PyCryptodome Documentation

Release 3.8.1

Legrandin

Apr 04, 2019

Contents

1 PyCryptodome **3**

2 Features **5**

3 Installation **9**
- 3.1 Compiling in Linux Ubuntu . . . 9
- 3.2 Compiling in Linux Fedora . . . 10
- 3.3 Windows (from sources, Python 2.x, Python <=3.2) . . . 10
- 3.4 Windows (from sources, Python 3.3 and 3.4) . . . 11
- 3.5 Windows (from sources, Python 3.5 and newer) . . . 12
- 3.6 Documentation . . . 12
- 3.7 PGP verification . . . 13

4 Compatibility with PyCrypto **15**

5 API documentation **17**
- 5.1 `Crypto.Cipher` package . . . 17
  - 5.1.1 Introduction . . . 17
  - 5.1.2 API principles . . . 17
  - 5.1.3 Symmetric ciphers . . . 18
  - 5.1.4 Legacy ciphers . . . 34
- 5.2 `Crypto.Signature` package . . . 34
  - 5.2.1 Signing a message . . . 34
  - 5.2.2 Verifying a signature . . . 34
  - 5.2.3 Available mechanisms . . . 35
- 5.3 `Crypto.Hash` package . . . 35
  - 5.3.1 API principles . . . 35
  - 5.3.2 Attributes of hash objects . . . 37
  - 5.3.3 Modern hash algorithms . . . 37
  - 5.3.4 Extensible-Output Functions (XOF) . . . 37
  - 5.3.5 Message Authentication Code (MAC) algorithms . . . 38
  - 5.3.6 Historich hash algorithms . . . 38
- 5.4 `Crypto.PublicKey` package . . . 38
  - 5.4.1 API principles . . . 38
  - 5.4.2 Available key types . . . 39
  - 5.4.3 Obsolete key type . . . 51
- 5.5 `Crypto.Protocol` package . . . 53

5.5.1 Key Derivation Functions . . . 53
5.5.2 Secret Sharing Schemes . . . 55
5.6 `Crypto.IO` package . . . 57
5.6.1 PEM . . . 57
5.6.2 PKCS#8 . . . 58
5.7 `Crypto.Random` package . . . 59
5.7.1 `Crypto.Random.random` module . . . 59
5.8 `Crypto.Util` package . . . 60
5.8.1 `Crypto.Util.asn1` module . . . 60
5.8.2 `Crypto.Util.Padding` module . . . 65
5.8.3 `Crypto.Util.RFC1751` module . . . 66
5.8.4 `Crypto.Util.strxor` module . . . 66
5.8.5 `Crypto.Util.Counter` module . . . 67
5.8.6 `Crypto.Util.number` module . . . 68

6 Examples **73**
6.1 Encrypt data with AES . . . 73
6.2 Generate an RSA key . . . 74
6.3 Generate public key and private key . . . 74
6.4 Encrypt data with RSA . . . 74

7 Frequently Asked Questions **77**
7.1 Is CTR cipher mode compatible with Java? . . . 77
7.2 Are RSASSA-PSS signatures compatible with Java or OpenSSL? . . . 77
7.3 Why do I get the error `No module named Crypto` on Windows? . . . 78

8 Contribute and support **79**

9 Future plans **81**

10 Changelog **83**
10.1 3.8.1 (4 April 2019) . . . 83
10.1.1 New features . . . 83
10.1.2 Resolved issues . . . 83
10.2 3.8.0 (23 March 2019) . . . 83
10.2.1 New features . . . 83
10.2.2 Resolved issues . . . 84
10.2.3 Breaks in compatibility . . . 84
10.3 3.7.3 (19 January 2019) . . . 84
10.3.1 Resolved issues . . . 84
10.4 3.7.2 (26 November 2018) . . . 84
10.4.1 Resolved issues . . . 84
10.5 3.7.1 (25 November 2018) . . . 84
10.5.1 New features . . . 84
10.5.2 Resolved issues . . . 84
10.6 3.7.0 (27 October 2018) . . . 85
10.6.1 New features . . . 85
10.6.2 Resolved issues . . . 85
10.6.3 Breaks in compatibility . . . 85
10.7 3.6.6 (17 August 2018) . . . 85
10.7.1 Resolved issues . . . 85
10.8 3.6.5 (12 August 2018) . . . 85
10.8.1 Resolved issues . . . 85
10.9 3.6.4 (10 July 2018) . . . 86
10.9.1 New features . . . 86

