if the specified `algorithm` is supported by this backend.

Parameters **algorithm** – An instance of *HashAlgorithm*.

Returns `True` if the specified `algorithm` is supported for PBKDF2 HMAC by this backend, otherwise `False`.

derive_pbkdf2_hmac(*self*, *algorithm*, *length*, *salt*, *iterations*, *key_material*)

Parameters

- **algorithm** – An instance of *HashAlgorithm*.
- **length** (*int*) – The desired length of the derived key. Maximum is (2^{32} - 1) * `algorithm.digest_size`
- **salt** (*bytes*) – A salt.
- **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.
- **key_material** (*bytes*) – The key material to use as a basis for the derived key. This is typically a password.

Return bytes Derived key.

class `cryptography.hazmat.backends.interfaces.`**RSABackend**

New in version 0.2.

A backend with methods for using RSA.

generate_rsa_private_key(*public_exponent*, *key_size*)

Parameters

- **public_exponent** (*int*) – The public exponent of the new key. Often one of the small Fermat primes 3, 5, 17, 257 or 65537.
- **key_size** (*int*) – The length in bits of the modulus. Should be at least 2048.

Returns A new instance of *RSAPrivateKey*.

Raises **ValueError** – If the public_exponent is not valid.

rsa_padding_supported(*padding*)

Check if the specified `padding` is supported by the backend.

Parameters **padding** – An instance of *AsymmetricPadding*.

Returns `True` if the specified `padding` is supported by this backend, otherwise `False`.

generate_rsa_parameters_supported(*public_exponent*, *key_size*)

Check if the specified parameters are supported for key generation by the backend.

Parameters

- **public_exponent** (*int*) – The public exponent.
- **key_size** (*int*) – The bit length of the generated modulus.

load_rsa_private_numbers(*numbers*)

Parameters **numbers** – An instance of *RSAPrivateNumbers*.

Returns An instance of *RSAPrivateKey*.

Raises

- **ValueError** – This is raised when the values of `p`, `q`, `private_exponent`, `public_exponent`, or `modulus` do not match the bounds specified in **RFC 3447**.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised when any backend specific criteria are not met.

load_rsa_public_numbers(*numbers*)

Parameters **numbers** – An instance of *RSAPublicNumbers*.

Returns An instance of *RSAPublicKey*.

Raises

- **ValueError** – This is raised when the values of `public_exponent` or `modulus` do not match the bounds specified in **RFC 3447**.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised when any backend specific criteria are not met.

class `cryptography.hazmat.backends.interfaces.`**DSABackend**

New in version 0.4.

A backend with methods for using DSA.

generate_dsa_parameters(*key_size*)

Parameters **key_size** (*int*) – The length of the modulus in bits. It should be either 1024, 2048 or 3072. For keys generated in 2015 this should be at least 2048. Note that some applications (such as SSH) have not yet gained support for larger key sizes specified in FIPS 186-3 and are still restricted to only the 1024-bit keys specified in FIPS 186-2.

Returns A new instance of *DSAParameters*.

generate_dsa_private_key(*parameters*)

Parameters **parameters** – An instance of *DSAParameters*.

Returns A new instance of *DSAPrivateKey*.

Raises **ValueError** – This is raised if the key size is not one of 1024, 2048, or 3072.

generate_dsa_private_key_and_parameters(*key_size*)

Parameters **key_size** (*int*) – The length of the modulus in bits. It should be either 1024, 2048 or 3072. For keys generated in 2015 this should be at least 2048. Note that some applications (such as SSH) have not yet gained support for larger key sizes specified in FIPS 186-3 and are still restricted to only the 1024-bit keys specified in FIPS 186-2.

Returns A new instance of *DSAPrivateKey*.

Raises **ValueError** – This is raised if the key size is not supported by the backend.

dsa_hash_supported(*algorithm*)

Parameters **algorithm** – An instance of *HashAlgorithm*.

Returns `True` if the specified `algorithm` is supported by this backend, otherwise `False`.

dsa_parameters_supported(*p*, *q*, *g*)

Parameters

- **p** (*int*) – The p value of a DSA key.
- **q** (*int*) – The q value of a DSA key.
- **g** (*int*) – The g value of a DSA key.

Returns `True` if the given values of `p`, `q`, and `g` are supported by this backend, otherwise `False`.

load_dsa_parameter_numbers(*numbers*)

Parameters **numbers** – An instance of *DSAParameterNumbers*.

Returns An instance of *DSAParameters*.

Raises ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised when any backend specific criteria are not met.

load_dsa_private_numbers(*numbers*)

Parameters **numbers** – An instance of *DSAPrivateNumbers*.

Returns An instance of *DSAPrivateKey*.

Raises ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised when any backend specific criteria are not met.

load_dsa_public_numbers(*numbers*)

Parameters **numbers** – An instance of *DSAPublicNumbers*.

Returns An instance of *DSAPublicKey*.

Raises ***cryptography.exceptions.UnsupportedAlgorithm*** – This is raised when any backend specific criteria are not met.

class cryptography.hazmat.backends.interfaces.**EllipticCurveBackend**

New in version 0.5.

elliptic_curve_supported(*curve*)

Parameters **curve** – An instance of *EllipticCurve*.

Returns True if the elliptic curve is supported by this backend.

elliptic_curve_signature_algorithm_supported(*signature_algorithm*, *curve*)

Parameters

- **signature_algorithm** – An instance of *EllipticCurveSignatureAlgorithm*.
- **curve** – An instance of *EllipticCurve*.

Returns True if the signature algorithm and curve are supported by this backend.

generate_elliptic_curve_private_key(*curve*)

Parameters **curve** – An instance of *EllipticCurve*.

load_elliptic_curve_private_numbers(*numbers*)

Parameters **numbers** – An instance of *EllipticCurvePrivateNumbers*.

Returns An instance of *EllipticCurvePrivateKey*.

load_elliptic_curve_public_numbers(*numbers*)

Parameters **numbers** – An instance of *EllipticCurvePublicNumbers*.

Returns An instance of *EllipticCurvePublicKey*.

derive_elliptic_curve_private_key(*private_value*, *curve*)

Parameters

- **private_value** – A secret scalar value.
- **curve** – An instance of *EllipticCurve*.

