PyCryptodome Documentation

Release 3.4.7

Legrandin

Sep 04, 2017

Contents

CHAPTER 1

PyCryptodome

PyCryptodome is a self-contained Python package of low-level cryptographic primitives.

It supports Python 2.4 or newer, all Python 3 versions and PyPy.

The installation procedure depends on the package you want the library in. PyCryptodome can be used as:

1. **a drop-in replacement for the old PyCrypto library**. You install it with:

   ```
   pip install pycryptodome
   ```

   In this case, all modules are installed under the `Crypto` package.

   One must avoid having both PyCrypto and PyCryptodome installed at the same time, as they will interfere with each other.

   This option is therefore recommended only when you are sure that the whole application is deployed in a `virtualenv`.

2. **a library independent of the old PyCrypto**. You install it with:

   ```
   pip install pycryptodomex
   ```

   In this case, all modules are installed under the `Cryptodome` package. PyCrypto and PyCryptodome can coexist.

For faster public key operations, you should have GMP installed in your system (except on Windows, as the wheel on PyPi already comes bundled with the equivalent MPIR library).

PyCryptodome is a fork of PyCrypto. It brings the following enhancements with respect to the last official version of PyCrypto (2.6.1):

- Authenticated encryption modes (GCM, CCM, EAX, SIV, OCB)
- Accelerated AES on Intel platforms via AES-NI
- First class support for PyPy
- Elliptic curves cryptography (NIST P-256 curve only)

- Better and more compact API (*nonce* and *iv* attributes for ciphers, automatic generation of random nonces and IVs, simplified CTR cipher mode, and more)
- SHA-3 (including SHAKE XOFs) and BLAKE2 hash algorithms
- Salsa20 and ChaCha20 stream ciphers
- scrypt and HKDF
- Deterministic (EC)DSA
- Password-protected PKCS#8 key containers
- Shamir's Secret Sharing scheme
- Random numbers get sourced directly from the OS (and not from a CSPRNG in userspace)
- Simplified install process, including better support for Windows
- Cleaner RSA and DSA key generation (largely based on FIPS 186-4)
- Major clean ups and simplification of the code base

PyCryptodome is not a wrapper to a separate C library like *OpenSSL*. To the largest possible extent, algorithms are implemented in pure Python. Only the pieces that are extremely critical to performance (e.g. block ciphers) are implemented as C extensions.

For more information, see the homepage.

All the code can be downloaded from GitHub.

News

- **26 Aug 2017 (NEW)**. Bugfix release 3.4.7.
- 17 May 2017. Bugfix release 3.4.6.
- 6 Feb 2017. Bugfix release 3.4.5.
- 1 Feb 2017. Bugfix release 3.4.4.
- 17 Oct 2016. Bugfix release 3.4.3.
- 8 Mar 2016. Bugfix release 3.4.2.
- 21 Feb 2016. Release 3.4.1.
- 7 Feb 2016. Release 3.4.
- Nov 2015. Release 3.3.1.
- 29 Oct 2015. Release 3.3.
- 9 Sep 2015. Minor release 3.2.1.
- 6 Sep 2015. Release 3.2.
- 15 Mar 2015. Release 3.1.
- 24 Jun 2014. Release 3.0.

CHAPTER 2

Features

This page lists the low-level primitives that PyCryptodome provides.

You are expected to have a solid understanding of cryptography and security engineering to successfully use them.

You must also be able to recognize that some primitives are obsolete (e.g. TDES) or even unsecure (RC4). They are provided only to enable backward compatibility where required by the applications.

A list of useful resources in that area can be found on Matthew Green's blog.

- Symmetric ciphers:
  - AES
  - Single and Triple DES
  - CAST-128
  - RC2
- Traditional modes of operations for symmetric ciphers:
  - ECB
  - CBC
  - CFB
  - OFB
  - CTR
  - OpenPGP (a variant of CFB, RFC4880)
- AEAD modes of operations for symmetric ciphers:
  - CCM (AES only)
  - EAX
  - GCM (AES only)
  - SIV (AES only)

  - OCB (AES only)
- Stream ciphers:
  - Salsa20
  - ChaCha20
  - RC4
- Cryptographic hashes:
  - SHA-1
  - SHA-2 hashes (224, 256, 384, 512)
  - SHA-3 hashes (224, 256, 384, 512) and XOFs (SHAKE128, SHAKE256)
  - Keccak (original submission to SHA-3)
  - BLAKE2b and BLAKE2s
  - RIPE-MD160
  - MD5
- Message Authentication Codes (MAC):
  - HMAC
  - CMAC
- Asymmetric key generation:
  - RSA
  - DSA
  - ECC (NIST P-256 curve only)
  - ElGamal
- Export and import format for asymmetric keys:
  - PEM (clear and encrypted)
  - PKCS#8 (clear and encrypted)
  - ASN.1 DER
- Asymmetric ciphers:
  - PKCS#1 (RSA)
    - RSAES-PKCS1-v1_5
    - RSAES-OAEP
- Asymmetric digital signatures:
  - PKCS#1 (RSA)
    - RSASSA-PKCS1-v1_5
    - RSASSA-PSS
  - (EC)DSA
    - Nonce-based (FIPS 186-3)
    - Deterministic (RFC6979)

- Key derivation:
  - PBKDF1
  - PBKDF2
  - scrypt
  - HKDF
- Other cryptographic protocols:
  - Shamir Secret Sharing
  - Padding
    - PKCS#7
    - ISO-7816
    - X.923

CHAPTER 3

Installation

The installation procedure depends on the package you want the library in. PyCryptodome can be used as:

1. **a drop-in replacement for the old PyCrypto library**. You install it with:

   ```
   pip install pycryptodome
   ```

   In this case, all modules are installed under the `Crypto` package. You can test everything is right with:

   ```
   python -m Crypto.SelfTest
   ```

   One must avoid having both PyCrypto and PyCryptodome installed at the same time, as they will interfere with each other.

   This option is therefore recommended only when you are sure that the whole application is deployed in a `virtualenv`.

2. **a library independent of the old PyCrypto**. You install it with:

   ```
   pip install pycryptodomex
   ```

   You can test everything is right with:

   ```
   python -m Cryptodome.SelfTest
   ```

   In this case, all modules are installed under the `Cryptodome` package. PyCrypto and PyCryptodome can coexist.

The procedures below go a bit more in detail, by explaining how to setup the environment for compiling the C extensions for each OS, and how to install the GMP library.

All instructions to follow install PyCryptodome as the `Cryptodome` package (option #2). Change `pycryptodomex` to `pycryptodome` if you prefer option #1 (`Crypto` package).

Linux Ubuntu

For Python 2.x:

```
$ sudo apt-get install build-essential libgmp3-dev python-dev
$ pip install pycryptodomex
$ python -m Cryptodome.SelfTest
```

For Python 3.x:

```
$ sudo apt-get install build-essential libgmp3-dev python3-dev
$ pip install pycryptodomex
$ python3 -m Cryptodome.SelfTest
```

For PyPy:

```
$ sudo apt-get install build-essential libgmp3-dev pypy-dev
$ pip install pycryptodomex
$ pypy -m Cryptodome.SelfTest
```

Linux Fedora

For Python 2.x:

```
$ sudo yum install gcc gmp python-devel
$ pip install pycryptodomex
$ python -m Cryptodome.SelfTest
```

For Python 3.x:

```
$ sudo yum install gcc gmp python3-devel
$ pip install pycryptodomex
$ python3 -m Cryptodome.SelfTest
```

For PyPy:

```
$ sudo yum install gcc gmp pypy-devel
$ pip install pycryptodomex
$ pypy -m Cryptodome.SelfTest
```

Windows (pre-compiled)

1. Install PyCryptodome as a wheel:

   ```
   > pip install pycryptodomex
   ```

2. To make sure everything works fine, run the test suite:

   ```
   > python -m Cryptodome.SelfTest
   ```

Windows (from sources, Python 2.x, Python <=3.2)

Windows does not come with a C compiler like most Unix systems. The simplest way to compile the *Pycryptodome* extensions from source code is to install the minimum set of Visual Studio components freely made available by Microsoft.

1. Run Python from the command line and note down its version and whether it is a 32 bit or a 64 bit application.

   For instance, if you see:

   ```
   Python 2.7.2+ ... [MSC v.1500 32 bit (Intel)] on win32
   ```

   you clearly have Python 2.7 and it is a 32 bit application.

2. **[Only once]** In order to speed up asymmetric key algorithms like RSA, it is recommended to install the MPIR library (a fork of the popular GMP library, more suitable for the Windows environment). For convenience, I made available pre-compiled *mpir.dll* files to match the various types of Python one may have:

   - Python 2.x, 3.1, 3.2 (VS2008 runtime)
     - 32 bits
     - 64 bits
   - Python 3.3 and 3.4 (VS2010 runtime)
     - 32 bits
     - 64 bits
   - Python 3.5 (VS2015 runtime)
     - 32 bits
     - 64 bits

   Download the correct *mpir.dll* and drop it into the Python interpreter directory (for instance `C:\Python34`). *Pycryptodome* will automatically make use of it.

3. **[Only once]** Install Virtual Clone Drive.

4. **[Only once]** Download the ISO image of the MS SDK for Windows 7 and . NET Framework 3.5 SP1. It contains the Visual C++ 2008 compiler.

   There are three ISO images available: you will need `GRMSDK_EN_DVD.iso` if your Windows OS is 32 bits or `GRMSDKX_EN_DVD.iso` if 64 bits.

   Mount the ISO with *Virtual Clone Drive* and install the C/C++ compilers and the redistributable only.

5. If your Python is a 64 bit application, open a command prompt and perform the following steps:

   ```
   > cd "C:\Program Files\Microsoft SDKs\Windows\v7.0"
   > cmd /V:ON /K Bin\SetEnv.Cmd /x64 /release
   > set DISTUTILS_USE_SDK=1
   ```

   Replace `/x64` with `/x86` if your Python is a 32 bit application.

6. Compile and install PyCryptodome:

   ```
   > pip install pycryptodomex --no-use-wheel
   ```

7. To make sure everything work fine, run the test suite:

```
> python -m Cryptodome.SelfTest
```

Windows (from sources, Python 3.3 and 3.4)

Windows does not come with a C compiler like most Unix systems. The simplest way to compile the *Pycryptodome* extensions from source code is to install the minimum set of Visual Studio components freely made available by Microsoft.

1. Run Python from the command line and note down its version and whether it is a 32 bit or a 64 bit application.

   For instance, if you see:

   ```
   Python 2.7.2+ ... [MSC v.1500 32 bit (Intel)] on win32
   ```

   you clearly have Python 2.7 and it is a 32 bit application.

2. **[Only once]** In order to speed up asymmetric key algorithms like RSA, it is recommended to install the MPIR library (a fork of the popular GMP library, more suitable for the Windows environment). For convenience, I made available pre-compiled *mpir.dll* files to match the various types of Python one may have:

   - Python 2.x, 3.1, 3.2 (VS2008 runtime)
     - 32 bits
     - 64 bits
   - Python 3.3 and 3.4 (VS2010 runtime)
     - 32 bits
     - 64 bits
   - Python 3.5 (VS2015 runtime)
     - 32 bits
     - 64 bits

   Download the correct *mpir.dll* and drop it into the Python interpreter directory (for instance `C:\Python34`). *Pycryptodome* will automatically make use of it.

3. **[Only once]** Install Virtual Clone Drive.

4. **[Only once]** Download the ISO image of the MS SDK for Windows 7 and . NET Framework 4. It contains the Visual C++ 2010 compiler.

   There are three ISO images available: you will need `GRMSDK_EN_DVD.iso` if your Windows OS is 32 bits or `GRMSDKX_EN_DVD.iso` if 64 bits.

   Mount the ISO with *Virtual Clone Drive* and install the C/C++ compilers and the redistributable only.

5. If your Python is a 64 bit application, open a command prompt and perform the following steps:

   ```
   > cd "C:\Program Files\Microsoft SDKs\Windows\v7.1"
   > cmd /V:ON /K Bin\SetEnv.Cmd /x64 /release
   > set DISTUTILS_USE_SDK=1
   ```

   Replace `/x64` with `/x86` if your Python is a 32 bit application.

6. Compile and install PyCryptodome:

```
> pip install pycryptodomex --no-use-wheel
```

7. To make sure everything work fine, run the test suite:

```
> python -m Cryptodome.SelfTest
```

Windows (from sources, Python 3.5 and newer)

Windows does not come with a C compiler like most Unix systems. The simplest way to compile the *Pycryptodome* extensions from source code is to install the minimum set of Visual Studio components freely made available by Microsoft.

1. **[Once only]** Download MS Visual Studio 2015 (Community Edition) and install the C/C++ compilers and the redistributable only.
2. Perform all steps from the section *Windows (pre-compiled)* but add the `--no-use-wheel` parameter when calling `pip`:

```
> pip install pycryptodomex --no-use-wheel
```

Documentation

Project documentation is written in reStructuredText and it is stored under `Doc/src`. To publish it as HTML files, you need to install *sphinx <http://www.sphinx-doc.org/en/stable/>* and use:

```
> make -C Doc/ html
```

It will then be available under `Doc/_build/html/`.

PGP verification

All source packages and wheels on PyPI are cryptographically signed. They can be verified with the following PGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFTXjPgBEADc3j7vnma9MXRshBPPXXenVpthQD6lrF/3XaBT2RptSf/viOD+
tz85du5XVp+r0SYYGeMNJCQ9NsztxblN/lnKgkfWRmSrB+V6QGS+e3bR5d9OIxzN
7haPxBnyRj//hCT/kKis6fa7N9wtwKBBjbaSX+9vpt7Rrt203sKfcChA4iR3EG89
TNQoc/kGGmwk/gyjfU38726v0NOhMKJp2154iQQVZ76hTDk6GkOYHTcPxdkAj4jS
Dd74M9sOtoOlyDLHOLcWNnlWGgZjtz0z0qSyFXRSuOfggTxrepWQgKWXXzgVB4Jo
0bhmXPAV8vkX5BoG6zGkYb47NGGvknax6jCvFYTCp1sOmVtf5UTVKPplFm077tQg
0KZNAvEQrdWRIiQ1cCGCoF2Alex3VmVdefHOhNmyY7xAlzpP0c8z1DsgZgMnytNn
GPusWeqQVijRxenl+lyhbkb9ZLDq7mOkCRXSze9J2+5aLTJbJu3+Wx6BEyNIHP/f
K3E77nXvC0oKaYTbTwEQSBAggAXP+7oQaA0ea2SLO176xJdNfC5lkQEtMMSZI4gN
iSqjUxXW2N5qEHHex1atmTtk4W9tQEw030a0UCxzDJMhD0aWFKq7wOxoCQ1q821R
vxBH4cfGWdL/1FUcuCMSUlc6fhTM9pvMXgjdEXcoiLSTdaHuVLuqmF/E0wARAQAB
tB9MZWdyYW5kaW4gPGhlbGRlcmlqc0BnbWFpbC5jb20+iQI4BBMBAgAiBQJU14z4
AhsDBgsJCAcDAgYVCAIJCgsEFgIDAQIeAQIXgAAKCRDabO+N4RaZEn7IEACpApha
vRwPB+Dv87aEyVmjZ96Nb3mxHdeP2uSmUxAODzoB5oJJ1QL6HRxEVlU8idjdf73H
DX39ZC7izD+oYIve9sNwTbKqJCZaTxlTDdgSF1N57eJOlELAy+SqpHtaMJPk7SfJ
```