10.9.2 Resolved issues . . . 86
10.10 3.6.3 (21 June 2018) . . . 86
10.10.1 Resolved issues . . . 86
10.11 3.6.2 (19 June 2018) . . . 86
10.11.1 New features . . . 86
10.11.2 Resolved issues . . . 86
10.11.3 Breaks in compatibility . . . 86
10.12 3.6.1 (15 April 2018) . . . 87
10.12.1 New features . . . 87
10.12.2 Resolved issues . . . 87
10.13 3.6.0 (8 April 2018) . . . 87
10.13.1 New features . . . 87
10.13.2 Resolved issues . . . 87
10.14 3.5.1 (8 March 2018) . . . 87
10.14.1 Resolved issues . . . 87
10.15 3.5.0 (7 March 2018) . . . 87
10.15.1 New features . . . 87
10.15.2 Resolved issues . . . 88
10.15.3 Breaks in compatibility . . . 88
10.16 3.4.12 (5 February 2018) . . . 88
10.16.1 Resolved issues . . . 88
10.17 3.4.11 (5 February 2018) . . . 88
10.17.1 Resolved issues . . . 88
10.18 3.4.10 (2 February 2018) . . . 88
10.18.1 Resolved issues . . . 88
10.19 3.4.9 (1 February 2018) . . . 88
10.19.1 New features . . . 88
10.19.2 Resolved issues . . . 89
10.20 3.4.8 (27 January 2018) . . . 89
10.20.1 New features . . . 89
10.20.2 Resolved issues . . . 89
10.21 3.4.7 (26 August 2017) . . . 89
10.21.1 New features . . . 89
10.21.2 Resolved issues . . . 89
10.22 3.4.6 (18 May 2017) . . . 89
10.22.1 Resolved issues . . . 89
10.23 3.4.5 (6 February 2017) . . . 90
10.23.1 Resolved issues . . . 90
10.24 3.4.4 (1 February 2017) . . . 90
10.24.1 Resolved issues . . . 90
10.25 3.4.3 (17 October 2016) . . . 90
10.25.1 Resolved issues . . . 90
10.26 3.4.2 (8 March 2016) . . . 90
10.26.1 Resolved issues . . . 90
10.27 3.4.1 (21 February 2016) . . . 90
10.27.1 New features . . . 90
10.28 3.4 (7 February 2016) . . . 90
10.28.1 New features . . . 90
10.28.2 Resolved issues . . . 91
10.28.3 Breaks in compatibility . . . 91
10.29 3.3.1 (1 November 2015) . . . 92
10.29.1 New features . . . 92
10.29.2 Resolved issues . . . 92
10.29.3 Breaks in compatibility . . . 92

10.30 3.3 (29 October 2015) . . . 92
10.30.1 New features . . . 92
10.30.2 Resolved issues . . . 92
10.30.3 Breaks in compatibility . . . 92
10.31 3.2.1 (9 September 2015) . . . 92
10.31.1 New features . . . 92
10.32 3.2 (6 September 2015) . . . 93
10.32.1 New features . . . 93
10.32.2 Resolved issues . . . 93
10.32.3 Breaks in compatibility . . . 93
10.33 3.1 (15 March 2015) . . . 93
10.33.1 New features . . . 93
10.33.2 Resolved issues . . . 93
10.33.3 Breaks in compatibility . . . 94
10.34 3.0 (24 June 2014) . . . 94
10.34.1 New features . . . 94
10.34.2 Resolved issues . . . 94
10.34.3 Breaks in compatibility . . . 95
10.34.4 Other changes . . . 95

11 License **97**
11.1 Public domain . . . 97
11.2 BSD license . . . 97
11.3 OCB license . . . 98
11.4 Apache 2.0 license (Wycheproof) . . . 98

Python Module Index **103**

CHAPTER 1

PyCryptodome

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

It supports Python 2.6 and 2.7, Python 3.4 and newer, and PyPy.

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

1. **an almost 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 in Unix, you should install GMP in your system.

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, P-384 and P-521 curves 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), truncated SHA-512 and BLAKE2 hash algorithms
- Salsa20 and ChaCha20 stream ciphers
- Poly1305 MAC
- ChaCha20-Poly1305 authenticated cipher
- 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.