Returns An instance of *EllipticCurvePrivateKey*.

class cryptography.hazmat.backends.interfaces.**PEMSerializationBackend**

New in version 0.6.

A backend with methods for working with any PEM encoded keys.

load_pem_private_key(*data*, *password*)

Parameters

- **data** (*bytes*) – PEM data to load.
- **password** (*bytes*) – The password to use if the data is encrypted. Should be `None` if the data is not encrypted.

Returns A new instance of the appropriate type of private key that the serialized data contains.

Raises

- **ValueError** – If the data could not be deserialized.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the data is encrypted with an unsupported algorithm.

load_pem_public_key(*data*)

Parameters **data** (*bytes*) – PEM data to load.

Returns A new instance of the appropriate type of public key serialized data contains.

Raises **ValueError** – If the data could not be deserialized.

load_pem_parameters(*data*)

New in version 2.0.

Parameters **data** (*bytes*) – PEM data to load.

Returns A new instance of the appropriate type of asymmetric parameters the serialized data contains.

Raises **ValueError** – If the data could not be deserialized.

class cryptography.hazmat.backends.interfaces.**DERSerializationBackend**

New in version 0.8.

A backend with methods for working with DER encoded keys.

load_der_private_key(*data*, *password*)

Parameters

- **data** (*bytes*) – DER data to load.
- **password** (*bytes*) – The password to use if the data is encrypted. Should be `None` if the data is not encrypted.

Returns A new instance of the appropriate type of private key that the serialized data contains.

Raises

- **ValueError** – If the data could not be deserialized.
- ***cryptography.exceptions.UnsupportedAlgorithm*** – If the data is encrypted with an unsupported algorithm.

load_der_public_key(*data*)

Parameters **data** (*bytes*) – DER data to load.

Returns A new instance of the appropriate type of public key serialized data contains.

Raises **ValueError** – If the data could not be deserialized.

load_der_parameters(*data*)

New in version 2.0.

Parameters **data** (*bytes*) – DER data to load.

Returns A new instance of the appropriate type of asymmetric parameters the serialized data contains.

Raises **ValueError** – If the data could not be deserialized.

class cryptography.hazmat.backends.interfaces.**X509Backend**

New in version 0.7.

A backend with methods for working with X.509 objects.

load_pem_x509_certificate(*data*)

Parameters **data** (*bytes*) – PEM formatted certificate data.

Returns An instance of *Certificate*.

load_der_x509_certificate(*data*)

Parameters **data** (*bytes*) – DER formatted certificate data.

Returns An instance of *Certificate*.

load_pem_x509_csr(*data*)

New in version 0.9.

Parameters **data** (*bytes*) – PEM formatted certificate signing request data.

Returns An instance of *CertificateSigningRequest*.

load_der_x509_csr(*data*)

New in version 0.9.

Parameters **data** (*bytes*) – DER formatted certificate signing request data.

Returns An instance of *CertificateSigningRequest*.

create_x509_csr(*builder*, *private_key*, *algorithm*)

New in version 1.0.

Parameters

- **builder** – An instance of *CertificateSigningRequestBuilder*.
- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* 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.
- **algorithm** – The *HashAlgorithm* that will be used to generate the request signature.

Returns A new instance of *CertificateSigningRequest*.

create_x509_certificate(*builder*, *private_key*, *algorithm*)

New in version 1.0.

Parameters

- **builder** – An instance of *CertificateBuilder*.
- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* that will be used to sign the certificate.
- **algorithm** – The *HashAlgorithm* that will be used to generate the certificate signature.

Returns A new instance of *Certificate*.

create_x509_crl(*builder*, *private_key*, *algorithm*)
New in version 1.2.

Parameters

- **builder** – An instance of *CertificateRevocationListBuilder*.
- **private_key** – The *RSAPrivateKey*, *DSAPrivateKey* or *EllipticCurvePrivateKey* that will be used to sign the CRL.
- **algorithm** – The *HashAlgorithm* that will be used to generate the CRL signature.

Returns A new instance of *CertificateRevocationList*.

create_x509_revoked_certificate(*builder*)
New in version 1.2.

Parameters **builder** – An instance of RevokedCertificateBuilder.

Returns A new instance of *RevokedCertificate*.

x509_name_bytes(*name*)
New in version 1.6.

Parameters **name** – An instance of *Name*.

Return bytes The DER encoded bytes.

class cryptography.hazmat.backends.interfaces.**DHBackend**
New in version 0.9.

A backend with methods for doing Diffie-Hellman key exchange.

generate_dh_parameters(*generator*, *key_size*)

Parameters

- **generator** (*int*) – The generator to use. Often 2 or 5.
- **key_size** (*int*) – The bit length of the prime modulus to generate.

Returns A new instance of *DHParameters*.

Raises **ValueError** – If `key_size` is not at least 512.

generate_dh_private_key(*parameters*)

Parameters **parameters** – An instance of *DHParameters*.

Returns A new instance of *DHPrivateKey*.

generate_dh_private_key_and_parameters(*generator*, *key_size*)

Parameters

- **generator** (*int*) – The generator to use. Often 2 or 5.
- **key_size** (*int*) – The bit length of the prime modulus to generate.

Returns A new instance of *DHPrivateKey*.

Raises **ValueError** – If `key_size` is not at least 512.

load_dh_private_numbers(*numbers*)

Parameters **numbers** – A *DHPrivateNumbers* instance.

Returns A new instance of *DHPrivateKey*.

Raises *cryptography.exceptions.UnsupportedAlgorithm* – This is raised when any backend specific criteria are not met.

load_dh_public_numbers(*numbers*)

Parameters **numbers** – A *DHPublicNumbers* instance.

Returns A new instance of *DHPublicKey*.

Raises *cryptography.exceptions.UnsupportedAlgorithm* – This is raised when any backend specific criteria are not met.

load_dh_parameter_numbers(*numbers*)

Parameters **numbers** – A *DHParameterNumbers* instance.

Returns A new instance of *DHParameters*.

Raises *cryptography.exceptions.UnsupportedAlgorithm* – This is raised when any backend specific criteria are not met.

dh_parameters_supported(*p*, *g*, *q=None*)

Parameters

- **p** (*int*) – The p value of the DH key.
- **g** (*int*) – The g value of the DH key.
- **q** (*int*) – The q value of the DH key.