```
l/iYoUYxByPLZU1wDwZEDNzt9RCGy3bd/vF/AxWjdUJJPh3E4j5hswvIGSf8/Tp3
MDROU1BaNBOd0CLvBHok8/xavwO6Dk/fE4hJhd5uZcEPtd1GJcPq51z2yr7PGUcb
oERsKZyG8cgfd7j8qoTd6jMIW6fBVHdxiMxW6/Z45X/vVciQSzzEl/yjPUW42kyr
Ib6M16YmnDzp8bl4NNFvvR9uWvOdUkep2Bi8s8kBMJ7G9rHHJcdVy/tP1ECS9Bse
hN4v5oJJ4v5mM/MiWRGKykZULWklonpiq6CewYkmXQDMRnjGXhjCWrB6LuSIkIXd
gKvDNpJ8yEhAfmpvA4I3laMoof/tSZ7ZuyLSZGLK16hoNIB13HCn4dnjNBeaXCWX
pThgeOWxV6u1fhz4CeC1Hc8WOYr8S7G8P10Ji6owOcj/a1QuCW8XDB2omCTXlhFj
zpC9dX8HgmUVnbPNiMjphihbKXoOcunRx4ZvqIa8mnTbI4tHtR0K0tI4MmbpcVOZ
8IFJ0nZJXuZiL57ijLREisPYmHfBHAgmh1j/W7kCDQRU14z4ARAA3QATRgvOSYFh
nJOnIz6PO3G9kXWjJ8wvp3yE1/PwwTc3NbVUSNCW14xgM2Ryhn9NVh8iEGtPGmUP
4vu7rvuLC2rBs1joBTyqf0mDghlZrb5ZjXv5LcG9SA6FdAXRU6T+b1G2ychKkhEh
d/ulLw/TKLds9zHhE+hkAagLQ5jqjcQN0iX5EYaOukiPUGmnd9fOEGi9YMYtRdrH
+3bZxUpsRStLBWJ6auY7Bla8NJOhaWpr5p/ls+mnDWoqf+tXCCps1Da/pfHKYDFc
2VVdyM/VfNny9eaczYpnj5hvIAACWChgGDBwxPh2DGdUfiQi/QqrK96+F7ulqz6V
2exX4CL0cPv5fUpQqSU/0R5WApM9bl2+wljFhoCXlydU9HNn+0GatGzEoo3yrV/m
PXv7d6NdZxyOqgxu/ai/z++F2pWUXSBxZN3Gv28boFKQhmtthTcFudNUtQOchhn8
Pf/ipVISqrsZorTx9Qx4fPScEWjwbh84Uz20bx0sQs1oYcek2YG5RhEdzqJ6W78R
S/dbzlNYMXGdkxB6C63m8oiGvw0hdN/iGVqpNAoldFmjnFqSgKpyPwfLmmdstJ6f
xFZdGPnKexCpHbKr9fg50jZRenIGai79qPIiEtCZHIdpeemSrc7TKRPV3H2aMNfG
L5HTqcyaM2+QrMtHPMoOFzcjkigLimMAEQEAAYkCHwQYAQIACQUCVNeM+AIbDAAK
CRDabO+N4RaZEo7lD/45J6z2wbL8aIudGEL0aY3hfmW3qrUyoHgaw35KsOY9vZwb
cZuJe0RlYptOreH/NrbR5SXODfhd2sxYyyvXBOuZh9i7OOBsrAd5UE01GCvToPwh
7IpMV3GSSAB4P8XyJh20tZqiZOYKhmbf29gUDzqAI6GzUa0U8xidUKpW2zqYGZjp
wk3RI1fS7tyi/0N8B9tIZF48kbvpFDAjF8w7NSCrgRquAL7zJZIG5o5zXJM/ffF3
67Dnz278MbifdM/HJ+Tj0R0Uvvki9Z61nT653SoUgvILQyC72XI+x0+3GQwsE38a
5aJNZ1NBD3/v+gERQxRfhM5iLFLXK0Xe4K2XFM1g0yN4L4bQPbhSCq88g9Dhmygk
XPbBsrK0NKPVnyGyUXM0VpgRbot11hxx02jC3HxS1nlLF+oQdkKFzJAMOU7UbpX/
oO+286J1FmpG+fihIbvp1Quq48immtnzTeLZbYCsG4mrM+ySYd0Er0G8TBdAOTiN
3zMbGX0QOO2fOsJ1d980cVjHn5CbAo8C0A/4/R2cXAfpacbvTiNq5BVk9NKa2dNb
kmnTStP2qILWmm5ASXlWhOjWNmptvsUcK+8T+uQboLioEv19Ob4j5Irs/OpOuP0K
v4woCi9+03HMS42qGSe/igClFO3+gUMZg9PJnTJhuaTbytXhUBgBRUPsS+lQAQ==
=DpoI
-----END PGP PUBLIC KEY BLOCK-----
```

CHAPTER 4

Compatibility with PyCrypto

PyCryptodome exposes *almost* the same API as the old PyCrypto so that *most* applications will run unmodified. However, a very few breaks in compatibility had to be introduced for those parts of the API that represented a security hazard or that were too hard to maintain.

Specifically, for public key cryptography:

- The following methods from public key objects (RSA, DSA, ElGamal) have been removed:
  - `sign()`
  - `verify()`
  - `encrypt()`
  - `decrypt()`
  - `blind()`
  - `unblind()`

  Applications should be updated to use instead:
  - `Crypto.Cipher.PKCS1_OAEP` for encrypting using RSA.
  - `Crypto.Signature.pkcs1_15` or `Crypto.Signature.pss` for signing using RSA.
  - `Crypto.Signature.DSS` for signing using DSA.
- Method: `generate()` for public key modules does not accept the `progress_func` parameter anymore.
- Ambiguous method `size` from RSA, DSA and ElGamal key objects have bene removed. Instead, use methods `size_in_bytes()` and `size_in_bits()` and check the documentation.
- The 3 public key object types (RSA, DSA, ElGamal) are now unpickable. You must use the `exportKey()` method of each key object and select a good output format: for private keys that means a good password-based encryption scheme.
- Removed attribute `Crypto.PublicKey.RSA.algorithmIdentifier`.
- Removed `Crypto.PublicKey.RSA.RSAImplementation` (which should have been private in the first place). Same for `Crypto.PublicKey.DSA.DSAImplementation`.

For symmetric key cryptography:

- Symmetric ciphers do not have ECB as default mode anymore. ECB is not semantically secure and it exposes correlation across blocks. An expression like `AES.new(key)` will now fail. If ECB is the desired mode, one has to explicitly use `AES.new(key, AES.MODE_ECB)`.
- `Crypto.Cipher.DES3` does not allow keys that degenerate to Single DES.
- Parameter `segment_size` cannot be 0 for the CFB mode.
- Parameters `disabled_shortcut` and `overflow` cannot be passed anymore to *Crypto.Util.Counter.new*. Parameter `allow_wraparound` is ignored (counter block wraparound will **always** be checked).
- The `counter` parameter of a CTR mode cipher must be generated via *Crypto.Util.Counter*. It cannot be a generic callable anymore.
- Keys for `Crypto.Cipher.ARC2`, `Crypto.Cipher.ARC4` and `Crypto.Cipher.Blowfish` must be at least 40 bits long (still very weak).

The following packages, modules and functions have been removed:

- `Crypto.Random.OSRNG`, `Crypto.Util.winrandom` and `Crypto.Random.randpool`. You should use `Crypto.Random` only.
- `Crypto.Cipher.XOR`. If you just want to XOR data, use *Crypto.Util.strxor*.
- `Crypto.Hash.new`. Use `Crypto.Hash.<algorithm>.new()` instead.
- `Crypto.Protocol.AllOrNothing`
- `Crypto.Protocol.Chaffing`
- `Crypto.Util.number.getRandomNumber`
- `Crypto.pct_warnings`

Others:

- Support for any Python version older than 2.4 is dropped. For Python 2.4 only, a dependency on the `ctypes` package is introduced.

CHAPTER 5

API documentation

`Crypto.Cipher` package

Introduction

The `Crypto.Cipher` package contains algorithms for protecting the confidentiality of data.

There are three types of encryption algorithms:

1. **Symmetric ciphers**: all parties use the same key, for both decrypting and encrypting data. Symmetric ciphers are typically very fast and can process very large amount of data.
2. **Asymmetric ciphers**: senders and receivers use different keys. Senders encrypt with *public* keys (non-secret) whereas receivers decrypt with *private* keys (secret). Asymmetric ciphers are typically very slow and can process only very small payloads. Example: oaep.
3. **Hybrid ciphers**: the two types of ciphers above can be combined in a construction that inherits the benefits of both. An *asymmetric* cipher is used to protect a short-lived symmetric key, and a *symmetric* cipher (under that key) encrypts the actual message.

API principles

The base API of a cipher is fairly simple:

- You instantiate a cipher object by calling the `new()` function from the relevant cipher module (e.g. `Crypto.Cipher.AES.new()`). The first parameter is always the *cryptographic key*; its length depends on the particular cipher. You can (and sometimes must) pass additional cipher- or mode-specific parameters to `new()` (such as a *nonce* or a *mode of operation*).
- For encrypting, you call the `encrypt()` method of the cipher object with the plaintext. The method returns the piece of ciphertext. For most algorithms, you may call `encrypt()` multiple times (i.e. once for each piece of plaintext).

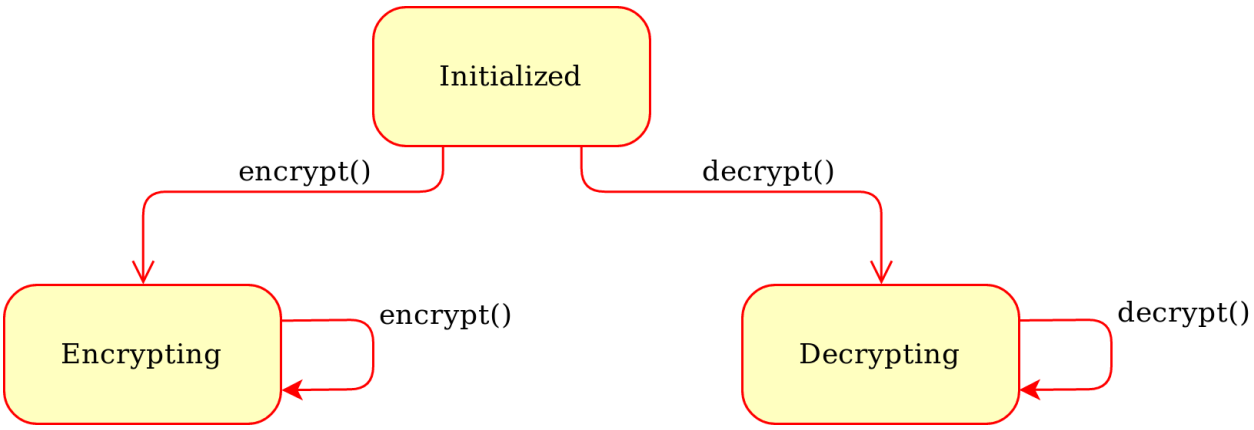

Fig. 5.1: Generic state diagram for a cipher object

- For decrypting, you call the `decrypt()` method of the cipher object with the ciphertext. The method returns the piece of plaintext. For most algorithms, you may call `decrypt()` multiple times (i.e. once for each piece of ciphertext).

Note: Plaintexts and ciphertexts (input/output) are all *byte strings*. An error will occur with Python 3 strings, Python 2 Unicode strings, or byte arrays.

Often, the sender has to deliver to the receiver other data in addition to ciphertext alone (e.g. **initialization vectors** or **nonces**, **MAC tags**, etc).

This is a basic example:

```
>>> from Crypto.Cipher import Salsa20
>>>
>>> key = b'0123456789012345'
>>> cipher = Salsa20.new(key)
>>> ciphertext =  cipher.encrypt(b'The secret I want to send.')
>>> ciphertext += cipher.encrypt(b'The second part of the secret.')
>>> print cipher.nonce  # A byte string you must send to the receiver too
```

Symmetric ciphers

There are two types of symmetric ciphers:

- **Stream ciphers**: the most natural kind of ciphers; they encrypt any piece of data by preserving its length: see chacha20, salsa20.
- **Block ciphers**: ciphers that can only operate on a fixed amount of data. The most important block cipher is aes, which has a block size of 16 bytes.

  A block cipher is in general useful only in combination with a *mode of operation* . There are *classic modes* like CTR or *authenticated modes* like GCM.

Classic modes of operation for symmetric block ciphers

Block ciphers are often only used together with a *mode of operation*.

When you create a block cipher object with the `new()` function, the second argument (after the cryptographic key) is a constant that sets the desired mode of operation. For instance:

```
>>> from Crypto.Cipher import AES
>>>
>>> cipher = AES.new(key, AES.MODE_CBC)
```

Constants are defined at the module level for each cipher algorithm, and their names start with `MODE_` (for instance `Crypto.Cipher.AES.MODE_CBC`).

This is the list of all classic modes (more modern modes are described in *another section*). Mind the not all modes are available for all block ciphers.

ECB mode

Constant: `Crypto.Cipher.<cipher>.MODE_ECB`.

Electronic CodeBook. A weak mode of operation whereby the cipher is applied in isolation to each of the blocks that compose the overall message.

This mode should not be used because it is not semantically secure and it exposes correlation between blocks.

`encrypt()` and `decrypt()` methods only accept data having length multiple of the block size.

CBC mode

Constant: `Crypto.Cipher.<cipher>.MODE_CBC`.

Ciphertext Block Chaining, defined in NIST SP 800-38A, section 6.2. It is a mode of operation where each plaintext block is XOR-ed with the last produced ciphertext block prior to encryption.

The `new()` function expects the following extra parameters:

- **`iv` (*byte string*): an unpredictable *Initialization Vector*** of length equal to the block size (e.g. 16 bytes for `Crypto.Cipher.AES`). If not present, a random IV will be created.

`encrypt()` and `decrypt()` methods only accept data with length multiple of the block size. You might need to use *`Crypto.Util.Padding`*.

The cipher object has a read-only attribute `iv`.

CFB mode

Constant: `Crypto.Cipher.<cipher>.MODE_CFB`.

Cipher FeedBack, defined in NIST SP 800-38A, section 6.3. It is a mode of operation which turns the block cipher into a stream cipher, with the plaintext getting XOR-ed with a *keystream* to obtain the ciphertext. The *keystream* is the last produced cipertext encrypted with the block cipher.

The `new()` function expects the following extra parameters:

- **`iv` (*byte string*): an non-repeatable *Initialization Vector*** of length equal to the block size (e.g. 16 bytes for `Crypto.Cipher.AES`). If not present, a random IV will be created.
- **`segment_size` (*integer*): the number of bits the plaintext and the** ciphertext are segmented in (default if not specified: 8).

  The cipher object has a read-only attribute `iv`.

OFB mode

Constant: `Crypto.Cipher.<cipher>.MODE_OFB`.

Output FeedBack, defined in NIST SP 800-38A, section 6.4. It is another mode that leads to a stream cipher. The *keystream* is obtained by recursively encrypting the *IV*.

The `new()` function expects the following extra parameters:

- **`iv` (*byte string*): an non-repeatable *Initialization Vector*** of length equal to the block size (e.g. 16 bytes for `Crypto.Cipher.AES`). If not present, a random IV will be created.

The cipher object has a read-only attribute `iv`.

CTR mode

Constant: `Crypto.Cipher.<cipher>.MODE_CTR`.

CounTeR mode, defined in NIST SP 800-38A, section 6.5 and Appendix B. It is another mode that leads to a stream cipher. The *keystream* is obtained by encrypting a *block counter*, which is the concatenation of a *nonce* (fixed during the computation) to a *counter field* (ever increasing).

The `new()` function expects the following extra parameters:

- **`nonce` (*byte string*): a mandatory non-repeatable value,** of length between 0 and block length minus 1.
- **`initial_value` (*integer*): the initial value for the counter field** (default if not specified: 0).

The cipher object has a read-only attribute `nonce`.

OpenPGP mode

Constant: `Crypto.Cipher.<cipher>.MODE_OPENPGP`.

OpenPGP (defined in RFC4880). A variant of CFB, with two differences:

1. The first invokation to the `encrypt()` method returns the encrypted IV concatenated to the first chunk on ciphertext (as opposed to the ciphertext only). The encrypted IV is as long as the block size plus 2 more bytes.
2. When the cipher object is intended for decryption, the parameter `iv` to `new()` is the encrypted IV (and not the IV, which is still the case for encryption).

Like for CTR, any cipher object has a read-only attribute `iv`.

Modern modes of operation for symmetric block ciphers

Classic modes of operation such as CBC only provide guarantees over the *confidentiality* of the message but not over its *integrity*. In other words, they don't allow the receiver to establish if the ciphertext was modified in transit or if it really originates from a certain source.

For that reason, classic modes of operation have been often paired with a MAC primitive (such as `Crypto.Hash.HMAC`), but the combination is not always straightforward, efficient or secure.

Recently, new modes of operations (AEAD, for Authenticated Encryption with Associated Data) have been designed to combine *encryption* and *authentication* into a single, efficient primitive. Optionally, some part of the message can also be left in the clear (non-confidential *associated data*, such as headers), while the whole message remains fully authenticated.

In addition to the **ciphertext** and a **nonce / IV**, AEAD modes require the additional delivery of a **MAC tag**.

The API of an AEAD cipher object is richer, as it include methods normally found in a MAC object:

- The `update()` method consumes data (if any) which must be authenticated but not encrypted. Note that any data passed to `encrypt()` or `decrypt()` is automatically authenticated.
- The `digest()` method creates an authentication tag (MAC tag) at the end of the encryption process (the variant `hexdigest()` exists to output the tag as a hexadecimal string).
- The `verify()` method checks if the provided authentication tag (MAC tag) is valid at the end of the decryption process (the variant `hexverify()` exists in case the MAC tag is a hexadecimal string).
- The `encrypt_and_digest()` method encrypts and creates a MAC tag in one go.
- The `decrypt_and_verify()` method decrypts and checks a MAC tag in one go.

The state machine for a cipher object becomes:

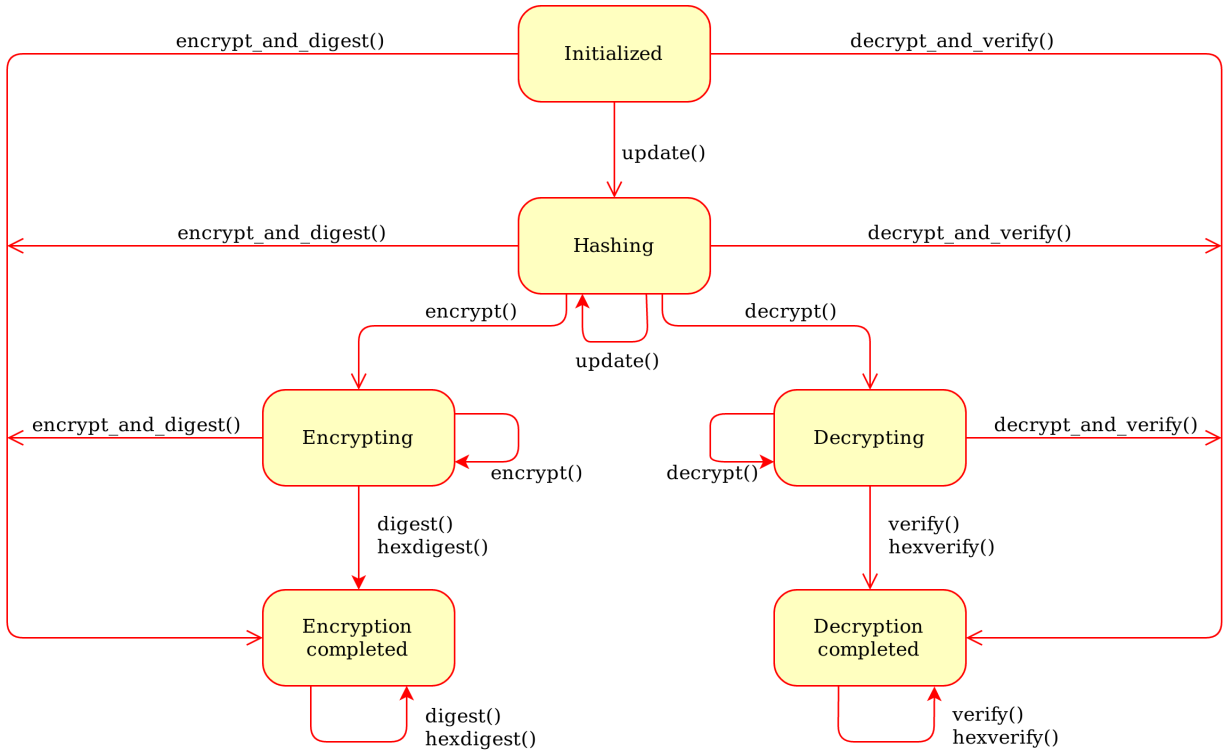

Fig. 5.2: Generic state diagram for a AEAD cipher mode

CCM mode

Constant: `Crypto.Cipher.<cipher>.MODE_CCM`.

Counter with CBC-MAC, defined in RFC3610 or NIST SP 800-38C. It only works with ciphers having block size 128 bits (like AES).

The `new()` function expects the following extra parameters:

- `nonce` (*byte string*): a non-repeatable value, of length between 7 and 13 bytes. The longer the nonce, the smaller the allowed message size (with a nonce of 13 bytes, the message cannot exceed 64KBi). If not present, a random 11 bytes long *nonce* will be created (the maximum message size is 8GBi).
- `mac_len` (*integer*): the desired length of the MAC tag (default if not present: 16 bytes).

- `msg_len` (*integer*): pre-declaration of the length of the message to encipher. If not specified, `encrypt()` and `decrypt()` can only be called once.
- `assoc_len` (*integer*): pre-declaration of the length of the associated data. If not specified, some extra buffering will take place internally.

The cipher object has a read-only attribute `nonce`.

EAX mode

Constant: `Crypto.Cipher.<cipher>.MODE_EAX`.

An AEAD mode designed for NIST by Bellare, Rogaway, and Wagner in 2003.

The `new()` function expects the following extra parameters:

- `nonce` (*byte string*): a non-repeatable value, of arbitrary length. If not present, a random *nonce* of the recommended length (16 bytes) will be created.
- `mac_len` (*integer*): the desired length of the MAC tag (default if not present: 16 bytes).

The cipher object has a read-only attribute `nonce`.

GCM mode

Constant: `Crypto.Cipher.<cipher>.MODE_GCM`.

Galois/Counter Mode, defined in NIST SP 800-38D. It only works in combination with a 128 bits cipher like AES.

The `new()` function expects the following extra parameters:

- `nonce` (*byte string*): a non-repeatable value, of arbitrary length. If not present, a random *nonce* of the recommended length (16 bytes) will be created.
- `mac_len` (*integer*): the desired length of the MAC tag (default if not present: 16 bytes).

The cipher object has a read-only attribute `nonce`.

SIV mode

Constant: `Crypto.Cipher.<cipher>.MODE_SIV`.

Synthetic Initialization Vector (SIV), defined in RFC5297. It only works with ciphers having block size 128 bits (like AES).

Although less efficient, SIV is unlike all other AEAD modes in that it is *nonce misuse-resistant*: the accidental reuse of a nonce does not have catastrophic effects as for CCM, GCM, etc. Instead, it will simply degrade into a **deterministic** cipher and therefore allow an attacker to know whether two ciphertexts contain the same message or not.

The `new()` function expects the following extra parameters:

- `nonce` (*byte string*): a non-repeatable value, of arbitrary length. If not present, the encryption will be **deterministic**.

The length of the key passed to `new()` must be twice as required by the underlying block cipher (e.g. 32 bytes for AES-128).

Each call to the method `update()` consumes an individual piece of associated data. That is, the sequence:

```
>>> siv_cipher.update(b"builtin")
>>> siv_cipher.update(b"securely")
```

is **not** equivalent to:

```
>>> siv_cipher.update(b"built")
>>> siv_cipher.update(b"insecurely")
```

The methods `encrypt()` and `decrypt()` can only be called **once**.

The cipher object has a read-only attribute `nonce`.

OCB mode

Constant: `Crypto.Cipher.<cipher>.MODE_OCB`.

Offset CodeBook mode, a cipher designed by Rogaway and specified in RFC7253 (more specifically, this module implements the last variant, OCB3). It only works in combination with a 128 bits cipher like AES.

OCB is patented in USA but free licenses exist for software implementations meant for non-military purposes and open source.

The `new()` function expects the following extra parameters:

- `nonce` (*byte string*): a non-repeatable value, of length between 1 and 15 bytes.. If not present, a random *nonce* of the recommended length (15 bytes) will be created.
- `mac_len` (*integer*): the desired length of the MAC tag (default if not present: 16 bytes).

The cipher object has a read-only attribute `nonce`.

Historic ciphers

A number of ciphers are implemented in this library purely for backward compatibility purposes. They are deprecated or even fully broken and should not be used in new designs.

- des and des3 (block ciphers)
- arc2 (block cipher)
- arc4 (stream cipher)
- blowfish (block cipher)
- cast (block cipher)
- pkcs1_v1_5 (asymmetric cipher)

`Crypto.Signature` package

The `Crypto.Signature` package contains algorithms for performing digital signatures, used to guarantee integrity and non-repudiation.

Digital signatures are based on public key cryptography: the party that signs a message holds the *private key*, the one that verifies the signature holds the *public key*.

Signing a message

1. You instatiate a new signer object using the `new()` method in the module of the desired algorithm. The first parameter is always the key object (*private* key) obtained via the `Crypto.PublicKey` module.
2. You instatiate a cryptographic hash (see `Crypto.Hash`) and digest the message with it.
3. You call `sign()` on the hash object. The output is the signature of the message (a byte string).

Verifying a signature

1. You instatiate a new verifier object using the `new()` method in the module of the desired algorithm. The first parameter is always the key object (*public* key) obtained via the `Crypto.PublicKey` module.
2. You instatiate a cryptographic hash (see `Crypto.Hash`) and digest the message with it.
3. You call `verify()` on the hash object and the incoming signature. If the message is not authentic, an `ValueError` is raised.

Available mechanisms

- pkcs1_v1_5
- pkcs1_pss
- dsa

`Crypto.Hash` package

Cryptographic hash functions take arbitrary binary strings as input, and produce a random-like fixed-length output (called *digest* or *hash value*).

It is practically infeasible to derive the original input data from the digest. In other words, the cryptographic hash function is *one-way* (*pre-image resistance*).

Given the digest of one message, it is also practically infeasible to find another message (*second pre-image*) with the same digest (*weak collision resistance*).

Finally, it is infeasible to find two arbitrary messages with the same digest (*strong collision resistance*).

Regardless of the hash algorithm, an *n* bits long digest is at most as secure as a symmetric encryption algorithm keyed with *n/2* bits (birthday attack).

Hash functions can be simply used as integrity checks. In combination with a public-key algorithm, you can implement a digital signature.

API principles

Every time you want to hash a message, you have to create a new hash object with the `new()` function in the relevant algorithm module (e.g. `Crypto.Hash.SHA256.new()`).

A first piece of message to hash can be passed to `new()` with the `data` parameter:

```
>> from Crypto.Hash import SHA256
>>
>> hash_object = SHA256.new(data=b'First')
```

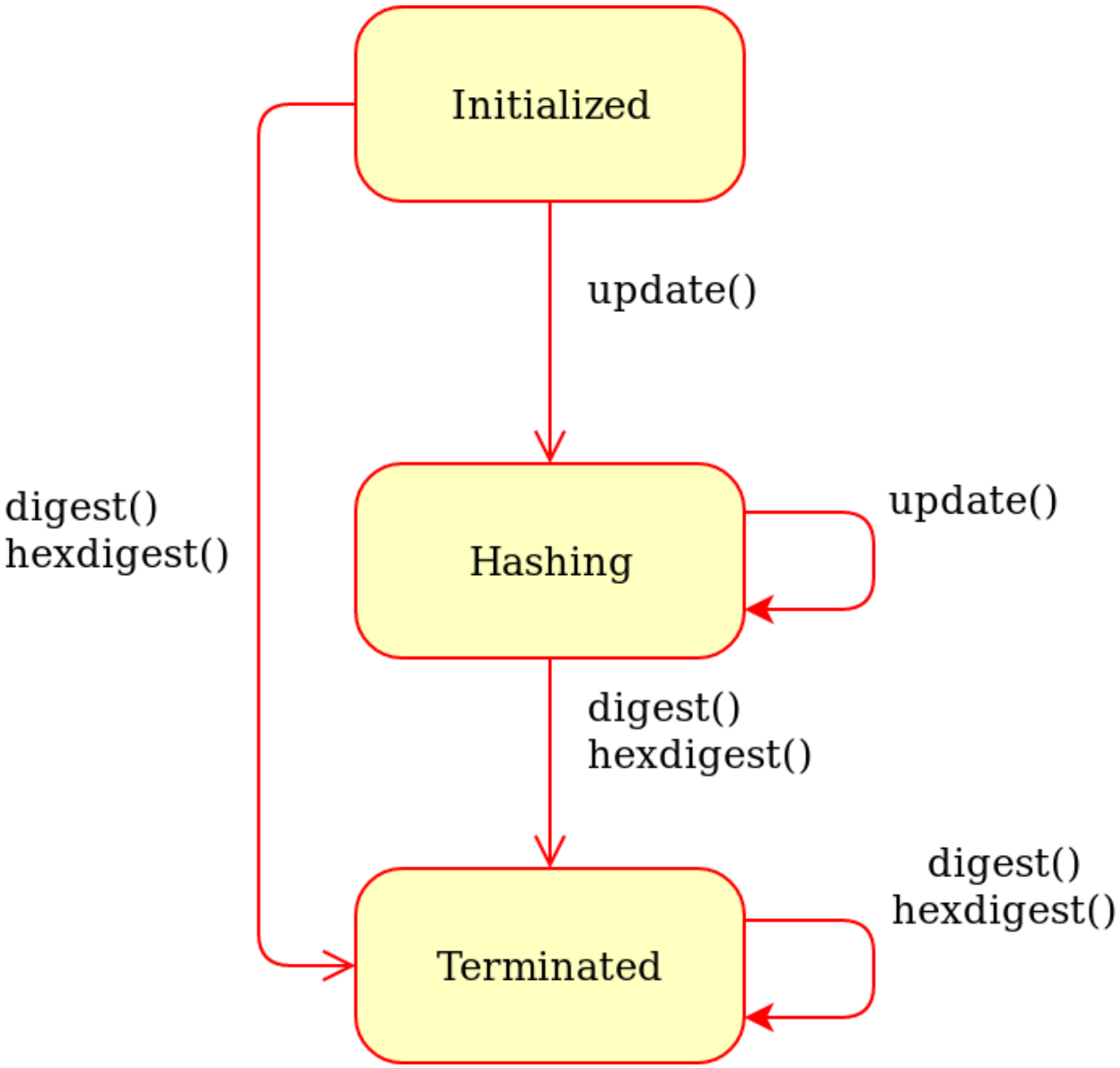

Fig. 5.3: Generic state diagram for a hash object

Note: You can only hash *byte strings* (no Python 2 Unicode strings, Python 3 strings or byte arrays).

Afterwards, the method `update()` can be invoked any number of times as necessary, with other pieces of message:

```
>>> hash_object.update(b'Second')
>>> hash_object.update(b'Third')
```

The two steps above are equivalent to:

```
>>> hash_object.update(b'SecondThird')
```

A the end, the digest can be retrieved with the methods `digest()` or `hexdigest()`:

```
>>> print(hash_object.digest())
b'}\x96\xfd@\xb2$?O\xca\xc1a\x10\x15\x8c\x94\xe4\xb4\x085"\xd5
↪"\xa8\xa4C\x9e+\x00\x859\xc7A'
>>> print(hash_object.hexdigest())
7d96fd40b2243f4fcac16110158c94e4b4083522d522a8a4439e2b008539c741
```

Attributes of hash objects

Every hash object has the following attributes:

Attribute	Description
digest_size	Size of the digest in bytes, that is, the output of the `digest()` method. It does not exist for hash functions with variable digest output (such as `Crypto.Hash.SHAKE128`). This is also a module attribute.
block_size	The size of the message block in bytes, input to the compression function. Only applicable for algorithms based on the Merkle-Damgard construction (e.g. `Crypto.Hash.SHA256`). This is also a module attribute.
oid	A string with the dotted representation of the ASN.1 OID assigned to the hash algorithm.

Modern hash algorithms

- SHA-2 family
  - sha224
  - sha256
  - sha384
  - sha512
- SHA-3 family
  - sha3_224
  - sha3_256
  - sha3_384
  - sha3_512
- BLAKE2

  - blake2s
  - blake2b

Extensible-Output Functions (XOF)

- SHAKE (in the SHA-3 family)
  - shake128
  - shake256

Message Authentication Code (MAC) algorithms

- hmac
- cmac

Historich hash algorithms

The following algorithms should not be used in new designs:

- sha1
- md2
- md5
- ripemd160
- keccak

`Crypto.PublicKey` package

In a public key cryptography system, senders and receivers do not use the same key. Instead, the system defines a *key pair*, with one of the keys being confidential (*private*) and the other not (*public*).

Algorithm	Sender uses..	Receiver uses...
Encryption	Public key	Private key
Signature	Private key	Public key

Unlike keys meant for symmetric cipher algorithms (typically just random bit strings), keys for public key algorithms have very specific properties. This module collects all methods to generate, validate, store and retrieve public keys.

API principles

Asymmetric keys are represented by Python objects. Each object can be either a *private* key or a *public* key (the method `has_private()` can be used to distinguish them).

A key object can be created in four ways:

1. `generate()` at the module level (e.g. *`Crypto.PublicKey.RSA.generate()`*). The key is randomly created each time.
2. `import_key()` at the module level (e.g. *`Crypto.PublicKey.RSA.import_key()`*). The key is loaded from memory.

3. `construct()` at the module level (e.g. *Crypto.PublicKey.RSA.construct()*). The key will be built from a set of sub-components.
4. `publickey()` at the object level (e.g. *Crypto.PublicKey.RSA.RsaKey.publickey()*). The key will be the public key matching the given object.

A key object can be serialized via its `exportKey()` method.

Keys objects can be compared via the usual operators == and != (note that the two halves of the same key, *private* and *public*, are considered as two different keys).

Available key types

RSA

RSA is the most widespread and used public key algorithm. Its security is based on the difficulty of factoring large integers. The algorithm has withstood attacks for more than 30 years, and it is therefore considered reasonably secure for new designs.

The algorithm can be used for both confidentiality (encryption) and authentication (digital signature). It is worth noting that signing and decryption are significantly slower than verification and encryption.

The cryptograhic strength is primarily linked to the length of the RSA modulus *n*. In 2017, a sufficient length is deemed to be 2048 bits. For more information, see the most recent ECRYPT report.

Both RSA ciphertexts and RSA signatures are as large as the RSA modulus *n* (256 bytes if *n* is 2048 bit long).

The module *Crypto.PublicKey.RSA* provides facilities for generating new RSA keys, reconstructing them from known components, exporting them, and importing them.

As an example, this is how you generate a new RSA key pair, save it in a file called `mykey.pem`, and then read it back:

```
>>> from Crypto.PublicKey import RSA
>>>
>>> key = RSA.generate(2048)
>>> f = open('mykey.pem','w')
>>> f.write(key.exportKey('PEM'))
>>> f.close()
...
>>> f = open('mykey.pem','r')
>>> key = RSA.import_key(f.read())
```

Crypto.PublicKey.RSA.**generate**(*bits*, *randfunc=None*, *e=65537*)

Create a new RSA key pair.

The algorithm closely follows NIST FIPS 186-4 in its sections B.3.1 and B.3.3. The modulus is the product of two non-strong probable primes. Each prime passes a suitable number of Miller-Rabin tests with random bases and a single Lucas test.

Parameters

- **bits** (*integer*) – Key length, or size (in bits) of the RSA modulus. It must be at least 1024, but **2048 is recommended.** The FIPS standard only defines 1024, 2048 and 3072.
- **randfunc** (*callable*) – Function that returns random bytes. The default is *Crypto.Random.get_random_bytes()*.

- **e** (*integer*) – Public RSA exponent. It must be an odd positive integer. It is typically a small number with very few ones in its binary representation. The FIPS standard requires the public exponent to be at least 65537 (the default).

Returns: an RSA key object (*RsaKey*, with private key).

`Crypto.PublicKey.RSA.`**`construct`**(*rsa_components*, *consistency_check=True*)

Construct an RSA key from a tuple of valid RSA components.

The modulus **n** must be the product of two primes. The public exponent **e** must be odd and larger than 1.

In case of a private key, the following equations must apply:

$$p * q = n \tag{5.1}$$
$$e * d \equiv 1 (\text{mod } \text{lcm}[(p-1)(q-1)]) \tag{5.2}$$
$$p * u \equiv 1 (\text{mod } q) \tag{5.3}$$

Parameters

- **rsa_components** (*tuple*) – A tuple of integers, with at least 2 and no more than 6 items. The items come in the following order:
  1. RSA modulus *n*.
  2. Public exponent *e*.
  3. Private exponent *d*. Only required if the key is private.
  4. First factor of *n* (*p*). Optional, but the other factor *q* must also be present.
  5. Second factor of *n* (*q*). Optional.
  6. CRT coefficient *q*, that is $p^{-1} \text{mod } q$. Optional.
- **consistency_check** (*boolean*) – If `True`, the library will verify that the provided components fulfil the main RSA properties.

Raises `ValueError` – when the key being imported fails the most basic RSA validity checks.

Returns: An RSA key object (*RsaKey*).

`Crypto.PublicKey.RSA.`**`import_key`**(*extern_key*, *passphrase=None*)

Import an RSA key (public or private half), encoded in standard form.

Parameters

- **extern_key** (*string or byte string*) – The RSA key to import.

  The following formats are supported for an RSA **public key**:

  – X.509 certificate (binary or PEM format)

  – X.509 `subjectPublicKeyInfo` DER SEQUENCE (binary or PEM encoding)

  – PKCS#1 `RSAPublicKey` DER SEQUENCE (binary or PEM encoding)

  – OpenSSH (textual public key only)

  The following formats are supported for an RSA **private key**:

  – PKCS#1 `RSAPrivateKey` DER SEQUENCE (binary or PEM encoding)

  – PKCS#8 `PrivateKeyInfo` or `EncryptedPrivateKeyInfo` DER SEQUENCE (binary or PEM encoding)

  – OpenSSH (textual public key only)

For details about the PEM encoding, see RFC1421/RFC1423.

The private key may be encrypted by means of a certain pass phrase either at the PEM level or at the PKCS#8 level.

- **passphrase** (*string*) – In case of an encrypted private key, this is the pass phrase from which the decryption key is derived.

Returns: An RSA key object (*RsaKey*).

Raises ValueError/IndexError/TypeError – When the given key cannot be parsed (possibly because the pass phrase is wrong).

class Crypto.PublicKey.RSA.**RsaKey**(***kwargs*)

Class defining an actual RSA key. Do not instantiate directly. Use *generate()*, *construct()* or *import_key()* instead.

Variables

- **n** (*integer*) – RSA modulus
- **e** (*integer*) – RSA public exponent
- **d** (*integer*) – RSA private exponent
- **p** (*integer*) – First factor of the RSA modulus
- **q** (*integer*) – Second factor of the RSA modulus
- **u** – Chinese remainder component ($p^{-1} \bmod q$)

exportKey(*format='PEM'*, *passphrase=None*, *pkcs=1*, *protection=None*, *randfunc=None*)

Export this RSA key.

Parameters

- **format** (*string*) – The format to use for wrapping the key:
  - *'PEM'*. (*Default*) Text encoding, done according to RFC1421/RFC1423.
  - *'DER'*. Binary encoding.
  - *'OpenSSH'*. Textual encoding, done according to OpenSSH specification. Only suitable for public keys (not private keys).
- **passphrase** (*string*) – (*For private keys only*) The pass phrase used for protecting the output.
- **pkcs** (*integer*) – (*For private keys only*) The ASN.1 structure to use for serializing the key. Note that even in case of PEM encoding, there is an inner ASN.1 DER structure.

  With `pkcs=1` (*default*), the private key is encoded in a simple PKCS#1 structure (`RSAPrivateKey`).

  With `pkcs=8`, the private key is encoded in a PKCS#8 structure (`PrivateKeyInfo`).

  Note: This parameter is ignored for a public key. For DER and PEM, an ASN.1 DER `SubjectPublicKeyInfo` structure is always used.

- **protection** (*string*) – (*For private keys only*) The encryption scheme to use for protecting the private key.

  If `None` (default), the behavior depends on `format`:

– For *'DER'*, the *PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC* scheme is used. The following operations are performed:

1. A 16 byte Triple DES key is derived from the passphrase using *Crypto.Protocol.KDF.PBKDF2()* with 8 bytes salt, and 1 000 iterations of `Crypto.Hash.HMAC`.
2. The private key is encrypted using CBC.
3. The encrypted key is encoded according to PKCS#8.

– For *'PEM'*, the obsolete PEM encryption scheme is used. It is based on MD5 for key derivation, and Triple DES for encryption.

Specifying a value for `protection` is only meaningful for PKCS#8 (that is, `pkcs=8`) and only if a pass phrase is present too.

The supported schemes for PKCS#8 are listed in the *Crypto.IO.PKCS8* module (see `wrap_algo` parameter).

- **randfunc** (*callable*) – A function that provides random bytes. Only used for PEM encoding. The default is *Crypto.Random.get_random_bytes()*.

Returns the encoded key

Return type byte string

Raises `ValueError` – when the format is unknown or when you try to encrypt a private key with *DER* format and PKCS#1.

Warning: If you don't provide a pass phrase, the private key will be exported in the clear!

has_private()
Whether this is an RSA private key

publickey()
A matching RSA public key.

Returns a new *RsaKey* object

size_in_bits()
Size of the RSA modulus in bits

size_in_bytes()
The minimal amount of bytes that can hold the RSA modulus

`Crypto.PublicKey.RSA.`**oid = '1.2.840.113549.1.1.1'**
Object ID for the RSA encryption algorithm. This OID often indicates a generic RSA key, even when such key will be actually used for digital signatures.

DSA

DSA is a widespread public key signature algorithm. Its security is based on the discrete logarithm problem (DLP). Given a cyclic group, a generator g, and an element h, it is hard to find an integer x such that $g^x = h$. The problem is believed to be difficult, and it has been proved such (and therefore secure) for more than 30 years.

The group is actually a sub-group over the integers modulo p, with p prime. The sub-group order is q, which is prime too; it always holds that $(p-1)$ is a multiple of q. The cryptographic strength is linked to the magnitude of p and q. The signer holds a value x ($0<x<q-1$) as private key, and its public key (y where $y = g^x \bmod p$) is distributed.

In 2017, a sufficient size is deemed to be 2048 bits for p and 256 bits for q. For more information, see the most recent ECRYPT report.

The algorithm can only be used for authentication (digital signature). DSA cannot be used for confidentiality (encryption).

The values *(p,q,g)* are called *domain parameters*; they are not sensitive but must be shared by both parties (the signer and the verifier). Different signers can share the same domain parameters with no security concerns.

The DSA signature is twice as big as the size of q (64 bytes if q is 256 bit long).

This module provides facilities for generating new DSA keys and for constructing them from known components.

As an example, this is how you generate a new DSA key pair, save the public key in a file called `public_key.pem`, sign a message (with `Crypto.Signature.DSS`), and verify it:

```
>>> from Crypto.PublicKey import DSA
>>> from Crypto.Signature import DSS
>>> from Crypto.Hash import SHA256
>>>
>>> # Create a new DSA key
>>> key = DSA.generate(2048)
>>> f = open("public_key.pem", "w")
>>> f.write(key.publickey().exportKey(key))
>>>
>>> # Sign a message
>>> message = b"Hello"
>>> hash_obj = SHA256.new(message)
>>> signer = DSS.new(key, 'fips-186-3')
>>> signature = key.sign(hash_obj)
>>>
>>> # Load the public key
>>> f = open("public_key.pem", "r")
>>> hash_obj = SHA256.new(message)
>>> pub_key = DSA.import_key(f.read())
>>>
>>> # Verify the authenticity of the message
>>> if pub_key.verify(hash_obj, signature):
>>>     print "OK"
>>> else:
>>>     print "Incorrect signature"
```

`Crypto.PublicKey.DSA.`**`generate`**(*bits*, *randfunc=None*, *domain=None*)

Generate a new DSA key pair.

The algorithm follows Appendix A.1/A.2 and B.1 of FIPS 186-4, respectively for domain generation and key pair generation.

Parameters

- **`bits`** (*integer*) – Key length, or size (in bits) of the DSA modulus p. It must be 1024, 2048 or 3072.
- **`randfunc`** (*callable*) – Random number generation function; it accepts a single integer N and return a string of random data N bytes long. If not specified, *`Crypto.Random.get_random_bytes()`* is used.
- **`domain`** (*tuple*) – The DSA domain parameters p, q and g as a list of 3 integers. Size of p and q must comply to FIPS 186-4. If not specified, the parameters are created anew.

Returns a new DSA key object

Return type *DsaKey*

Raises `ValueError` – when **bits** is too little, too big, or not a multiple of 64.

`Crypto.PublicKey.DSA.`**construct**(*tup*, *consistency_check=True*)

Construct a DSA key from a tuple of valid DSA components.

Parameters

- **tup** (*tuple*) – A tuple of long integers, with 4 or 5 items in the following order:
  1. Public key (*y*).
  2. Sub-group generator (*g*).
  3. Modulus, finite field order (*p*).
  4. Sub-group order (*q*).
  5. Private key (*x*). Optional.
- **consistency_check** (*boolean*) – If `True`, the library will verify that the provided components fulfil the main DSA properties.

Raises `ValueError` – when the key being imported fails the most basic DSA validity checks.

Returns a DSA key object

Return type *DsaKey*

class `Crypto.PublicKey.DSA.`**DsaKey**(*key_dict*)

Class defining an actual DSA key. Do not instantiate directly. Use *generate()*, *construct()* or *import_key()* instead.

Variables

- **p** (*integer*) – DSA modulus
- **q** (*integer*) – Order of the subgroup
- **g** (*integer*) – Generator
- **y** (*integer*) – Public key
- **x** (*integer*) – Private key

domain()

The DSA domain parameters.

Returns tuple : (p,q,g)

exportKey(*format='PEM'*, *pkcs8=None*, *passphrase=None*, *protection=None*, *randfunc=None*)

Export this DSA key.

Parameters

- **format** (*string*) – The encoding for the output:
  - *'PEM'* (default). ASCII as per RFC1421/ RFC1423.
  - *'DER'*. Binary ASN.1 encoding.
  - *'OpenSSH'*. ASCII one-liner as per RFC4253. Only suitable for public keys, not for private keys.
- **passphrase** (*string*) – *Private keys only*. The pass phrase to protect the output.
- **pkcs8** (*boolean*) – *Private keys only*. If `True` (default), the key is encoded with PKCS#8. If `False`, it is encoded in the custom OpenSSL/OpenSSH container.

- **protection** (*string*) – *Only in combination with a pass phrase*. The encryption scheme to use to protect the output.

  If `pkcs8` takes value `True`, this is the PKCS#8 algorithm to use for deriving the secret and encrypting the private DSA key. For a complete list of algorithms, see *Crypto.IO.PKCS8*. The default is *PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC*.

  If `pkcs8` is `False`, the obsolete PEM encryption scheme is used. It is based on MD5 for key derivation, and Triple DES for encryption. Parameter `protection` is then ignored.

  The combination `format='DER'` and `pkcs8=False` is not allowed if a passphrase is present.

- **randfunc** (*callable*) – A function that returns random bytes. By default it is *Crypto.Random.get_random_bytes()*.

Returns the encoded key

Return type byte string

Raises `ValueError` – when the format is unknown or when you try to encrypt a private key with *DER* format and OpenSSL/OpenSSH.

> **Warning:** If you don't provide a pass phrase, the private key will be exported in the clear!

has_private()
Whether this is a DSA private key

publickey()
A matching DSA public key.

Returns a new *DsaKey* object

Crypto.PublicKey.DSA.**import_key**(*extern_key*, *passphrase=None*)
Import a DSA key.

Parameters

- **extern_key** (*string or byte string*) – The DSA key to import.

  The following formats are supported for a DSA **public** key:

  – X.509 certificate (binary DER or PEM)

  – X.509 `subjectPublicKeyInfo` (binary DER or PEM)

  – OpenSSH (ASCII one-liner, see RFC4253)

  The following formats are supported for a DSA **private** key:

  – PKCS#8 `PrivateKeyInfo` or `EncryptedPrivateKeyInfo` DER SEQUENCE (binary or PEM)

  – OpenSSL/OpenSSH custom format (binary or PEM)

  For details about the PEM encoding, see RFC1421/RFC1423.

- **passphrase** (*string*) – In case of an encrypted private key, this is the pass phrase from which the decryption key is derived.

  Encryption may be applied either at the PKCS#8 or at the PEM level.

Returns a DSA key object

Return type *DsaKey*

Raises `ValueError` – when the given key cannot be parsed (possibly because the pass phrase is wrong).

ECC

ECC (Elliptic Curve Cryptography) is a modern and efficient type of public key cryptography. Its security is based on the difficulty to solve discrete logarithms on the field defined by specific equations computed over a curve.

ECC can be used to create digital signatures or encrypting data.

The main benefit of ECC is that the size of a key is significantly smaller than with more traditional algorithms like RSA or DSA.

For instance, consider the security level equivalent to AES128: an RSA key of similar strength must have a modulus of 3072 bits (therefore the total size is 768 bytes, comprising modulus and private exponent). An ECC private needs as little as 256 bits (32 bytes).

This module provides mechanisms for generating new ECC keys, exporting them using widely supported formats like PEM or DER and importing them back.

Note: This module currently supports only ECC keys defined over the standard **NIST P-256 curve** (see FIPS 186-4, Section D.1.2.3). More curves will be added in the future.

The following example demonstrates how to generate a new key, export it, and subsequentely reload it back into the application:

```
>>> from Crypto.PublicKey import ECC
>>>
>>> key = ECC.generate(curve='P-256')
>>>
>>> f = open('myprivatekey.pem','wt')
>>> f.write(key.export_key('PEM'))
>>> f.close()
...
>>> f = open('myprivatekey.pem','rt')
>>> key = RSA.import_key(f.read())
```

The ECC key can be used to perform or verify ECDSA signatures, using the module `Crypto.Signature.DSS`.

class `Crypto.PublicKey.ECC.`**`EccKey`**(***kwargs*)

Class defining an ECC key. Do not instantiate directly. Use *generate()*, *construct()* or *import_key()* instead.

Variables

- **`curve`** (*string*) – The name of the ECC curve
- **`pointQ`** (*EccPoint*) – an ECC point representating the public component
- **`q`** (*integer*) – A scalar representating the private component

`export_key`(***kwargs*)

Export this ECC key.

Parameters

- **format** (*string*) – The format to use for wrapping the key:
  - *'DER'*. The key will be encoded in an ASN.1 DER structure (binary).
  - *'PEM'*. The key will be encoded in a PEM envelope (ASCII).
  - *'OpenSSH'*. The key will be encoded in the OpenSSH format (ASCII, public keys only).
- **passphrase** (*byte string or string*) – The passphrase to use for protecting the private key.
- **use_pkcs8** (*boolean*) – If `True` (default and recommended), the PKCS#8 representation will be used.

  If `False`, the much weaker and PEM encryption mechanism will be used.
- **protection** (*string*) – When a private key is exported with password-protection and PKCS#8 (both `DER` and `PEM` formats), this parameter MUST be present and be a valid algorithm supported by *Crypto.IO.PKCS8*. It is recommended to use `PBKDF2WithHMAC-SHA1AndAES128-CBC`.

Warning: If you don't provide a passphrase, the private key will be exported in the clear!

Note: When exporting a private key with password-protection and PKCS#8 (both `DER` and `PEM` formats), any extra parameters is passed to *Crypto.IO.PKCS8*.

Returns A multi-line string (for PEM and OpenSSH) or bytes (for DER) with the encoded key.

has_private()
`True` if this key can be used for making signatures or decrypting data.

public_key()
A matching ECC public key.

Returns a new *EccKey* object

class Crypto.PublicKey.ECC.**EccPoint**(*x*, *y*)
A class to abstract a point over an Elliptic Curve.

Variables

- **x** (*integer*) – The X-coordinate of the ECC point
- **y** (*integer*) – The Y-coordinate of the ECC point

double()
Double this point (in-place operation).

Return *EccPoint* : this same object (to enable chaining)

Crypto.PublicKey.ECC.**construct**(***kwargs*)
Build a new ECC key (private or public) starting from some base components.

Parameters

- **curve** (*string*) – Mandatory. It must be "P-256", "prime256v1" or "secp256r1".
- **d** (*integer*) – Only for a private key. It must be in the range `[1..order-1]`.

- **point_x** (*integer*) – Mandatory for a public key. X coordinate (affine) of the ECC point.
- **point_y** (*integer*) – Mandatory for a public key. Y coordinate (affine) of the ECC point.

Returns a new ECC key object

Return type *EccKey*

Crypto.PublicKey.ECC.**generate**(***kwargs*)

Generate a new private key on the given curve.

Parameters

- **curve** (*string*) – Mandatory. It must be "P-256", "prime256v1" or "secp256r1".
- **randfunc** (*callable*) – Optional. The RNG to read randomness from. If None, *Crypto.Random.get_random_bytes()* is used.

Crypto.PublicKey.ECC.**import_key**(*encoded*, *passphrase=None*)

Import an ECC key (public or private).

Parameters

- **encoded** (*bytes or multi-line string*) – The ECC key to import.

  An ECC **public** key can be:

  - An X.509 certificate, binary (DER) or ASCII (PEM)
  - An X.509 subjectPublicKeyInfo, binary (DER) or ASCII (PEM)
  - An OpenSSH line (e.g. the content of ~/.ssh/id_ecdsa, ASCII)

  An ECC **private** key can be:

  - In binary format (DER, see section 3 of RFC5915 or PKCS#8)
  - In ASCII format (PEM or OpenSSH)

  Private keys can be in the clear or password-protected.

  For details about the PEM encoding, see RFC1421/RFC1423.

- **passphrase** (*byte string*) – The passphrase to use for decrypting a private key. Encryption may be applied protected at the PEM level or at the PKCS#8 level. This parameter is ignored if the key in input is not encrypted.

Returns a new ECC key object

Return type *EccKey*

Raises ValueError – when the given key cannot be parsed (possibly because the pass phrase is wrong).

- *RSA keys*
- *DSA keys*
- *Elliptic Curve keys*

Obsolete key type

El Gamal

Warning: Even though ElGamal algorithms are in theory reasonably secure, in practice there are no real good reasons to prefer them to *RSA* instead.

Signature algorithm

The security of the ElGamal signature scheme is based (like DSA) on the discrete logarithm problem (DLP). Given a cyclic group, a generator g, and an element h, it is hard to find an integer x such that $g^x = h$.

The group is the largest multiplicative sub-group of the integers modulo p, with p prime. The signer holds a value x ($0<x<p-1$) as private key, and its public key (y where $y = g^x \bmod p$) is distributed.

The ElGamal signature is twice as big as p.

Encryption algorithm

The security of the ElGamal encryption scheme is based on the computational Diffie-Hellman problem (CDH). Given a cyclic group, a generator g, and two integers a and b, it is difficult to find the element g^{ab} when only g^a and g^b are known, and not a and b.

As before, the group is the largest multiplicative sub-group of the integers modulo p, with p prime. The receiver holds a value a ($0<a<p-1$) as private key, and its public key (b where $b = g^a$) is given to the sender.

The ElGamal ciphertext is twice as big as p.

Domain parameters

For both signature and encryption schemes, the values (p,g) are called *domain parameters*. They are not sensitive but must be distributed to all parties (senders and receivers). Different signers can share the same domain parameters, as can different recipients of encrypted messages.

Security

Both DLP and CDH problem are believed to be difficult, and they have been proved such (and therefore secure) for more than 30 years.

The cryptographic strength is linked to the magnitude of p. In 2017, a sufficient size for p is deemed to be 2048 bits. For more information, see the most recent ECRYPT report.

The signature is four times larger than the equivalent DSA, and the ciphertext is two times larger than the equivalent RSA.

Functionality

This module provides facilities for generating new ElGamal keys and constructing them from known components.

Crypto.PublicKey.ElGamal.**generate**(*bits*, *randfunc*)
Randomly generate a fresh, new ElGamal key.

The key will be safe for use for both encryption and signature (although it should be used for **only one** purpose).

Parameters

- **bits** (*int*) – Key length, or size (in bits) of the modulus *p*. The recommended value is 2048.
- **randfunc** (*callable*) – Random number generation function; it should accept a single integer *N* and return a string of random *N* random bytes.

Returns an *ElGamalKey* object

Crypto.PublicKey.ElGamal.**construct**(*tup*)
Construct an ElGamal key from a tuple of valid ElGamal components.

The modulus *p* must be a prime. The following conditions must apply:

$$1 < g < p - 1 \tag{5.4}$$
$$g^{p-1} = 1 \bmod p \tag{5.5}$$
$$1 < x < p \tag{5.6}$$
$$g^x = y \bmod p \tag{5.7}$$

Parameters **tup** (*tuple*) – A tuple with either 3 or 4 integers, in the following order:

1. Modulus (*p*).
2. Generator (*g*).
3. Public key (*y*).
4. Private key (*x*). Optional.

Raises ValueError – when the key being imported fails the most basic ElGamal validity checks.

Returns an *ElGamalKey* object

class Crypto.PublicKey.ElGamal.**ElGamalKey**(*randfunc=None*)
Class defining an ElGamal key. Do not instantiate directly. Use *generate()* or *construct()* instead.

Variables

- **p** – Modulus
- **g** – Generator
- **y** (*integer*) – Public key component
- **x** (*integer*) – Private key component

has_private()
Whether this is an ElGamal private key

publickey()
A matching ElGamal public key.

Returns a new *ElGamalKey* object

- *ElGamal keys*

Crypto.Protocol package

Key Derivation Functions

This module contains a collection of standard key derivation functions.

A key derivation function derives one or more secondary secret keys from one primary secret (a master key or a pass phrase).

This is typically done to insulate the secondary keys from each other, to avoid that leakage of a secondary key compromises the security of the master key, or to thwart attacks on pass phrases (e.g. via rainbow tables).

Crypto.Protocol.KDF.**HKDF**(*master*, *key_len*, *salt*, *hashmod*, *num_keys=1*, *context=None*)

Derive one or more keys from a master secret using the HMAC-based KDF defined in RFC5869.

This KDF is not suitable for deriving keys from a password or for key stretching. Use *PBKDF2()* instead.

HKDF is a key derivation method approved by NIST in SP 800 56C.

Parameters

- **master** (*byte string*) – The unguessable value used by the KDF to generate the other keys. It must be a high-entropy secret, though not necessarily uniform. It must not be a password.
- **salt** (*byte string*) – A non-secret, reusable value that strengthens the randomness extraction step. Ideally, it is as long as the digest size of the chosen hash. If empty, a string of zeroes in used.
- **key_len** (*integer*) – The length in bytes of every derived key.
- **hashmod** (*module*) – A cryptographic hash algorithm from Crypto.Hash. Crypto.Hash.SHA512 is a good choice.
- **num_keys** (*integer*) – The number of keys to derive. Every key is key_len bytes long. The maximum cumulative length of all keys is 255 times the digest size.
- **context** (*byte string*) – Optional identifier describing what the keys are used for.

Returns A byte string or a tuple of byte strings.

Crypto.Protocol.KDF.**PBKDF1**(*password*, *salt*, *dkLen*, *count=1000*, *hashAlgo=None*)

Derive one key from a password (or passphrase).

This function performs key derivation according to an old version of the PKCS#5 standard (v1.5) or RFC2898.

> **Warning:** Newer applications should use the more secure and versatile *PBKDF2()* instead.

Parameters

- **password** (*string*) – The secret password to generate the key from.
- **salt** (*byte string*) – An 8 byte string to use for better protection from dictionary attacks. This value does not need to be kept secret, but it should be randomly chosen for each derivation.
- **dkLen** (*integer*) – The length of the desired key. The default is 16 bytes, suitable for instance for Crypto.Cipher.AES.
- **count** (*integer*) – The number of iterations to carry out. The recommendation is 1000 or more.

- **hashAlgo** (*module*) – The hash algorithm to use, as a module or an object from the `Crypto.Hash` package. The digest length must be no shorter than `dkLen`. The default algorithm is `Crypto.Hash.SHA1`.

Returns A byte string of length `dkLen` that can be used as key.

`Crypto.Protocol.KDF.`**PBKDF2** (*password*, *salt*, *dkLen=16*, *count=1000*, *prf=None*)

Derive one or more keys from a password (or passphrase).

This function performs key derivation according to the PKCS#5 standard (v2.0).

Parameters

- **password** (*string*) – The secret password to generate the key from.
- **salt** (*string*) – A string to use for better protection from dictionary attacks. This value does not need to be kept secret, but it should be randomly chosen for each derivation. It is recommended to be at least 8 bytes long.
- **dkLen** (*integer*) – The cumulative length of the desired keys.
- **count** (*integer*) – The number of iterations to carry out.
- **prf** (*callable*) – A pseudorandom function. It must be a function that returns a pseudorandom string from two parameters: a secret and a salt. If not specified, **HMAC-SHA1** is used.

Returns A byte string of length `dkLen` that can be used as key material. If you wanted multiple keys, just break up this string into segments of the desired length.

`Crypto.Protocol.KDF.`**scrypt** (*password*, *salt*, *key_len*, *N*, *r*, *p*, *num_keys=1*)

Derive one or more keys from a passphrase.

This function performs key derivation according to the scrypt algorithm, introduced in Percival's paper "Stronger key derivation via sequential memory-hard functions".

This implementation is based on RFC7914.

Parameters

- **password** (*string*) – The secret pass phrase to generate the keys from.
- **salt** (*string*) – A string to use for better protection from dictionary attacks. This value does not need to be kept secret, but it should be randomly chosen for each derivation. It is recommended to be at least 8 bytes long.
- **key_len** (*integer*) – The length in bytes of every derived key.
- **N** (*integer*) – CPU/Memory cost parameter. It must be a power of 2 and less than 2^{32}.
- **r** (*integer*) – Block size parameter.
- **p** (*integer*) – Parallelization parameter. It must be no greater than $(2^{32} - 1)/(4r)$.
- **num_keys** (*integer*) – The number of keys to derive. Every key is `key_len` bytes long. By default, only 1 key is generated. The maximum cumulative length of all keys is $(2^{32} - 1) * 32$ (that is, 128TB).

A good choice of parameters *(N, r , p)* was suggested by Colin Percival in his presentation in 2009:

•*(16384, 8, 1)* for interactive logins (<=100ms)

•*(1048576, 8, 1)* for file encryption (<=5s)

Returns A byte string or a tuple of byte strings.

Secret Sharing Schemes

This file implements secret sharing protocols.

In a *(k, n)* secret sharing protocol, a honest dealer breaks a secret into multiple shares that are distributed amongst *n* players.

The protocol guarantees that nobody can learn anything about the secret, unless *k* players gather together to assemble their shares.

class `Crypto.Protocol.SecretSharing.`**`Shamir`**

Shamir's secret sharing scheme.

This class implements the Shamir's secret sharing protocol described in his original paper "How to share a secret".

All shares are points over a 2-dimensional curve. At least *k* points (that is, shares) are required to reconstruct the curve, and therefore the secret.

This implementation is primarilly meant to protect AES128 keys. To that end, the secret is associated to a curve in the field GF(2^128) defined by the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$ (the same used in AES-GCM). The shares are always 16 bytes long.

Data produced by this implementation are compatible to the popular ssss tool if used with 128 bit security (parameter *"-s 128"*) and no dispersion (parameter *"-D"*).

As an example, the following code shows how to protect a file meant for 5 people, in such a way that 2 of the 5 are required to reassemble it:

```
>>> from binascii import hexlify
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>> from Crypto.Protocol.secret_sharing import Shamir
>>>
>>> key = get_random_bytes(16)
>>> shares = Shamir.split(2, 5, key)
>>> for idx, share in shares:
>>>     print "Index #%d: %s" % (idx, hexlify(share))
>>>
>>> fi = open("clear_file.txt", "rb")
>>> fo = open("enc_file.txt", "wb")
>>>
>>> cipher = AES.new(key, AES.MODE_EAX)
>>> ct, tag = cipher.encrypt(fi.read()), cipher.digest()
>>> fo.write(nonce + tag + ct)
```

Each person can be given one share and the encrypted file.

When 2 people gather together with their shares, the can decrypt the file:

```
>>> from binascii import unhexlify
>>> from Crypto.Cipher import AES
>>> from Crypto.Protocol.secret_sharing import Shamir
>>>
>>> shares = []
>>> for x in range(2):
>>>     in_str = raw_input("Enter index and share separated by comma: ")
>>>     idx, share = [ strip(s) for s in in_str.split(",") ]
>>>     shares.append((idx, unhexlify(share)))
>>> key = Shamir.combine(shares)
>>>
```

```
>>> fi = open("enc_file.txt", "rb")
>>> nonce, tag = [ fi.read(16) for x in range(2) ]
>>> cipher = AES.new(key, AES.MODE_EAX, nonce)
>>> try:
>>>     result = cipher.decrypt(fi.read())
>>>     cipher.verify(tag)
>>>     with open("clear_file2.txt", "wb") as fo:
>>>         fo.write(result)
>>> except ValueError:
>>>     print "The shares were incorrect"
```

Attention: Reconstruction does not guarantee that the result is authentic. In particular, a malicious participant in the scheme has the ability to force an algebric transformation on the result by manipulating her share.

It is important to use the scheme in combination with an authentication mechanism (the EAX cipher mode in the example).

static combine(*shares*)

Recombine a secret, if enough shares are presented.

Parameters **shares** (*tuples*) – At least *k* tuples, each containin the index (an integer) and the share (a byte string, 16 bytes long) that were assigned to a participant.

Returns The original secret, as a byte string (16 bytes long).

static split(*k*, *n*, *secret*)

Split a secret into *n* shares.

The secret can be reconstructed later when *k* shares out of the original *n* are recombined. Each share must be kept confidential to the person it was assigned to.

Each share is associated to an index (starting from 1), which must be presented when the secret is recombined.

Parameters

- **k** (*integer*) – The number of shares that must be present in order to reconstruct the secret.
- **n** (*integer*) – The total number of shares to create (larger than *k*).
- **secret** (*byte string*) – The 16 byte string (e.g. the AES128 key) to split.

Returns *n* tuples, each containing the unique index (an integer) and the share (a byte string, 16 bytes long) meant for a participant.

- *Key Derivation Functions*
- *Secret Sharing Schemes*

Crypto.IO package

Modules for reading and writing cryptographic data.

- *PEM*
- *PKCS#8*

PEM

Set of functions for encapsulating data according to the PEM format.

PEM (Privacy Enhanced Mail) was an IETF standard for securing emails via a Public Key Infrastructure. It is specified in RFC 1421-1424.

Even though it has been abandoned, the simple message encapsulation it defined is still widely used today for encoding *binary* cryptographic objects like keys and certificates into text.

`Crypto.IO.PEM.`**`encode`**(*data*, *marker*, *passphrase=None*, *randfunc=None*)
Encode a piece of binary data into PEM format.

Parameters

- **`data`** (*`byte string`*) – The piece of binary data to encode.
- **`marker`** (*`string`*) – The marker for the PEM block (e.g. "PUBLIC KEY"). Note that there is no official master list for all allowed markers. Still, you can refer to the OpenSSL source code.
- **`passphrase`** (*`byte string`*) – If given, the PEM block will be encrypted. The key is derived from the passphrase.
- **`randfunc`** (*`callable`*) – Random number generation function; it accepts an integer N and returns a byte string of random data, N bytes long. If not given, a new one is instantiated.

Returns The PEM block, as a string.

`Crypto.IO.PEM.`**`decode`**(*pem_data*, *passphrase=None*)
Decode a PEM block into binary.

Parameters

- **`pem_data`** (*`string`*) – The PEM block.
- **`passphrase`** (*`byte string`*) – If given and the PEM block is encrypted, the key will be derived from the passphrase.

Returns A tuple with the binary data, the marker string, and a boolean to indicate if decryption was performed.

Raises `ValueError` – if decoding fails, if the PEM file is encrypted and no passphrase has been provided or if the passphrase is incorrect.

PKCS#8

PKCS#8 is a standard for storing and transferring private key information. The wrapped key can either be clear or encrypted.

All encryption algorithms are based on passphrase-based key derivation. The following mechanisms are fully supported:

- *PBKDF2WithHMAC-SHA1AndAES128-CBC*
- *PBKDF2WithHMAC-SHA1AndAES192-CBC*
- *PBKDF2WithHMAC-SHA1AndAES256-CBC*
- *PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC*
- *scryptAndAES128-CBC*

- *scryptAndAES192-CBC*
- *scryptAndAES256-CBC*

The following mechanisms are only supported for importing keys. They are much weaker than the ones listed above, and they are provided for backward compatibility only:

- *pbeWithMD5AndRC2-CBC*
- *pbeWithMD5AndDES-CBC*
- *pbeWithSHA1AndRC2-CBC*
- *pbeWithSHA1AndDES-CBC*

`Crypto.IO.PKCS8.`**`wrap`**(*private_key*, *key_oid*, *passphrase=None*, *protection=None*, *prot_params=None*, *key_params=None*, *randfunc=None*)

Wrap a private key into a PKCS#8 blob (clear or encrypted).

Parameters

- **`private_key`** (*`byte string`*) – The private key encoded in binary form. The actual encoding is algorithm specific. In most cases, it is DER.
- **`key_oid`** (*`string`*) – The object identifier (OID) of the private key to wrap. It is a dotted string, like `1.2.840.113549.1.1.1` (for RSA keys).
- **`passphrase`** (*`bytes string or string`*) – The secret passphrase from which the wrapping key is derived. Set it only if encryption is required.
- **`protection`** (*`string`*) – The identifier of the algorithm to use for securely wrapping the key. The default value is `PBKDF2WithHMAC-SHA1AndDES-EDE3-CBC`.
- **`prot_params`** (*`dictionary`*) – Parameters for the protection algorithm.

Key	Description
iteration_count	The KDF algorithm is repeated several times to slow down brute force attacks on passwords (called *N* or CPU/memory cost in scrypt). The default value for PBKDF2 is 1000. The default value for scrypt is 16384.
salt_size	Salt is used to thwart dictionary and rainbow attacks on passwords. The default value is 8 bytes.
block_size	*(scrypt only)* Memory-cost (r). The default value is 8.
parallelization	*(scrypt only)* CPU-cost (p). The default value is 1.

- **`key_params`** (*`DER object`*) – The algorithm parameters associated to the private key. It is required for algorithms like DSA, but not for others like RSA.
- **`randfunc`** (*`callable`*) – Random number generation function; it should accept a single integer N and return a string of random data, N bytes long. If not specified, a new RNG will be instantiated from `Crypto.Random`.

Returns The PKCS#8-wrapped private key (possibly encrypted), as a byte string.

`Crypto.IO.PKCS8.`**`unwrap`**(*p8_private_key*, *passphrase=None*)

Unwrap a private key from a PKCS#8 blob (clear or encrypted).

Parameters

- **`p8_private_key`** (*`byte string`*) – The private key wrapped into a PKCS#8 blob, DER encoded.

- **passphrase** (*byte string or string*) – The passphrase to use to decrypt the blob (if it is encrypted).

Returns

A tuple containing

1. the algorithm identifier of the wrapped key (OID, dotted string)
2. the private key (byte string, DER encoded)
3. the associated parameters (byte string, DER encoded) or `None`

Raises `ValueError` – if decoding fails

`Crypto.Random` package

`Crypto.Random.`**`get_random_bytes`**(*N*)
Return a random byte string of length *N*.

`Crypto.Random.random` module

`Crypto.Random.random.`**`getrandbits`**(*N*)
Return a random integer, at most *N* bits long.

`Crypto.Random.random.`**`randrange`**([*start*], *stop*[, *step*])
Return a random integer in the range *(start, stop, step)*. By default, *start* is 0 and *step* is 1.

`Crypto.Random.random.`**`randint`**(*a*, *b*)
Return a random integer in the range no smaller than *a* and no larger than *b*.

`Crypto.Random.random.`**`choice`**(*seq*)
Return a random element picked from the sequence *seq*.

`Crypto.Random.random.`**`shuffle`**(*seq*)
Randomly shuffle the sequence *seq* in-place.

`Crypto.Random.random.`**`sample`**(*population*, *k*)
Randomly chooses *k* distinct elements from the list *population*.

`Crypto.Util` package

Useful modules that don't belong in any other package.

`Crypto.Util.asn1` module

This module provides minimal support for encoding and decoding ASN.1 DER objects.

class `Crypto.Util.asn1.`**`DerObject`**(*asn1Id=None*, *payload=''*, *implicit=None*, *constructed=False*, *explicit=None*)
Base class for defining a single DER object.

This class should never be directly instantiated.

`decode`(*der_encoded*)
Decode a complete DER element, and re-initializes this object with it.

Parameters **der_encoded** (*byte string*) – A complete DER element.

Raises ValueError – in case of parsing errors.

encode()
Return this DER element, fully encoded as a binary byte string.

class Crypto.Util.asn1.**DerInteger**(*value=0*, *implicit=None*, *explicit=None*)
Class to model a DER INTEGER.

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerInteger
>>> from binascii import hexlify, unhexlify
>>> int_der = DerInteger(9)
>>> print hexlify(int_der.encode())
```

which will show 020109, the DER encoding of 9.

And for decoding:

```
>>> s = unhexlify(b'020109')
>>> try:
>>>   int_der = DerInteger()
>>>   int_der.decode(s)
>>>   print int_der.value
>>> except ValueError:
>>>   print "Not a valid DER INTEGER"
```

the output will be 9.

Variables **value** (*integer*) – The integer value

decode(*der_encoded*)
Decode a complete DER INTEGER DER, and re-initializes this object with it.

Parameters **der_encoded** (*byte string*) – A complete INTEGER DER element.

Raises ValueError – in case of parsing errors.

encode()
Return the DER INTEGER, fully encoded as a binary string.

class Crypto.Util.asn1.**DerOctetString**(*value=''*, *implicit=None*)
Class to model a DER OCTET STRING.

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerOctetString
>>> from binascii import hexlify, unhexlify
>>> os_der = DerOctetString(b'\xaa')
>>> os_der.payload += b'\xbb'
>>> print hexlify(os_der.encode())
```

which will show 0402aabb, the DER encoding for the byte string b'\xAA\xBB'.

For decoding:

```
>>> s = unhexlify(b'0402aabb')
>>> try:
>>>   os_der = DerOctetString()
>>>   os_der.decode(s)
>>>   print hexlify(os_der.payload)
```

```
>>> except ValueError:
>>>   print "Not a valid DER OCTET STRING"
```

the output will be `aabb`.

Variables **payload** (*byte string*) – The content of the string

class `Crypto.Util.asn1.`**DerNull**
Class to model a DER NULL element.

class `Crypto.Util.asn1.`**DerSequence**(*startSeq=None*, *implicit=None*)
Class to model a DER SEQUENCE.

This object behaves like a dynamic Python sequence.

Sub-elements that are INTEGERs behave like Python integers.

Any other sub-element is a binary string encoded as a complete DER sub-element (TLV).

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerSequence, DerInteger
>>> from binascii import hexlify, unhexlify
>>> obj_der = unhexlify('070102')
>>> seq_der = DerSequence([4])
>>> seq_der.append(9)
>>> seq_der.append(obj_der.encode())
>>> print hexlify(seq_der.encode())
```

which will show `300902010402010907 0102`, the DER encoding of the sequence containing `4`, `9`, and the object with payload `02`.

For decoding:

```
>>> s = unhexlify(b'300902010402010907 0102')
>>> try:
>>>   seq_der = DerSequence()
>>>   seq_der.decode(s)
>>>   print len(seq_der)
>>>   print seq_der[0]
>>>   print seq_der[:]
>>> except ValueError:
>>>   print "Not a valid DER SEQUENCE"
```

the output will be:

```
3
4
[4, 9, b'{}']
```

decode(*der_encoded*, *nr_elements=None*, *only_ints_expected=False*)
Decode a complete DER SEQUENCE, and re-initializes this object with it.

Parameters

- **der_encoded** (*byte string*) – A complete SEQUENCE DER element.
- **nr_elements** (*None or integer or list of integers*) – The number of members the SEQUENCE can have
- **only_ints_expected** (*boolean*) – Whether the SEQUENCE is expected to contain only integers.

Raises `ValueError` – in case of parsing errors.

DER INTEGERs are decoded into Python integers. Any other DER element is not decoded. Its validity is not checked.

encode()
Return this DER SEQUENCE, fully encoded as a binary string.

Raises `ValueError` – if some elements in the sequence are neither integers nor byte strings.

hasInts(*only_non_negative=True*)
Return the number of items in this sequence that are integers.

Parameters **only_non_negative** (*boolean*) – If `True`, negative integers are not counted in.

hasOnlyInts(*only_non_negative=True*)
Return `True` if all items in this sequence are integers or non-negative integers.

This function returns False is the sequence is empty, or at least one member is not an integer.

Parameters **only_non_negative** (*boolean*) – If `True`, the presence of negative integers causes the method to return `False`.

class `Crypto.Util.asn1.`**DerObjectId**(*value=''*, *implicit=None*, *explicit=None*)
Class to model a DER OBJECT ID.