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 (legacy)
  - CAST-128 (legacy)
  - RC2 (legacy)
- Traditional modes of operations for symmetric ciphers:
  - ECB
  - CBC
  - CFB
  - OFB
  - CTR
  - OpenPGP (a variant of CFB, RFC4880)
- Authenticated Encryption:
  - CCM (AES only)
  - EAX
  - GCM (AES only)
  - SIV (AES only)

  - OCB (AES only)
  - ChaCha20-Poly1305
- Stream ciphers:
  - Salsa20
  - ChaCha20
  - RC4 (legacy)
- Cryptographic hashes:
  - SHA-1
  - SHA-2 hashes (224, 256, 384, 512, 512/224, 512/256)
  - SHA-3 hashes (224, 256, 384, 512) and XOFs (SHAKE128, SHAKE256)
  - Keccak (original submission to SHA-3)
  - BLAKE2b and BLAKE2s
  - RIPE-MD160 (legacy)
  - MD5 (legacy)
- Message Authentication Codes (MAC):
  - HMAC
  - CMAC
  - Poly1305
- Asymmetric key generation:
  - RSA
  - ECC (NIST P-256, P-384 and P-521 curve only)
  - DSA
  - ElGamal (legacy)
- 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:
    - PBKDF2
    - scrypt
    - HKDF
    - PBKDF1 (legacy)
- 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 to be in. PyCryptodome can be used as:

1. **an almost 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.

3.1 Compiling in Linux Ubuntu

Note: If you want to install under the `Crypto` package, replace below `pycryptodomex` with `pycryptodome`.

For Python 2.x:

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

For Python 3.x:

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

For PyPy:

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

3.2 Compiling in Linux Fedora

Note: If you want to install under the `Crypto` package, replace below `pycryptodomex` with `pycryptodome`.

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
```

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

Note: If you want to install under the `Crypto` package, replace below `pycryptodomex` with `pycryptodome`.

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]** Install Virtual Clone Drive.

3. **[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.

4. 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.

5. Compile and install PyCryptodome:

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

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

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

3.4 Windows (from sources, Python 3.3 and 3.4)

Note: If you want to install under the `Crypto` package, replace below `pycryptodomex` with `pycryptodome`.

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]** Install Virtual Clone Drive.

3. **[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.

4. 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.

5. Compile and install PyCryptodome:

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

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

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

3.5 Windows (from sources, Python 3.5 and newer)

Note: If you want to install under the `Crypto` package, replace below `pycryptodomex` with `pycryptodome`.

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. Compile and install PyCryptodome:

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

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

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

3.6 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 and use:

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

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

3.7 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/tSZ7ZuyLSZGLKl6hoNIB13HCn4dnjNBeaXCWX
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+uQboLioEvl9Ob4j5Irs/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 `export_key()` 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 2.6 is dropped.

CHAPTER 5

API documentation

5.1 `Crypto.Cipher` package

5.1.1 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.

5.1.2 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 data, you call the `encrypt()` method of the cipher object with the plaintext. The method returns the piece of ciphertext. Alternatively, with the `output` parameter you can specify a pre-allocated buffer for the result.

  For most algorithms, you may call `encrypt()` multiple times (i.e. once for each piece of plaintext).

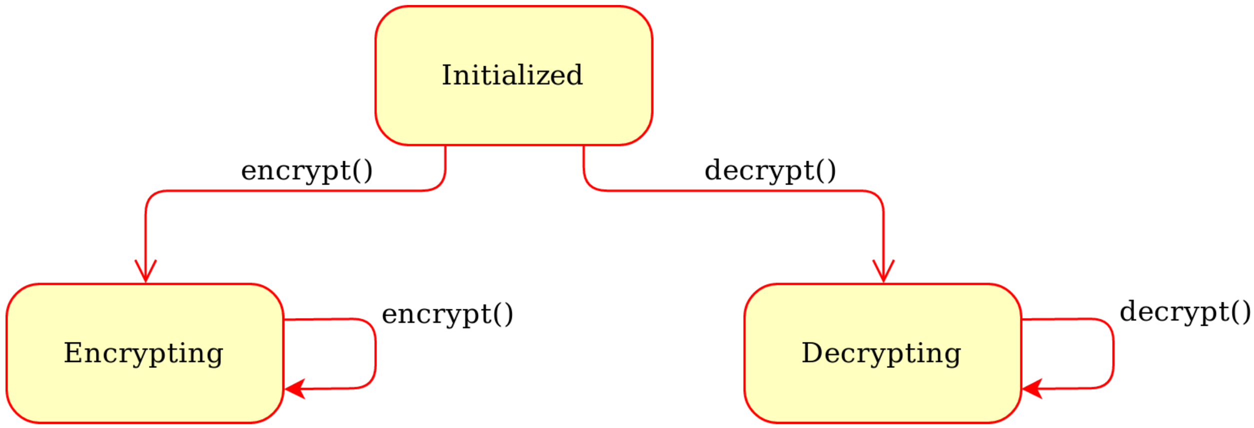


Fig. 5.1: Generic state diagram for a cipher object

- For decrypting data, you call the `decrypt()` method of the cipher object with the ciphertext. The method returns the piece of plaintext. The `output` parameter can be passed here too.

  For most algorithms, you may call `decrypt()` multiple times (i.e. once for each piece of ciphertext).

Note: Plaintexts and ciphertexts (input/output) can only be `bytes`, `bytearray` or `memoryview`. In Python 3, you cannot pass strings. In Python 2, you cannot pass Unicode strings.

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
```