Returns `True` if the given values of `p`, `g` and `q` are supported by this backend, otherwise `False`.

New in version 1.8.

dh_x942_serialization_supported()

Returns True if serialization of DH objects with subgroup order (q) is supported by this backend.

class cryptography.hazmat.backends.interfaces.**ScryptBackend**

New in version 1.6.

A backend with methods for using Scrypt.

The following backends implement this interface:

- *OpenSSL backend*

derive_scrypt(*self*, *key_material*, *salt*, *length*, *n*, *r*, *p*)

Parameters

- **key_material** (*bytes*) – The key material to use as a basis for the derived key. This is typically a password.
- **salt** (*bytes*) – A salt.
- **length** (*int*) – The desired length of the derived key.
- **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.

Return bytes 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.

2.7 Bindings

`cryptography` aims to provide low-level CFFI based bindings to multiple native C libraries. These provide no automatic initialization of the library and may not provide complete wrappers for its API.

Using these functions directly is likely to require you to be careful in managing memory allocation, locking and other resources.

2.7.1 Individual bindings

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.

OpenSSL binding

These are CFFI bindings to the OpenSSL C library. Cryptography supports OpenSSL version 1.0.1 and greater.

class `cryptography.hazmat.bindings.openssl.binding.`**Binding**

This is the exposed API for the OpenSSL bindings. It has two public attributes:

ffi

This is a `cffi.FFI` instance. It can be used to allocate and otherwise manipulate OpenSSL structures.

lib

This is a `cffi` library. It can be used to call OpenSSL functions, and access constants.

classmethod init_static_locks()

Enables the best available locking callback for OpenSSL. See *Threading*.

Threading

`cryptography` enables OpenSSLs thread safety facilities in several different ways depending on the configuration of your system. For users on OpenSSL 1.1.0 or newer (including anyone who uses a binary wheel) the OpenSSL internal locking callbacks are automatically used. Otherwise, we first attempt to use the callbacks provided by your Python implementation specifically for OpenSSL. This will work in every case except where `cryptography` is linked against a different version of OpenSSL than the one used by your Python implementation. For this final case we have a C-based locking callback.

2.8 Installation

You can install `cryptography` with `pip`:

```
$ pip install cryptography
```

2.8.1 Supported platforms

Currently we test `cryptography` on Python 2.7, 3.4+, and PyPy 5.4+ on these operating systems.

- x86-64 CentOS 7.x
- x86-64 Fedora (latest)
- macOS 10.13 High Sierra, 10.14 Mojave
- x86-64 Ubuntu 14.04, 16.04, and rolling
- x86-64 Debian Jessie (8.x), Stretch (9.x), Buster (10.x), and Sid (unstable)
- x86-64 Alpine (latest)
- 32-bit and 64-bit Python on 64-bit Windows Server 2019

We test compiling with `clang` as well as `gcc` and use the following OpenSSL releases:

- `OpenSSL 1.0.1`
- `OpenSSL 1.0.1e-fips` (`RHEL/CentOS 7`)
- `OpenSSL 1.0.1f`
- `OpenSSL 1.0.2-latest`
- `OpenSSL 1.1.0-latest`
- `OpenSSL 1.1.1-latest`

Warning: Cryptography 2.4 has deprecated support for OpenSSL 1.0.1.

2.8.2 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 the binaries we build for our own infrastructure. Be sure to download the proper version for your architecture and Python (2010 works for Python 2.7, 3.3, and 3.4 while 2015 is required for 3.5 and above). Wherever you place your copy of OpenSSL you'll need to set the `LIB` and `INCLUDE` environment variables to include the proper locations. For example:

```
C:\> \path\to\vcvarsall.bat x86_amd64
C:\> set LIB=C:\OpenSSL-win64\lib;%LIB%
C:\> set INCLUDE=C:\OpenSSL-win64\include;%INCLUDE%
C:\> pip install cryptography
```

As of OpenSSL 1.1.0 the library names have changed from `libeay32` and `ssleay32` to `libcrypto` and `libssl` (matching their names on all other platforms). `cryptography` links against the new 1.1.0 names by default. If you need to compile `cryptography` against an older version then you **must** set `CRYPTOGRAPHY_WINDOWS_LINK_LEGACY_OPENSSL` or else installation will fail.

If you need to rebuild `cryptography` for any reason be sure to clear the local wheel cache.

2.8.3 Building cryptography on Linux

`cryptography` ships a `manylinux1` wheel (as of 2.0) so all dependencies are included. For users on pip 8.1 or above running on a `manylinux1` compatible distribution (almost everything except Alpine) all you should need to do is:

```
$ pip install cryptography
```

If you are on Alpine or just want to compile it yourself then `cryptography` requires a compiler, headers for Python (if you're not using `pypy`), and headers for the OpenSSL and `libffi` libraries available on your system.

Alpine

Replace `python3-dev` with `python-dev` if you're using Python 2.

```
$ sudo apk add gcc musl-dev python3-dev libffi-dev openssl-dev
```

If you get an error with `openssl-dev` you may have to use `libressl-dev`.

Debian/Ubuntu

Replace `python3-dev` with `python-dev` if you're using Python 2.

```
$ sudo apt-get install build-essential libssl-dev libffi-dev python3-dev
```

RHEL/CentOS

```
$ sudo yum install redhat-rpm-config gcc libffi-devel python-devel \
    openssl-devel
```

Building

You should now be able to build and install cryptography. To avoid getting the pre-built wheel on `manylinux1` 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,--version-script=openssl.ld -Wl,-Bsymbolic-functions -fPIC shared
```

You'll also need to generate your own `openssl.ld` file. For example:

```
OPENSSL_1.1.0E_CUSTOM {
    global:
        *;
};
```

You should replace the version string on the first line as appropriate for your build.

Static Wheels

Cryptography ships statically-linked wheels for macOS, Windows, and Linux (via `manylinux1`). This allows compatible environments to use the most recent OpenSSL, regardless of what is shipped by default on those platforms. Some Linux distributions (most notably Alpine) are not `manylinux1` compatible so we cannot distribute wheels for them.

However, 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 1.0.2k, use `OPENSSL_VERSION="1.0.2k"`.

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

OPENSSL_VERSION="VERSIONGOESHERE"
CWD=$(pwd)

virtualenv env
. env/bin/activate
pip install -U setuptools
pip install -U wheel pip
curl -O https://www.openssl.org/source/openssl-${OPENSSL_VERSION}.tar.gz
tar xvf openssl-${OPENSSL_VERSION}.tar.gz
cd openssl-${OPENSSL_VERSION}
./config no-shared no-ssl2 no-ssl3 -fPIC --prefix=${CWD}/openssl
make && make install
cd ..
CFLAGS="-I${CWD}/openssl/include" LDFLAGS="-L${CWD}/openssl/lib" pip wheel --no-
↪binary :all: cryptography
```

2.8.4 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 10.9+) open a terminal window and run:

```
$ xcode-select --install
```

This will install a compiler (clang) along with (most of) the required development headers.