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerObjectId
>>> from binascii import hexlify, unhexlify
>>> oid_der = DerObjectId("1.2")
>>> oid_der.value += ".840.113549.1.1.1"
>>> print hexlify(oid_der.encode())
```

which will show `06092a864886f70d010101`, the DER encoding for the RSA Object Identifier `1.2.840.113549.1.1.1`.

For decoding:

```
>>> s = unhexlify(b'06092a864886f70d010101')
>>> try:
>>>   oid_der = DerObjectId()
>>>   oid_der.decode(s)
>>>   print oid_der.value
>>> except ValueError:
>>>   print "Not a valid DER OBJECT ID"
```

the output will be `1.2.840.113549.1.1.1`.

Variables **value** (*string*) – The Object ID (OID), a dot separated list of integers

decode(*der_encoded*)
Decode a complete DER OBJECT ID, and re-initializes this object with it.

Parameters **der_encoded** (*byte string*) – A complete DER OBJECT ID.

Raises `ValueError` – in case of parsing errors.

encode()
Return the DER OBJECT ID, fully encoded as a binary string.

class `Crypto.Util.asn1.`**DerBitString**(*value=''*, *implicit=None*, *explicit=None*)
Class to model a DER BIT STRING.

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerBitString
>>> from binascii import hexlify, unhexlify
>>> bs_der = DerBitString(b'\xaa')
>>> bs_der.value += b'\xbb'
>>> print hexlify(bs_der.encode())
```

which will show `040300aabb`, the DER encoding for the bit string `b'\xAA\xBB'`.

For decoding:

```
>>> s = unhexlify(b'040300aabb')
>>> try:
>>>   bs_der = DerBitString()
>>>   bs_der.decode(s)
>>>   print hexlify(bs_der.value)
>>> except ValueError:
>>>   print "Not a valid DER BIT STRING"
```

the output will be `aabb`.

Variables **value** (*byte string*) – The content of the string

decode(*der_encoded*)
Decode a complete DER BIT STRING, and re-initializes this object with it.

Parameters **der_encoded** (*byte string*) – a complete DER BIT STRING.

Raises `ValueError` – in case of parsing errors.

encode()
Return the DER BIT STRING, fully encoded as a binary string.

class `Crypto.Util.asn1.`**DerSetOf**(*startSet=None*, *implicit=None*)
Class to model a DER SET OF.

An example of encoding is:

```
>>> from Crypto.Util.asn1 import DerBitString
>>> from binascii import hexlify, unhexlify
>>> so_der = DerSetOf([4,5])
>>> so_der.add(6)
>>> print hexlify(so_der.encode())
```

which will show `3109020104020105020106`, the DER encoding of a SET OF with items 4,5, and 6.

For decoding:

```
>>> s = unhexlify(b'3109020104020105020106')
>>> try:
>>>   so_der = DerSetOf()
>>>   so_der.decode(s)
>>>   print [x for x in so_der]
>>> except ValueError:
>>>   print "Not a valid DER SET OF"
```

the output will be `[4, 5, 6]`.

add(*elem*)
Add an element to the set.

Parameters **elem** (*byte string or integer*) – An element of the same type of objects already in the set. It can be an integer or a DER encoded object.

decode(*der_encoded*)
Decode a complete SET OF DER element, and re-initializes this object with it.

DER INTEGERs are decoded into Python integers. Any other DER element is left undecoded; its validity is not checked.

Parameters **der_encoded** (*byte string*) – a complete DER BIT SET OF.

Raises ValueError – in case of parsing errors.

encode()
Return this SET OF DER element, fully encoded as a binary string.

Crypto.Util.Padding module

This module provides minimal support for adding and removing standard padding from data.

Crypto.Util.Padding.**pad**(*data_to_pad*, *block_size*, *style='pkcs7'*)
Apply standard padding.

Parameters

- **data_to_pad** (*byte string*) – The data that needs to be padded.
- **block_size** (*integer*) – The block boundary to use for padding. The output length is guaranteed to be a multiple of block_size.
- **style** (*string*) – Padding algorithm. It can be *'pkcs7'* (default), *'iso7816'* or *'x923'*.

Returns the original data with the appropriate padding added at the end.

Return type byte string

Crypto.Util.Padding.**unpad**(*padded_data*, *block_size*, *style='pkcs7'*)
Remove standard padding.

Parameters

- **padded_data** (*byte string*) – A piece of data with padding that needs to be stripped.
- **block_size** (*integer*) – The block boundary to use for padding. The input length must be a multiple of block_size.
- **style** (*string*) – Padding algorithm. It can be *'pkcs7'* (default), *'iso7816'* or *'x923'*.

Returns data without padding.

Return type byte string

Raises ValueError – if the padding is incorrect.

Crypto.Util.RFC1751 module

Crypto.Util.RFC1751.**english_to_key**(*s*)
Transform a string into a corresponding key.

Example:

```
>>> from Crypto.Util.RFC1751 import english_to_key
>>> english_to_key('RAM LOIS GOAD CREW CARE HIT')
b'66666666'
```

Parameters **s** (*string*) – the string with the words separated by whitespace; the number of words must be a multiple of 6.

Returns A byte string.

Crypto.Util.RFC1751.**key_to_english**(*key*)

Transform an arbitrary key into a string containing English words.

Example:

```
>>> from Crypto.Util.RFC1751 import key_to_english
>>> key_to_english(b'66666666')
'RAM LOIS GOAD CREW CARE HIT'
```

Parameters **key** (*byte string*) – The key to convert. Its length must be a multiple of 8.

Returns A string of English words.

Crypto.Util.strxor module

Fast XOR for byte strings.

Crypto.Util.strxor.**strxor**(*term1*, *term2*)

XOR of two byte strings. They must have equal length.

Returns A new byte string, `term1` xored with `term2`.

Crypto.Util.strxor.**strxor_c**(*term*, *c*)

XOR of a byte string with a repeated sequence of characters.

Returns A new byte string, `term` with all its bytes xored with `c`.

Crypto.Util.Counter module

Fast counter functions for CTR cipher modes.

CTR is a chaining mode for symmetric block encryption or decryption. Messages are divideded into blocks, and the cipher operation takes place on each block using the secret key and a unique *counter block*.

The most straightforward way to fulfil the uniqueness property is to start with an initial, random *counter block* value, and increment it as the next block is processed.

The block ciphers from `Crypto.Cipher` (when configured in *MODE_CTR* mode) invoke a callable object (the *counter* parameter) to get the next *counter block*. Unfortunately, the Python calling protocol leads to major performance degradations.

The counter functions instantiated by this module will be invoked directly by the ciphers in `Crypto.Cipher`. The fact that the Python layer is bypassed lead to more efficient (and faster) execution of CTR cipher modes.

An example of usage is the following:

```
>>> from Crypto.Cipher import AES
>>> from Crypto.Util import Counter
>>> from Crypto import Random
>>>
>>> nonce = Random.get_random_bytes(8)
>>> ctr = Counter.new(64, nonce)
>>> key = b'AES-128 symm key'
>>> plaintext = b'X'*1000000
>>> cipher = AES.new(key, AES.MODE_CTR, counter=ctr)
>>> ciphertext = cipher.encrypt(plaintext)
```

Crypto.Util.Counter.**new**(*nbits*, *prefix=''*, *suffix=''*, *initial_value=1*, *little_endian=False*, *allow_wraparound=False*)

Create a stateful counter block function suitable for CTR encryption modes.

Each call to the function returns the next counter block. Each counter block is made up by three parts:

prefix	counter value	postfix

The counter value is incremented by 1 at each call.

Parameters

- **nbits** (*integer*) – Length of the desired counter value, in bits. It must be a multiple of 8.
- **prefix** (*byte string*) – The constant prefix of the counter block. By default, no prefix is used.
- **suffix** (*byte string*) – The constant postfix of the counter block. By default, no suffix is used.
- **initial_value** (*integer*) – The initial value of the counter. Default value is 1.
- **little_endian** (*boolean*) – If `True`, the counter number will be encoded in little endian format. If `False` (default), in big endian format.
- **allow_wraparound** (*boolean*) – This parameter is ignored.

Returns An object that can be passed with the `counter` parameter to a CTR mode cipher.

It must hold that *len(prefix) + nbits//8 + len(suffix)* matches the block size of the underlying block cipher.

Crypto.Util.number module

Crypto.Util.number.**GCD**(*x*, *y*)

Greatest Common Denominator of `x` and `y`.

Crypto.Util.number.**bytes_to_long**(*s*)

Convert a byte string to a long integer (big endian).

In Python 3.2+, use the native method instead:

```
>>> int.from_bytes(s, 'big')
```

For instance:

```
>>> int.from_bytes(b'\P', 'big')
80
```

This is (essentially) the inverse of *long_to_bytes()*.

Crypto.Util.number.**getPrime**(*N*, *randfunc=None*)
Return a random N-bit prime number.

If randfunc is omitted, then `Random.get_random_bytes()` is used.

Crypto.Util.number.**getRandomInteger**(*N*, *randfunc=None*)
Return a random number at most N bits long.

If `randfunc` is omitted, then `Random.get_random_bytes()` is used.

Deprecated since version 3.0: This function is for internal use only and may be renamed or removed in the future. Use *Crypto.Random.random.getrandbits()* instead.

Crypto.Util.number.**getRandomNBitInteger**(*N*, *randfunc=None*)
Return a random number with exactly N-bits, i.e. a random number between 2**(N-1) and (2**N)-1.

If `randfunc` is omitted, then `Random.get_random_bytes()` is used.

Deprecated since version 3.0: This function is for internal use only and may be renamed or removed in the future.

Crypto.Util.number.**getRandomRange**(*a*, *b*, *randfunc=None*)
Return a random number *n* so that *a* <= *n* < *b*.

If `randfunc` is omitted, then `Random.get_random_bytes()` is used.

Deprecated since version 3.0: This function is for internal use only and may be renamed or removed in the future. Use *Crypto.Random.random.randrange()* instead.

Crypto.Util.number.**getStrongPrime**(*N*, *e=0*, *false_positive_prob=1e-06*, *randfunc=None*)
Return a random strong *N*-bit prime number. In this context, *p* is a strong prime if *p-1* and *p+1* have at least one large prime factor.

Parameters

- **N** (*integer*) – the exact length of the strong prime. It must be a multiple of 128 and > 512.
- **e** (*integer*) – if provided, the returned prime (minus 1) will be coprime to *e* and thus suitable for RSA where *e* is the public exponent.
- **false_positive_prob** (*float*) – The statistical probability for the result not to be actually a prime. It defaults to 10^{-6}. Note that the real probability of a false-positive is far less. This is just the mathematically provable limit.
- **randfunc** (*callable*) – A function that takes a parameter *N* and that returns a random byte string of such length. If omitted, *Crypto.Random.get_random_bytes()* is used.

Returns The new strong prime.

Deprecated since version 3.0: This function is for internal use only and may be renamed or removed in the future.

Crypto.Util.number.**inverse**(*u*, *v*)
The inverse of `u` *mod* `v`.

Crypto.Util.number.**isPrime**(*N*, *false_positive_prob=1e-06*, *randfunc=None*)
Test if a number *N* is a prime.

Parameters

- **false_positive_prob** (*float*) – The statistical probability for the result not to be actually a prime. It defaults to 10^{-6}. Note that the real probability of a false-positive is far less. This is just the mathematically provable limit.

- **randfunc** (*callable*) – A function that takes a parameter *N* and that returns a random byte string of such length. If omitted, *Crypto.Random.get_random_bytes()* is used.

Returns *True* is the input is indeed prime.

Crypto.Util.number.**long_to_bytes**(*n*, *blocksize=0*)

Convert an integer to a byte string.

In Python 3.2+, use the native method instead:

```
>>> n.to_bytes(blocksize, 'big')
```

For instance:

```
>>> n = 80
>>> n.to_bytes(2, 'big')
b'\P'
```

If the optional `blocksize` is provided and greater than zero, the byte string is padded with binary zeros (on the front) so that the total length of the output is a multiple of blocksize.

If `blocksize` is zero or not provided, the byte string will be of minimal length.

Crypto.Util.number.**size**(*N*)

Returns the size of the number N in bits.

All cryptographic functionalities are organized in sub-packages; each sub-package is dedicated to solving a specific class of problems.

Package	Description
Crypto.Cipher	Modules for protecting **confidentiality** that is, for encrypting and decrypting data (example: AES).
Crypto.Signature	Modules for assuring **authenticity**, that is, for creating and verifying digital signatures of messages (example: PKCS#1 v1.5).
Crypto.Hash	Modules for creating cryptographic **digests** (example: SHA-256).
Crypto.PublicKey	Modules for generating, exporting or importing *public keys* (example: RSA or ECC).
Crypto.Protocol	Modules for faciliting secure communications between parties, in most cases by leveraging cryptograpic primitives from other modules (example: Shamir's Secret Sharing scheme).
Crypto.IO	Modules for dealing with encodings commonly used for cryptographic data (example: PEM).
Crypto.Random	Modules for generating random data.
Crypto.Util	General purpose routines (example: XOR for byte strings).

In certain cases, there is some overlap between these categories. For instance, **authenticity** is also provided by *Message Authentication Codes*, and some can be built using digests, so they are included in the `Crypto.Hash` package (example: HMAC). Also, cryptographers have over time realized that encryption without **authentication** is often of limited value so recent ciphers found in the `Crypto.Cipher` package embed it (example: GCM).

PyCryptodome strives to maintain strong backward compatibility with the old *PyCrypto*'s API (except for those few cases where that is harmful to security) so a few modules don't appear where they should (example: the ASN.1 module is under `Crypto.Util` as opposed to `Crypto.IO`).

CHAPTER 6

Examples

Encrypt data with AES

The following code generates a new AES128 key and encrypts a piece of data into a file. We use the EAX mode because it allows the receiver to detect any unauthorized modification (similarly, we could have used other authenticated encryption modes like GCM, CCM or SIV).

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)
cipher = AES.new(key, AES.MODE_EAX)
ciphertext, tag = cipher.encrypt_and_digest(data)

file_out = open("encrypted.bin", "wb")
[ file_out.write(x) for x in (cipher.nonce, tag, ciphertext) ]
```

At the other end, the receiver can securely load the piece of data back (if they know the key!). Note that the code generates a `ValueError` exception when tampering is detected.