5.1.3 Symmetric ciphers

There are two types of symmetric ciphers:

- **Stream ciphers**: the most natural kind of ciphers: they encrypt data one byte at a time. See chacha20 and 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 128 bits (16 bytes).

  In general, a block cipher is mostly useful only together with a *mode of operation*, which allows one to encrypt a variable amount of data. Some modes (like CTR) effectively turn a block cipher into a stream cipher.

The widespread consensus is that ciphers that provide only confidentiality, without any form of authentication, are undesireable. Instead, primitives have been defined to integrate symmetric encryption and authentication (MAC). For instance:

- *Modern modes of operation* for block ciphers (like GCM).
- Stream ciphers paired with a MAC function, like chacha20_poly1305.

Classic modes of operation for symmetric block ciphers

A block cipher uses a symmetric key to encrypt data of fixed and very short length (the *block size*), such as 16 bytes for AES. In order to cope with data of arbitrary length, the cipher must be combined with a *mode of operation*.

You create a cipher object with the `new()` function in the relevant module under `Crypto.Cipher`:

1. the first parameter is always the cryptographic key (a byte string)
2. the second parameter is always the constant that selects the desired mode of operation

Constants for each mode of operation are defined at the module level for each algorithm. Their name starts with `MODE_`, for instance `Crypto.Cipher.AES.MODE_CBC`. Note that not all ciphers support all modes.

For instance:

```
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CBC)
>>>
>>> # You can now use use cipher to encrypt or decrypt...
```

The state machine for a cipher configured with a classic mode is:

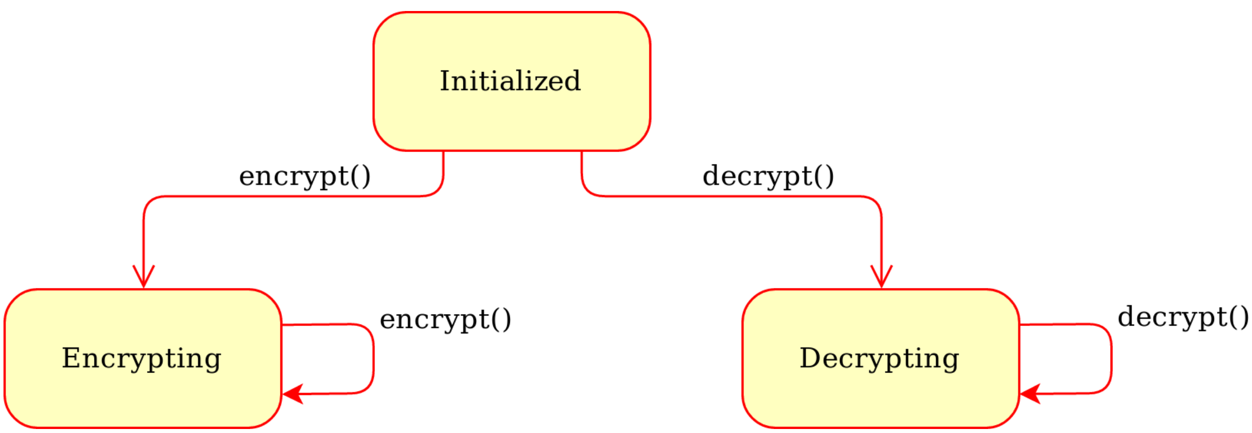


Fig. 5.2: Generic state diagram for a cipher object

What follows is a list of classic modes of operation: they all provide confidentiality but not data integrity (unlike modern AEAD modes, which are described in *another section*).