You'll also need OpenSSL, which you can obtain from Homebrew or MacPorts. Cryptography does **not** support Apple's deprecated OpenSSL distribution.

To build cryptography and dynamically link it:

Homebrew

```
$ brew install openssl@1.1
$ env LDFLAGS="-L$(brew --prefix openssl@1.1)/lib" CFLAGS="-I$(brew --prefix␣
↪openssl@1.1)/include" pip install cryptography
```

MacPorts:

```
$ sudo port install openssl
$ env LDFLAGS="-L/opt/local/lib" CFLAGS="-I/opt/local/include" pip install␣
↪cryptography
```

You can also build cryptography statically:

Homebrew

```
$ brew install openssl@1.1
$ env CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS=1 LDFLAGS="$(brew --prefix openssl@1.1)/lib/
↪libssl.a $(brew --prefix openssl@1.1)/lib/libcrypto.a" CFLAGS="-I$(brew --prefix␣
↪openssl@1.1)/include" pip install cryptography
```

MacPorts:

```
$ sudo port install openssl
$ env CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS=1 LDFLAGS="/opt/local/lib/libssl.a /opt/local/
↪lib/libcrypto.a" CFLAGS="-I/opt/local/include" pip install cryptography
```

If you need to rebuild `cryptography` for any reason be sure to clear the local wheel cache.

2.9 Changelog

2.9.1 2.7 - master

Note: This version is not yet released and is under active development.

- **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.
- 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`.

2.9.2 2.6.1 - 2019-02-27

- Resolved an error in our build infrastructure that broke our Python3 wheels for macOS and Linux.

2.9.3 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.

2.9.4 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 *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 *encode_point()*.

2.9.5 2.4.2 - 2018-11-21

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0j.

2.9.6 2.4.1 - 2018-11-11

- Fixed a build breakage in our `manylinux1` wheels.

2.9.7 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*.

2.9.8 2.3.1 - 2018-08-14

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0i.

2.9.9 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`.

2.9.10 2.2.2 - 2018-03-27

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0h.

2.9.11 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.

2.9.12 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`.

2.9.13 2.1.4 - 2017-11-29

- Added `X509_up_ref` for an upcoming `pyOpenSSL` release.

2.9.14 2.1.3 - 2017-11-02

- Updated Windows, macOS, and `manylinux1` wheels to be compiled with OpenSSL 1.1.0g.

2.9.15 2.1.2 - 2017-10-24

- Corrected a bug with the `manylinux1` wheels where OpenSSL's stack was marked executable.

2.9.16 2.1.1 - 2017-10-12

- Fixed support for install with the system `pip` on Ubuntu 16.04.

2.9.17 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.

2.9.18 2.0.3 - 2017-08-03

- Fixed an issue with weak linking symbols when compiling on macOS versions older than 10.12.

2.9.19 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.

2.9.20 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.

2.9.21 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.

2.9.22 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.

2.9.23 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.

2.9.24 1.8.1 - 2017-03-10

- Fixed macOS wheels to properly link against 1.1.0 rather than 1.0.2.

2.9.25 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 *`DHPrivateKeyWithSerialization`*.
- 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*.

2.9.26 1.7.2 - 2017-01-27

- Updated Windows and macOS wheels to be compiled against OpenSSL 1.0.2k.

2.9.27 1.7.1 - 2016-12-13

- Fixed a regression in `int_from_bytes` where it failed to accept `bytearray`.

2.9.28 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.

2.9.29 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.

2.9.30 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*

2.9.31 1.5.2 - 2016-09-26

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2j.

2.9.32 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.

2.9.33 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.

2.9.34 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`*.

2.9.35 1.3.4 - 2016-06-03

- Added another OpenSSL function to the bindings to support an upcoming `pyOpenSSL` release.

2.9.36 1.3.3 - 2016-06-02

- Added two new OpenSSL functions to the bindings to support an upcoming `pyOpenSSL` release.

2.9.37 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.

2.9.38 1.3.1 - 2016-03-21

- Fixed a bug that caused an `AttributeError` when using `mock` to patch some `cryptography` modules.

2.9.39 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()*.

2.9.40 1.2.3 - 2016-03-01

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2g.

2.9.41 1.2.2 - 2016-01-29

- Updated Windows and OS X wheels to be compiled against OpenSSL 1.0.2f.

2.9.42 1.2.1 - 2016-01-08

- Reverts a change to an OpenSSL `EVP_PKEY` object that caused errors with `pyOpenSSL`.

2.9.43 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.

2.9.44 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.

2.9.45 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.

2.9.46 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 *encode_point()* and *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.

2.9.47 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.

2.9.48 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+.

2.9.49 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()*.

2.9.50 0.9.3 - 2015-07-09

- Updated Windows wheels to be compiled against OpenSSL 1.0.2d.

2.9.51 0.9.2 - 2015-07-04

- Updated Windows wheels to be compiled against OpenSSL 1.0.2c.

2.9.52 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.

2.9.53 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 *RSAPublicKeyWithSerialization*, *DSAPublicKeyWithSerialization*, and *EllipticCurvePublicKeyWithSerialization*.
- Support *DER* serialization of private keys using the `private_bytes` method of *RSAPrivateKeyWithSerialization*, *DSAPrivateKeyWithSerialization*, and *EllipticCurvePrivateKeyWithSerialization*.

- 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.

2.9.54 0.8.2 - 2015-04-10

- Fixed a race condition when initializing the OpenSSL or CommonCrypto backends in a multi-threaded scenario.