```
from Crypto.Cipher import AES

file_in = open("encrypted.bin", "rb")
nonce, tag, ciphertext = [ file_in.read(x) for x in (16, 16, -1) ]

# let's assume that the key is somehow available again
cipher = AES.new(key, AES.MODE_EAX, nonce)
data = cipher.decrypt_and_verify(ciphertext, tag)
```

Generate an RSA key

The following code generates a new RSA key pair (secret) and saves it into a file, protected by a password. We use the scrypt key derivation function to thwart dictionary attacks. At the end, the code prints our the RSA public key in ASCII/PEM format:

```
from Crypto.PublicKey import RSA

secret_code = "Unguessable"
key = RSA.generate(2048)
encrypted_key = key.exportKey(passphrase=secret_code, pkcs=8,
                              protection="scryptAndAES128-CBC")

file_out = open("rsa_key.bin", "wb")
file_out.write(encrypted_key)

print key.publickey().exportKey()
```

The following code reads the private RSA key back in, and then prints again the public key:

```
from Crypto.PublicKey import RSA

secret_code = "Unguessable"
encoded_key = open("rsa_key.bin", "rb").read()
key = RSA.import_key(encoded_key, passphrase=secret_code)

print key.publickey().exportKey()
```

Encrypt data with RSA

The following code encrypts a piece of data for a receiver we have the RSA public key of. The RSA public key is stored in a file called `receiver.pem`.

Since we want to be able to encrypt an arbitrary amount of data, we use a hybrid encryption scheme. We use RSA with PKCS#1 OAEP for asymmetric encryption of an AES session key. The session key can then be used to encrypt all the actual data.

As in the first example, we use the EAX mode to allow detection of unauthorized modifications.

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP

file_out = open("encrypted_data.bin", "wb")

recipient_key = RSA.import_key(open("receiver.pem").read())
session_key = get_random_bytes(16)

# Encrypt the session key with the public RSA key
cipher_rsa = PKCS1_OAEP.new(recipient_key)
file_out.write(cipher_rsa.encrypt(session_key))

# Encrypt the data with the AES session key
cipher_aes = AES.new(session_key, AES.MODE_EAX)
ciphertext, tag = cipher_aes.encrypt_and_digest(data)
[ file_out.write(x) for x in (cipher.nonce, tag, ciphertext) ]
```

The receiver has the private RSA key. They will use it to decrypt the session key first, and with that the rest of the file:

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import AES, PKCS1_OAEP

file_in = open("encrypted_data.bin", "rb")

private_key = RSA.import_key(open("private.pem").read())

enc_session_key, nonce, tag, ciphertext = \
   [ file_in.read(x) for x in (private_key.size_in_bytes(), 16, 16, -1) ]

# Decrypt the session key with the public RSA key
cipher_rsa = PKCS1_OAEP.new(private_key)
session_key = cipher_rsa.decrypt(enc_session_key)

# Decrypt the data with the AES session key
cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
data = cipher.decrypt_and_verify(ciphertext, tag)
```

CHAPTER 7

Contribute and support

- Do not be afraid to contribute with small and apparently insignificant improvements like correction to typos. Every change counts.
- Read carefully the *License* of PyCryptodome. By submitting your code, you acknowledge that you accept to release it according to the BSD 2-clause license.
- You must disclaim which parts of your code in your contribution were partially copied or derived from an existing source. Ensure that the original is licensed in a way compatible to the *BSD 2-clause license*.
- You can propose changes in any way you find most convenient. However, the preferred approach is to:
  - Clone the main repository on GitHub.
  - Create a branch and modify the code.
  - Send a pull request upstream with a meaningful description.
- Provide tests (in `Crypto.SelfTest`) along with code. If you fix a bug add a test that fails in the current version and passes with your change.
- If your change breaks backward compatibility, hightlight it and include a justification.
- Ensure that your code complies to PEP8 and PEP257.
- Ensure that your code does not use constructs or includes modules not present in Python 2.4.
- Add a short summary of the change to the file `Changelog.rst`.
- Add your name to the list of contributors in the file `AUTHORS.rst`.

The PyCryptodome mailing list is hosted on Google Groups. You can mail any comment or question to *pycryptodome@googlegroups.com*.

Bug reports can be filed on the GitHub tracker.

CHAPTER 8

Future plans

Future releases will include:

- Update *Crypto.Signature.DSS* to FIPS 186-4
- Make all hash objects non-copiable and immutable after the first digest
- Add alias ‘segment_bits’ to parameter ‘segment_size’ for CFB
- Coverage testing
- Add support for memoryview/buffer interface
- Implement AES with bitslicing
- Add unit tests for PEM I/O
- Move old ciphers into a Museum submodule
- Add more ECC curves
- Import/export of ECC keys with compressed points
- **Add algorithms:**
  - Poly1305
  - Elliptic Curves (ECIES, ECDH)
  - Camellia, GOST
  - Diffie-Hellman
  - bcrypt
  - argon2
  - SRP
- **Add more key management:**
  - Export/import of DSA domain parameters
  - JWK

- Add support for CMS/PKCS#7
- Add support for RNG backed by PKCS#11 and/or KMIP
- Add support for Format-Preserving Encryption
- Remove dependency on libtomcrypto headers
- Speed up (T)DES with a bitsliced implementation
- Add support for PCLMULQDQ in AES-GCM
- Run lint on the C code
- Add (minimal) support for PGP
- Add (minimal) support for PKIX / X.509

CHAPTER 9

Changelog

3.4.7 (26 August 2017)

New features

- API documentation is made with sphinx instead of epydoc.
- Start using `importlib` instead of `imp` where available.

Resolved issues

- GH#82. Fixed PEM header for RSA/DSA public keys.

3.4.6 (18 May 2017)

Resolved issues

- GH#65. Keccak, SHA3, SHAKE and the seek functionality for ChaCha20 were not working on big endian machines. Fixed. Thanks to Mike Gilbert.
- A few fixes in the documentation.

3.4.5 (6 February 2017)

Resolved issues

- The library can also be compiled using MinGW.

3.4.4 (1 February 2017)

Resolved issues

- Removed use of `alloca()`.
- [Security] Removed implementation of deprecated "quick check" feature of PGP block cipher mode.
- Improved the performance of `scrypt` by converting some Python to C.

3.4.3 (17 October 2016)

Resolved issues

- Undefined warning was raised with libgmp version < 5
- Forgot inclusion of `alloca.h`
- Fixed a warning about type mismatch raised by recent versions of cffi

3.4.2 (8 March 2016)

Resolved issues

- Fix renaming of package for `install` command.

3.4.1 (21 February 2016)

New features

- Added option to install the library under the `Cryptodome` package (instead of `Crypto`).

3.4 (7 February 2016)

New features

- Added `Crypto.PublicKey.ECC` module (NIST P-256 curve only), including export/import of ECC keys.
- Added support for ECDSA (FIPS 186-3 and RFC6979).
- For CBC/CFB/OFB/CTR cipher objects, `encrypt()` and `decrypt()` cannot be intermixed.
- CBC/CFB/OFB, the cipher objects have both `IV` and `iv` attributes. `new()` accepts `IV` as well as `iv` as parameter.
- For CFB/OPENPGP cipher object, `encrypt()` and `decrypt()` do not require the plaintext or ciphertext pieces to have length multiple of the CFB segment size.
- Added dedicated tests for all cipher modes, including NIST test vectors

- CTR/CCM/EAX/GCM/SIV/Salsa20/ChaCha20 objects expose the `nonce` attribute.
- For performance reasons, CCM cipher optionally accepted a pre-declaration of the length of the associated data, but never checked if the actual data passed to the cipher really matched that length. Such check is now enforced.
- CTR cipher objects accept parameter `nonce` and possibly `initial_value` in alternative to `counter` (which is deprecated).
- All `iv`/`IV` and `nonce` parameters are optional. If not provided, they will be randomly generated (exception: `nonce` for CTR mode in case of block sizes smaller than 16 bytes).
- Refactored ARC2 cipher.
- Added `Crypto.Cipher.DES3.adjust_key_parity()` function.
- Added `RSA.import_key` as an alias to the deprecated `RSA.importKey` (same for the `DSA` module).
- Added `size_in_bits()` and `size_in_bytes()` methods to `RsaKey`.

Resolved issues

- RSA key size is now returned correctly in `RsaKey.__repr__()` method (kudos to *hannesv*).
- CTR mode does not modify anymore `counter` parameter passed to `new()` method.
- CTR raises `OverflowError` instead of `ValueError` when the counter wraps around.
- PEM files with Windows newlines could not be imported.
- `Crypto.IO.PEM` and `Crypto.IO.PKCS8` used to accept empty passphrases.
- GH#6: NotImplementedError now raised for unsupported methods `sign`, `verify`, `encrypt`, `decrypt`, `blind`, `unblind` and `size` in objects `RsaKey`, `DsaKey`, `ElGamalKey`.

Breaks in compatibility

- Parameter `segment_size` cannot be 0 for the CFB mode.
- For OCB ciphers, a final call without parameters to `encrypt` must end a sequence of calls to `encrypt` with data (similarly for `decrypt`).
- Key size for `ARC2`, `ARC4` and `Blowfish` must be at least 40 bits long (still very weak).
- DES3 (Triple DES module) does not allow keys that degenerate to Single DES.
- Removed method `getRandomNumber` in `Crypto.Util.number`.
- Removed module `Crypto.pct_warnings`.
- Removed attribute `Crypto.PublicKey.RSA.algorithmIdentifier`.

3.3.1 (1 November 2015)

New features

- Opt-in for `update()` after `digest()` for SHA-3, keccak, BLAKE2 hashes

Resolved issues

- Removed unused SHA-3 and keccak test vectors, therefore significantly reducing the package from 13MB to 3MB.

Breaks in compatibility

- Removed method `copy()` from BLAKE2 hashes
- Removed ability to `update()` a BLAKE2 hash after the first call to `(hex)digest()`

3.3 (29 October 2015)

New features

- Windows wheels bundle the MPIR library
- Detection of faults occuring during secret RSA operations
- Detection of non-prime (weak) q value in DSA domain parameters
- Added original Keccak hash family (b=1600 only). In the process, simplified the C code base for SHA-3.
- Added SHAKE128 and SHAKE256 (of SHA-3 family)

Resolved issues

- GH#3: gcc 4.4.7 unhappy about double typedef

Breaks in compatibility

- Removed method `copy()` from all SHA-3 hashes
- Removed ability to `update()` a SHA-3 hash after the first call to `(hex)digest()`

3.2.1 (9 September 2015)

New features

- Windows wheels are automatically built on Appveyor

3.2 (6 September 2015)

New features

- Added hash functions BLAKE2b and BLAKE2s.
- Added stream cipher ChaCha20.

- Added OCB cipher mode.
- CMAC raises an exception whenever the message length is found to be too large and the chance of collisions not negligeable.
- New attribute `oid` for Hash objects with ASN.1 Object ID
- Added `Crypto.Signature.pss` and `Crypto.Signature.pkcs1_15`
- Added NIST test vectors (roughly 1200) for PKCS#1 v1.5 and PSS signatures.

Resolved issues

- tomcrypt_macros.h asm error #1

Breaks in compatibility

- Removed keyword `verify_x509_cert` from module method `importKey` (RSA and DSA).
- Reverted to original PyCrypto behavior of method `verify` in `PKCS1_v1_5` and `PKCS1_PSS`.

3.1 (15 March 2015)

New features

- Speed up execution of Public Key algorithms on PyPy, when backed by the Gnu Multiprecision (GMP) library.
- GMP headers and static libraries are not required anymore at the time PyCryptodome is built. Instead, the code will automatically use the GMP dynamic library (.so/.DLL) if found in the system at runtime.
- Reduced the amount of C code by almost 40% (4700 lines). Modularized and simplified all code (C and Python) related to block ciphers. Pycryptodome is now free of CPython extensions.
- Add support for CI in Windows via Appveyor.
- RSA and DSA key generation more closely follows FIPS 186-4 (though it is not 100% compliant).

Resolved issues

- None

Breaks in compatibility

- New dependency on ctypes with Python 2.4.
- The `counter` parameter of a CTR mode cipher must be generated via `Crypto.Util.Counter`. It cannot be a generic callable anymore.
- Removed the `Crypto.Random.Fortuna` package (due to lack of test vectors).
- Removed the `Crypto.Hash.new` function.
- The `allow_wraparound` parameter of `Crypto.Util.Counter` is ignored. An exception is always generated if the counter is reused.

- `DSA.generate`, `RSA.generate` and `ElGamal.generate` do not accept the `progress_func` parameter anymore.
- Removed `Crypto.PublicKey.RSA.RSAImplementation`.
- Removed `Crypto.PublicKey.DSA.DSAImplementation`.
- Removed ambiguous method `size()` from RSA, DSA and ElGamal keys.

3.0 (24 June 2014)

New features

- Initial support for PyPy.
- SHA-3 hash family based on the April 2014 draft of FIPS 202. See modules `Crypto.Hash.SHA3_224/256/384/512`. Initial Keccak patch by Fabrizio Tarizzo.
- Salsa20 stream cipher. See module `Crypto.Cipher.Salsa20`. Patch by Fabrizio Tarizzo.
- Colin Percival's `scrypt` key derivation function (`Crypto.Protocol.KDF.scrypt`).
- Proper interface to FIPS 186-3 DSA. See module `Crypto.Signature.DSS`.
- Deterministic DSA (RFC6979). Again, see `Crypto.Signature.DSS`.
- HMAC-based Extract-and-Expand key derivation function (`Crypto.Protocol.KDF.HKDF`, RFC5869).
- Shamir's Secret Sharing protocol, compatible with *ssss* (128 bits only). See module `Crypto.Protocol.SecretSharing`.
- Ability to generate a DSA key given the domain parameters.
- Ability to test installation with a simple `python -m Crypto.SelfTest`.

Resolved issues

- LP#1193521: `mpz_powm_sec()` (and Python) crashed when modulus was odd.
- Benchmarks work again (they broke when ECB stopped working if an IV was passed. Patch by Richard Mitchell.
- LP#1178485: removed some catch-all exception handlers. Patch by Richard Mitchell.
- LP#1209399: Removal of Python wrappers caused HMAC to silently produce the wrong data with SHA-2 algorithms.
- LP#1279231: remove dead code that does nothing in SHA-2 hashes. Patch by Richard Mitchell.
- LP#1327081: AESNI code accesses memory beyond buffer end.
- Stricter checks on ciphertext and plaintext size for textbook RSA (kudos to sharego).

Breaks in compatibility

- Removed support for Python < 2.4.
- Removed the following methods from all 3 public key object types (RSA, DSA, ElGamal):
  - `sign`
  - `verify`

  - `encrypt`
  - `decrypt`
  - `blind`
  - `unblind`

  Code that uses such methods is doomed anyway. It should be fixed ASAP to use the algorithms available in `Crypto.Signature` and `Crypto.Cipher`.

- The 3 public key object types (RSA, DSA, ElGamal) are now unpickable.
- Symmetric ciphers do not have a default mode anymore (used to be ECB). An expression like `AES.new(key)` will now fail. If ECB is the desired mode, one has to explicitly use `AES.new(key, AES.MODE_ECB)`.
- Unsuccessful verification of a signature will now raise an exception [reverted in 3.2].
- Removed the `Crypto.Random.OSRNG` package.
- Removed the `Crypto.Util.winrandom` module.
- Removed the `Crypto.Random.randpool` module.
- Removed the `Crypto.Cipher.XOR` module.
- Removed the `Crypto.Protocol.AllOrNothing` module.
- Removed the `Crypto.Protocol.Chaffing` module.
- Removed the parameters `disabled_shortcut` and `overflow` from `Crypto.Util.Counter.new`.

Other changes

- `Crypto.Random` stops being a userspace CSPRNG. It is now a pure wrapper over `os.urandom`.
- Added certain resistance against side-channel attacks for GHASH (GCM) and DSA.
- More test vectors for `HMAC-RIPEMD-160`.
- Update `libtomcrypt` headers and code to v1.17 (kudos to Richard Mitchell).
- RSA and DSA keys are checked for consistency as they are imported.
- Simplified build process by removing autoconf.
- Speed optimization to PBKDF2.
- Add support for MSVC.
- Replaced HMAC code with a BSD implementation. Clarified that starting from the fork, all contributions are released under the BSD license.

CHAPTER 10

License

The source code in PyCryptodome is partially in the public domain and partially released under the BSD 2-Clause license.

In either case, there are minimal if no restrictions on the redistribution, modification and usage of the software.

Public domain

All code originating from PyCrypto is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <http://unlicense.org>

BSD license

All direct contributions to PyCryptodome are released under the following license. The copyright of each piece belongs to the respective author.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

OCB license

The OCB cipher mode is patended in US. The directory Doc/ocb contains three free licenses for implementors and users. As a general statement, OCB can be freely used for software not meant for military purposes. Contact your attorney for further information.

MPIR license

When distributed as a Windows wheel, Pycryptodome bundles an unmodified, binary version of the MPIR library (https://www.mpir.org) which is licensed under the LGPLv3, a copy of which is available under Doc/mpir.

Python Module Index

C

Index

G

H

I

K

L

N

O

P

R

S

U

W