ECB mode

Electronic CodeBook. The most basic but also the weakest mode of operation. Each block of plaintext is encrypted independently of any other block.

> **Warning:** The ECB mode should not be used because it is semantically insecure. For one, it exposes correlation between blocks.

The `new()` function at the module level under `Crypto.Cipher` instantiates a new ECB cipher object for the relevant base algorithm. In the following definition, `<algorithm>` could be `AES`:

`Crypto.Cipher.<algorithm>.new(key, mode)`
Create a new ECB object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_ECB`

Returns an ECB cipher object

The method `encrypt()` (and likewise `decrypt()`) of an ECB cipher object expects data to have length multiple of the block size (e.g. 16 bytes for AES). You might need to use *`Crypto.Util.Padding`* to align the plaintext to the right boundary.

CBC mode

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

The `new()` function at the module level under `Crypto.Cipher` instantiates a new CBC cipher object for the relevant base algorithm. In the following definition, `<algorithm>` could be `AES`:

`Crypto.Cipher.<algorithm>.new(key, mode, *, iv=None)`
Create a new CBC object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_CBC`
- **iv** (*bytes*) – the *Initialization Vector*. A piece of data unpredictable to adversaries. It is as long as the block size (e.g. 16 bytes for AES). If not present, the library creates a random IV value.

Returns a CBC cipher object

The method `encrypt()` (and likewise `decrypt()`) of a CBC cipher object expects data to have length multiple of the block size (e.g. 16 bytes for AES). You might need to use *`Crypto.Util.Padding`* to align the plaintext to the right boundary.

A CBC cipher object has a read-only attribute `iv`, holding the *Initialization Vector* (*bytes*).

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import pad
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b"secret"
>>> key = get_random_bytes(16)
```

(continues on next page)

(continued from previous page)

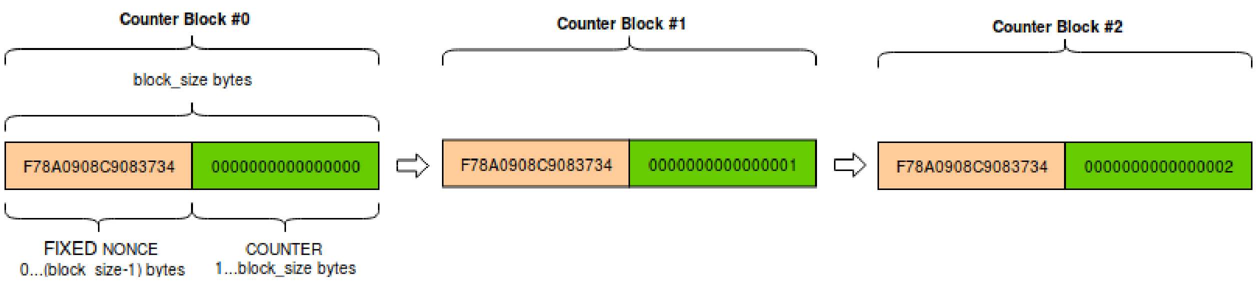
```
>>> cipher = AES.new(key, AES.MODE_CBC)
>>> ct_bytes = cipher.encrypt(pad(data, AES.block_size))
>>> iv = b64encode(cipher.iv).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> result = json.dumps({'iv':iv, 'ciphertext':ct})
>>> print(result)
'{"iv": "bWRHdzkzVDFJbWNBY0EwSmQ1UXFuQT09", "ciphertext":
↪"VDdxQVo3TFFCbXIzcGpYa1lJbFFZQT09"}'
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import unpad
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     iv = b64decode(b64['iv'])
>>>     ct = b64decode(b64['ciphertext'])
>>>     cipher = AES.new(key, AES.MODE_CBC, iv)
>>>     pt = unpad(cipher.decrypt(ct), AES.block_size)
>>>     print("The message was: ", pt)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

CTR mode

CounTeR mode, defined in NIST SP 800-38A, section 6.5 and Appendix B. This mode turns the block cipher into a stream cipher. Each byte of plaintext is XOR-ed with a byte taken from a *keystream*: the result is the ciphertext. The *keystream* is generated by encrypting a sequence of *counter blocks* with ECB.



A *counter block* is exactly as long as the cipher block size (e.g. 16 bytes for AES). It consist of the concatenation of two pieces:

1. a fixed **nonce**, set at initialization.
2. a variable **counter**, which gets increased by 1 for any subsequent counter block. The counter is big endian encoded.