2.9.55 0.8.1 - 2015-03-20

- Updated Windows wheels to be compiled against OpenSSL 1.0.2a.

2.9.56 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 *EllipticCurvePrivateKeyWithSerialization*.
- Added *RSAPrivateKeyWithSerialization* and deprecated RSAPrivateKeyWithNumbers.
- Added *private_bytes()* to *RSAPrivateKeyWithSerialization*.
- Added *DSAPrivateKeyWithSerialization* and deprecated DSAPrivateKeyWithNumbers.
- Added *private_bytes()* to *DSAPrivateKeyWithSerialization*.
- Added *RSAPublicKeyWithSerialization* and deprecated RSAPublicKeyWithNumbers.
- Added public_bytes to *RSAPublicKeyWithSerialization*.
- Added *EllipticCurvePublicKeyWithSerialization* and deprecated EllipticCurvePublicKeyWithNumbers.
- Added public_bytes to *EllipticCurvePublicKeyWithSerialization*.
- Added *DSAPublicKeyWithSerialization* and deprecated DSAPublicKeyWithNumbers.
- Added public_bytes to *DSAPublicKeyWithSerialization*.
- *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*.

2.9.57 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.

2.9.58 0.7.1 - 2014-12-28

- Fixed an issue preventing compilation on platforms where `OPENSSL_NO_SSL3` was defined.

2.9.59 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.

2.9.60 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.

2.9.61 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_numbe` 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.

2.9.62 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.

2.9.63 0.5.3 - 2014-08-06

- Updated Windows wheels to be compiled against OpenSSL 1.0.1i.

2.9.64 0.5.2 - 2014-07-09

- Add `TraditionalOpenSSLSerializationBackend` support to `multibackend`.
- Fix compilation error on OS X 10.8 (Mountain Lion).

2.9.65 0.5.1 - 2014-07-07

- Add `PKCS8SerializationBackend` support to `multibackend`.

2.9.66 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 backend*.
- Added `AES` *CTR* support to the OpenSSL backend when linked against 0.9.8.
- Added `PKCS8SerializationBackend` and `TraditionalOpenSSLSerializationBackend` support to the *OpenSSL backend*.
- Added *Elliptic curve cryptography* and *EllipticCurveBackend*.
- Added *ECB* support for *TripleDES* on `commoncrypto` and *OpenSSL backend*.
- 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*.

2.9.67 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*.

2.9.68 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.

2.9.69 0.2.2 - 2014-03-03

- Removed a constant definition that was causing compilation problems with specific versions of OpenSSL.

2.9.70 0.2.1 - 2014-02-22

- Fix a bug where importing cryptography from multiple paths could cause initialization to fail.

2.9.71 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 the *OpenSSL backend* to the OS random engine.
- Added *`CAST5`* (CAST-128) support.

2.9.72 0.1 - 2014-01-08

- Initial release.

2.10 Frequently asked questions

2.10.1 `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.

2.10.2 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.

2.10.3 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*.

2.10.4 Compiling `cryptography` on macOS produces a `fatal error: 'openssl/aes.h' file not found` error

This happens because macOS 10.11 no longer includes a copy of OpenSSL. `cryptography` now provides wheels which include a statically linked copy of OpenSSL. You're seeing this error because your copy of pip is too old to find our wheel files. Upgrade your copy of pip with `pip install -U pip` and then try install `cryptography` again.

If you are using PyPy, we do not currently ship `cryptography` wheels for PyPy. You will need to install your own copy of OpenSSL – we recommend using Homebrew.

2.10.5 `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.

2.10.6 error: `-Werror=sign-conversion`: No option `-Wsign-conversion` during installation

The compiler you are using is too old and not supported by `cryptography`. Please upgrade to a more recent version. If you are running OpenBSD 6.1 or earlier the default compiler is extremely old. Use `pkg_add` to install a newer `gcc` and then install `cryptography` using `CC=/path/to/newer/gcc pip install cryptography`.

2.10.7 Installing `cryptography` fails with `Invalid environment marker: python_version < '3'`

Your `pip` and/or `setuptools` are outdated. Please upgrade to the latest versions with `pip install -U pip setuptools` (or on Windows `python -m pip install -U pip setuptools`).

2.10.8 Installing cryptography with OpenSSL 0.9.8 or 1.0.0 fails

The OpenSSL project has dropped support for the 0.9.8 and 1.0.0 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 a newer version of OpenSSL (1.0.2 or later). This may require you to upgrade to a newer operating system.

2.10.9 Why are there no wheels for Python 3.5+ on Linux or macOS?

Our Python3 wheels, for macOS and Linux, are `abi3` wheels. This means they support multiple versions of Python. The Python 3.4 `abi3` wheel can be used with any version of Python greater than or equal to 3.4. Recent versions of `pip` will automatically install `abi3` wheels.

2.10.10 `ImportError: idna` is not installed

`cryptography` deprecated passing *U-label* strings to various X.509 constructors in version 2.1 and in version 2.5 moved the `idna` dependency to a `setuptools` extra. If you see this exception you should upgrade your software so that it no longer depends on this deprecated feature. If that is not yet possible you can also install `cryptography` with `pip install cryptography[idna]` to automatically install the missing dependency. This workaround will be available until the feature is fully removed.

2.10.11 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 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.

2.11.1 Getting started

Development dependencies

Working on `cryptography` requires the installation of a small number of development dependencies in addition to the dependencies for *Installation*. These are listed in `dev-requirements.txt` and they can be installed in a virtualenv using pip. Before you install them, follow the **build** instructions in *Installation* (be sure to stop before actually installing `cryptography`). Once you've done that, install the development dependencies, and then install `cryptography` in `editable` mode. For example:

```
$ # Create a virtualenv and activate it
$ # Set up your cryptography build environment
$ pip install --requirement dev-requirements.txt
$ pip install --editable .
```

Make sure that `pip install --requirement ...` has installed the Python package `vectors/` and packages on `tests/` . If it didn't, you may install them manually by using `pip` on each directory.

You will also need to install `enchant` using your system's package manager to check spelling in the documentation.

Note: There is an upstream bug in `enchant` that prevents its installation on Windows with 64-bit Python. See this Github issue for more information. The easiest workaround is to use 32-bit Python for `cryptography` development, even on 64-bit Windows.

You are now ready to run the tests and build the documentation.

OpenSSL on macOS

You must have installed OpenSSL via Homebrew or MacPorts and must set `CFLAGS` and `LDFLAGS` environment variables before installing the `dev-requirements.txt` otherwise pip will fail with include errors.

For example, with Homebrew:

```
$ env LDFLAGS="-L$(brew --prefix openssl@1.1)/lib" \
    CFLAGS="-I$(brew --prefix openssl@1.1)/include" \
    pip install --requirement ./dev-requirements.txt
```

Alternatively for a static build you can specify `CRYPTOGRAPHY_SUPPRESS_LINK_FLAGS=1` and ensure `LDFLAGS` points to the absolute path for the OpenSSL libraries before calling pip.

Tip: You will also need to set these values when *Building documentation*.

Running tests

`cryptography` unit tests are found in the `tests/` directory and are designed to be run using pytest. pytest will discover the tests automatically, so all you have to do is:

```
$ pytest
...
62746 passed in 220.43 seconds
```

This runs the tests with the default Python interpreter.

You can also verify that the tests pass on other supported Python interpreters. For this we use tox, which will automatically create a virtualenv for each supported Python version and run the tests. For example:

```
$ tox
...
 py27: commands succeeded
ERROR:   pypy: InterpreterNotFound: pypy
 py34: commands succeeded
```

(continues on next page)

(continued from previous page)

```
docs: commands succeeded
pep8: commands succeeded
```

You may not have all the required Python versions installed, in which case you will see one or more `InterpreterNotFound` errors.

Building documentation

`cryptography` documentation is stored in the `docs/` directory. It is written in reStructured Text and rendered using Sphinx.

Use tox to build the documentation. For example:

```
$ tox -e docs
...
docs: commands succeeded
congratulations :)
```

The HTML documentation index can now be found at `docs/_build/html/index.html`.

2.11.2 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 running `flake8` against it. If you've installed the development requirements this will automatically use our configuration. You can also run the `tox` job with `tox -e pep8`.

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.
```

Additionally, every Python code file must contain

```
from __future__ import absolute_import, division, print_function
```

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):
    # ...
    return is_valid

# Good!
def verify(sig):
    # ...
    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.

APIs at the *Primitives* layer should always take an explicit backend, APIs at the recipes layer should automatically use the *`default_backend()`*, but optionally allow specifying a different backend.

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 and passes doc8 style checks you can run `tox -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.

2.11.3 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 a 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 `master`, 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 `master`.
- If somehow the tests get into a failing state on `master` (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.

2.11.4 Test vectors

Testing the correctness of the primitives implemented in each `cryptography` backend 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 to verify consistency between implemented backends.

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.

We have ensured all test vectors are used as of commit `c313761979d74b0417230eddd0f87d0cfab2b46b`.

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_*.
- OpenSSL PEM RSA serialization vectors from the OpenSSL example key and GnuTLS key parsing tests.
- 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**.

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.

```
from __future__ import absolute_import, print_function

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

from cryptography_vectors import open_vector_file
```

(continues on next page)

(continued from previous page)

```
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(object):
    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 "[K-256,{0}]".format(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 "Msg = {0}".format(hexlify(message))
            yield "d = {0:x}".format(secret_key.privkey.secret_multiplier)
            yield "Qx = {0:x}".format(public_key.pubkey.point.x())
            yield "Qy = {0:x}".format(public_key.pubkey.point.y())
            yield "R = {0:x}".format(r)
            yield "S = {0:x}".format(s)
            yield ""
```

(continues on next page)

(continued from previous page)

```
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.

```
from __future__ import absolute_import, print_function

import os

from cryptography.hazmat.backends import default_backend
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']
```

(continues on next page)

(continued from previous page)

```
    message = vector['message']
    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(default_backend())

    verifier = key.verifier(
        signature,
        ec.ECDSA(CRYPTOGRAPHY_HASH_TYPES[digest_algorithm]())
    )
    verifier.update(message)
    return verifier.verify()


def verify_vectors(vectors):
    for vector in vectors:
        assert 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.

from __future__ import absolute_import, division, print_function

import binascii
```

(continues on next page)

(continued from previous page)

```
import itertools
import os

from cryptography.hazmat.backends.openssl.backend import backend
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


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, backend)
        pn = skey.private_numbers()
        examples = private["examples"]
        output.append(b"# =============================================")
        output.append(b"# Example")
        output.append(b"# Public key")
        output.append(b"# Modulus:")
        output.append(format(pn.public_numbers.n, "x"))
        output.append(b"# Exponent:")
        output.append(format(pn.public_numbers.e, "x"))
        output.append(b"# Private key")
        output.append(b"# Modulus:")
        output.append(format(pn.public_numbers.n, "x"))
        output.append(b"# Public exponent:")
        output.append(format(pn.public_numbers.e, "x"))
        output.append(b"# Exponent:")
        output.append(format(pn.d, "x"))
        output.append(b"# Prime 1:")
        output.append(format(pn.p, "x"))
        output.append(b"# Prime 2:")
        output.append(format(pn.q, "x"))
        output.append(b"# Prime exponent 1:")
        output.append(format(pn.dmp1, "x"))
        output.append(b"# Prime exponent 2:")
        output.append(format(pn.dmq1, "x"))
        output.append(b"# 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(
```

(continues on next page)

(continued from previous page)

```
                e=private["public_exponent"],
                n=private["modulus"]
            )
        ).private_key(backend)
        count = 1

        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(
                b"# OAEP Example {0} alg={1} mgf1={2}".format(
                    count, hashalg.name, mgf1alg.name
                )
            )
            count += 1
            output.append(b"# Message:")
            output.append(example["message"])
            output.append(b"# Encryption:")
            output.append(binascii.hexlify(ct))

    return b"\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)
```

(continues on next page)

(continued from previous page)

```
    ):
        continue

    write_file(
        build_vectors(hashtuple[0], hashtuple[1], oaep_path),
        "oaep-{0}-{1}.txt".format(hashtuple[0].name, hashtuple[1].name)
    )
```

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";
    private static final String PUBLIC_KEY = "# Public key";
```

(continues on next page)

(continued from previous page)

```
    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();
        }
    }

    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();
```

(continues on next page)

(continued from previous page)

```
        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;
    }

    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(); // #
```

(continues on next page)

(continued from previous page)

```
    }

    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);

        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);
```

(continues on next page)

(continued from previous page)

```
        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;
        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) {
```

(continues on next page)

(continued from previous page)

```
        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:
            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) {
```

(continues on next page)

(continued from previous page)

```
        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;
        }

        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.");
        }
    }
```

(continues on next page)

(continued from previous page)

```
    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 + "...");

                    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 8 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/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/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/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 as well as the OpenSSL example key for RSA.
- 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`. This key is encoded in several of the PKCS12 custom vectors.

- `asymmetric/EC/compressed_points.txt` - Contains compressed public points generated using OpenSSL.
- `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/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.

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/cryptoraphy_vectors/asymmetric/ECDH/brainpool.txt` contains Brainpool vectors from **RFC 7027**.

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. This is a PEM conversion of the root data in the NSS source tree.
- `cryptography.io.pem` - A leaf certificate issued by RapidSSL for the cryptography website.
- `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.io.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.pem` - A certificate from an unknown signature OID Mozilla bug that uses an alternate signature OID for RSA with SHA1.

- `badssl-sct.pem` - A certificate with the certificate transparency signed certificate timestamp extension.
- `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.

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!™".
- `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_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.
- `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.
- `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.
- `ca/ca.pem` - A self-signed certificate with `basicConstraints` set to true. Its private key is `ca/ca_key.pem`. This certificate 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.

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.
- `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.
- `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".
- `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".
- `basic_constraints.pem` - A certificate signing request for an RSA 2048 bit key containing a basic constraints extension marked as critical.
- `invalid_signature.pem` - A certificate signing request for an RSA 1024 bit key containing an invalid signature with correct padding.

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".

- `crl_dup_entry_ext.pem` - Contains a CRL with one revocation which has a duplicate entry extension.
- `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.
- `crl_unsupported_reason.pem` - Contains a CRL with one revocation which has an unsupported reason code.
- `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.
- `crl_empty.pem` - Contains a CRL with no revoked certificates.
- `crl_ian_aia_aki.pem` - Contains a CRL with `IssuerAlternativeName`, `AuthorityInformationAccess`, `AuthorityKeyIdentifier` and `CRLNumber` extensions.
- `valid_signature.pem` - Contains a CRL with the public key which was used to generate it.
- `invalid_signature.pem` - Contains a CRL with the last signature byte incremented by 1 to produce an invalid signature, and the public key which was used to generate it.
- `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.

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.

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.

Custom PKCS12 Test Vectors

- `pkcs12/cert-key-aes256cbc.p12` - A PKCS12 file containing a cert (`x509/custom/ca/ca.pem`) and key (`x509/custom/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 (`x509/custom/ca/ca.pem`) and key (`x509/custom/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 (`x509/custom/ca/ca.pem`) encrypted with RC2 and key (`x509/custom/ca/ca_key.pem`) encrypted via 3DES with the password `cryptography`.
- `pkcs12/no-password.p12` - A PKCS12 file containing a cert (`x509/custom/ca/ca.pem`) and key (`x509/custom/ca/ca_key.pem`) with no encryption and no password.
- `pkcs12/no-cert-key-aes256cbc.p12` - A PKCS12 file containing a key (`x509/custom/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 (`x509/custom/ca/ca.pem`) encrypted via AES 256 CBC with the password `cryptography` and no private key.

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.

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.

Symmetric ciphers

- AES (CBC, CFB, ECB, GCM, OFB, CCM) from NIST CAVP.
- AES CTR from **RFC 3686**.
- 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*
- SEED (ECB) from **RFC 4269**.
- SEED (CBC) from **RFC 4196**.
- SEED (CFB, OFB) generated by this project. See: *SEED vector creation*

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

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.

from __future__ import absolute_import, division, print_function

import binascii

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import ciphers
from cryptography.hazmat.primitives.ciphers import algorithms
```

(continues on next page)

(continued from previous page)

```
_RFC6229_KEY_MATERIALS = [
    (True,
     8 * '0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20'),
    (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,
                default_backend())
```

(continues on next page)

(continued from previous page)

```
            encryptor = cipher.encryptor()
            current_offset = 0
            for offset in _RFC6229_OFFSETS:
                if offset % 16 != 0:
                    raise ValueError(
                        "Offset {} is not evenly divisible by 16"
                        .format(offset))
                while current_offset < offset:
                    encryptor.update(plaintext)
                    current_offset += len(plaintext)
                output.append("\nCOUNT = {}".format(count))
                count += 1
                output.append("KEY = {}".format(key))
                output.append("OFFSET = {}".format(offset))
                output.append("PLAINTEXT = {}".format(
                    binascii.hexlify(plaintext)))
                output.append("CIPHERTEXT = {}".format(
                    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)
        if err != nil {
                panic(err)
        }
```

(continues on next page)

(continued from previous page)

```
        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)
        }
        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,
                                        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.

from __future__ import absolute_import, division, print_function

import binascii

from cryptography.hazmat.backends import default_backend
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)),
        default_backend()
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    vector_file = open(filename, "r")

    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("CIPHERTEXT = {}".format(
                    encrypt(mode, key, iv, plaintext))
                )
            output.append("\nCOUNT = {}".format(count))
            count += 1
            name, key = line.split(" = ")
            output.append("KEY = {}".format(key))
        elif line.startswith("IV"):
            name, iv = line.split(" = ")
            iv = iv[0:16]
            output.append("IV = {}".format(iv))
        elif line.startswith("PLAINTEXT"):
            name, plaintext = line.split(" = ")
            output.append("PLAINTEXT = {}".format(plaintext))

    output.append("CIPHERTEXT = {}".format(encrypt(mode, key, iv, plaintext)))
    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)


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
        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) {
```

(continues on next page)

(continued from previous page)

```
        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{}

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)))
        }
```

(continues on next page)

(continued from previous page)

```
}

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)
        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]
                }
        }
```

(continues on next page)

(continued from previous page)

```
}

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.backends.openssl.backend import backend
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)),
        backend
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    with open(filename, "r") as f:
        vector_file = f.read().splitlines()
```

(continues on next page)

(continued from previous page)

```
    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("CIPHERTEXT = {0}".format(
                    encrypt(mode, key, iv, plaintext))
                )
            output.append("\nCOUNT = {0}".format(count))
            count += 1
            name, key = line.split(" = ")
            output.append("KEY = {0}".format(key))
        elif line.startswith("IV"):
            name, iv = line.split(" = ")
            iv = iv[0:16]
            output.append("IV = {0}".format(iv))
        elif line.startswith("PLAINTEXT"):
            name, plaintext = line.split(" = ")
            output.append("PLAINTEXT = {0}".format(plaintext))

    output.append("CIPHERTEXT = {0}".format(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), "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
```

(continues on next page)

(continued from previous page)

```
def encrypt(mode, key, iv, plaintext):
    encryptor = botan.Cipher("IDEA/{0}/NoPadding".format(mode), "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, "r") 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"
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.backends.openssl.backend import backend
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)),
```

(continues on next page)

(continued from previous page)

```
        backend
    )
    encryptor = cipher.encryptor()
    ct = encryptor.update(binascii.unhexlify(plaintext))
    ct += encryptor.finalize()
    return binascii.hexlify(ct)


def build_vectors(mode, filename):
    with open(filename, "r") 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("CIPHERTEXT = {0}".format(
                    encrypt(mode, key, iv, plaintext))
                )
            output.append("\nCOUNT = {0}".format(count))
            count += 1
            name, key = line.split(" = ")
            output.append("KEY = {0}".format(key))
        elif line.startswith("IV"):
            name, iv = line.split(" = ")
            output.append("IV = {0}".format(iv))
        elif line.startswith("PLAINTEXT"):
            name, plaintext = line.split(" = ")
            output.append("PLAINTEXT = {0}".format(plaintext))

    output.append("CIPHERTEXT = {0}".format(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("SEED/{0}/NoPadding".format(mode), "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, "r") 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"]


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.

from __future__ import absolute_import, division, print_function

import binascii
```

(continues on next page)

(continued from previous page)

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

IKM = binascii.unhexlify(b"0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b")
L = 1200
OKM = HKDF(
    algorithm=hashes.SHA256(), length=L, salt=None, info=None,
    backend=default_backend()
).derive(IKM)


def _build_vectors():
    output = []
    output.append("COUNT = 0")
    output.append("Hash = SHA-256")
    output.append("IKM = " + binascii.hexlify(IKM).decode("ascii"))
    output.append("salt = ")
    output.append("info = ")
    output.append("L = {}".format(L))
    output.append("OKM = " + binascii.hexlify(OKM).decode("ascii"))
    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(), '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 {
```

(continues on next page)

(continued from previous page)

```
        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

        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`

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
```

2.11.5 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 1.1.0 or newer:

```
#if CRYPTOGRAPHY_OPENSSL_110_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.

2.12 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.

2.12.1 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.

2.12.2 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.

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.

2.12.3 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 to `alex.gaynor@gmail.com`. Messages may be optionally encrypted with PGP using key fingerprint `F7FC 698F AAE2 D2EF BECD E98E D1B3 ADC0 E023 8CA6` (this public key is available from most commonly-used key servers).

Once you've submitted an issue via email, you should receive an acknowledgment within 48 hours, and depending on the action to be taken, you may receive further follow-up emails.

2.12.4 Supported Versions

At any given time, we will provide security support for the master branch as well as the most recent release.

2.12.5 New releases for OpenSSL updates

As of versions 0.5, 1.0.1, and 2.0.0, `cryptography` statically links OpenSSL on 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.

2.12.6 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.

2.13 Known security limitations

2.13.1 Lack of secure memory wiping

Memory wiping is used to protect secret data or key material from attackers with access to uninitialized memory. This can be either because the attacker has some kind of local user access or because of how other software uses uninitialized memory.

Python exposes no API for us to implement this reliably and as such 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.

2.14 API stability

From its first release, `cryptography` will have a strong API stability policy.

2.14.1 What does this policy cover?

This policy includes any API or behavior that is documented in this documentation.

2.14.2 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.

2.14.3 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.
- 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.

2.14.4 Versioning

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.

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.Y` the feature exists.
- In `cryptography X.Y + 0.1` using that feature will emit a `UserWarning`.
- In `cryptography X.Y + 0.2` using that feature will emit a `UserWarning`.
- In `cryptography X.Y + 0.3` the feature will be removed or changed.

In short, code that runs without warnings will always continue to work for a period of two 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.

2.15 Doing a release

Doing a release of `cryptography` requires a few steps.

2.15.1 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.
- Ensure that the *Changelog* entry credits whoever reported the issue.
- The release should be announced on the oss-security mailing list, in addition to the regular announcement lists.

2.15.2 Verifying OpenSSL version

The release process creates wheels bundling OpenSSL for Windows, macOS, and Linux. Check that the Windows and macOS Azure Pipelines builders have the latest version of OpenSSL installed and verify that the latest version is present in the `pyca/cryptography-manylinux1` docker containers. If anything is out of date follow the instructions for upgrading OpenSSL.

2.15.3 Upgrading OpenSSL

Use the upgrading OpenSSL issue template.

2.15.4 Bumping the version number

The next step in doing a release is bumping the version number in the software.

- Update the version number in `src/cryptography/__about__.py`.
- Update the version number in `vectors/cryptography_vectors/__about__.py`.
- Set the release date in the *Changelog*.
- Do a commit indicating this.
- Send a pull request with this.
- Wait for it to be merged.

2.15.5 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 `gpg` installed and a `gpg` key in order to do a release. Once this has happened:

- Run `python release.py {version}`.

The release should now be available on PyPI and a tag should be available in the repository.

2.15.6 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.

2.15.7 Post-release tasks

- Update the version number to the next major (e.g. `0.5.dev1`) in `src/cryptography/__about__.py` and `vectors/cryptography_vectors/__about__.py`.
- Close the milestone for the previous release on GitHub.
- 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.
- Send an email to the mailing list and python-announce announcing the release.

2.16 Community

You can find `cryptography` all over the web:

- Mailing list
- Source code
- Issue tracker
- Documentation
- IRC: `#cryptography-dev` on `irc.freenode.net`

Wherever we interact, we adhere to the Python Community Code of Conduct.

2.17 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.

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.

text This type corresponds to `unicode` on Python 2 and `str` on Python 3. This is equivalent to `six.text_type`.

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.

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

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y