The `new()` function at the module level under `Crypto.Cipher` instantiates a new CTR cipher object for the relevant base algorithm. In the following definition, `<algorithm>` could be `AES`:

`Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None, initial_value=None, counter=None)`

Create a new CTR object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_CTR`
- **nonce** (*bytes*) – the value of the fixed nonce. It must be unique for the combination message/key. Its length varies from 0 to the block size minus 1. If not present, the library creates a random nonce of length equal to block size/2.
- **initial_value** (*integer or bytes*) – the value of the counter for the first counter block. It can be either an integer or *bytes* (which is the same integer, just big endian encoded). If not specified, the counter starts at 0.
- **counter** – a custom counter object created with *Crypto.Util.Counter.new()*. This allows the definition of a more complex counter block.

Returns a CTR cipher object

The methods `encrypt()` and `decrypt()` of a CTR cipher object accept data of any length (i.e. padding is not needed). Both raise an `OverflowError` exception as soon as the counter wraps around to repeat the original value.

The CTR cipher object has a read-only attribute `nonce` (*bytes*).

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CTR)
>>> ct_bytes = cipher.encrypt(data)
>>> nonce = b64encode(cipher.nonce).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> result = json.dumps({'nonce':nonce, 'ciphertext':ct})
>>> print(result)
{"nonce": "XqP8WbylRt0=", "ciphertext": "Mie5lqje"}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     nonce = b64decode(b64['nonce'])
>>>     ct = b64decode(b64['ciphertext'])
>>>     cipher = AES.new(key, AES.MODE_CTR, nonce=nonce)
>>>     pt = cipher.decrypt(ct)
>>>     print("The message was: ", pt)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

CFB mode

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. Each byte of plaintext is XOR-ed with a byte taken from a *keystream*: the result is the ciphertext.

The *keystream* is obtained on a per-segment basis: the plaintext is broken up in segments (from 1 byte up to the size of a block). Then, for each segment, the keystream is obtained by encrypting with the block cipher the last piece of ciphertext produced so far - possibly backfilled with the *Initialization Vector*, if not enough ciphertext is available yet.

The `new()` function at the module level under `Crypto.Cipher` instantiates a new CFB cipher object for the relevant base algorithm. In the following definition, `<algorithm>` could be `AES`:

`Crypto.Cipher.<algorithm>.new(key, mode, *, iv=None, segment_size=8)`
Create a new CFB object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_CFB`
- **iv** (*bytes*) – the *Initialization Vector*. It must be unique for the combination message/key. It is as long as the block size (e.g. 16 bytes for AES). If not present, the library creates a random IV.
- **segment_size** (*integer*) – the number of **bits** (not bytes!) the plaintext and the ciphertext are segmented in (default if not specified: 8 bits = 1 byte).

Returns a CFB cipher object

The methods `encrypt()` and `decrypt()` of a CFB cipher object accept data of any length (i.e. padding is not needed).

The CFB cipher object has a read-only attribute `iv` (*bytes*), holding the Initialization Vector.

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CFB)
>>> ct_bytes = cipher.encrypt(data)
>>> iv = b64encode(cipher.iv).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> result = json.dumps({'iv':iv, 'ciphertext':ct})
>>> print(result)
{"iv": "VoamO23kFSOZcK1O2WiCDQ==", "ciphertext": "f8jciJ8/"}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
```

(continues on next page)

(continued from previous page)

```
>>>     nonce = b64decode(b64['nonce'])
>>>     ct = b64decode(b64['ciphertext'])
>>>     cipher = AES.new(key, AES.MODE_CFB, iv=iv)
>>>     pt = cipher.decrypt(ct)
>>>     print("The message was: ", pt)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

OFB mode

Output FeedBack, defined in NIST SP 800-38A, section 6.4. It is another mode that leads to a stream cipher. Each byte of plaintext is XOR-ed with a byte taken from a *keystream*: the result is the ciphertext. The *keystream* is obtained by recursively encrypting the *Initialization Vector*.

The `new()` function at the module level under `Crypto.Cipher` instantiates a new OFB cipher object for the relevant base algorithm. In the following definition, `<algorithm>` could be `AES`:

`Crypto.Cipher.<algorithm>.new(key, mode, *, iv=None)`

Create a new OFB object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_OFB`
- **iv** (*bytes*) – the *Initialization Vector*. It must be unique for the combination message/key. It is as long as the block size (e.g. 16 bytes for AES). If not present, the library creates a random IV.

Returns an OFB cipher object

The methods `encrypt()` and `decrypt()` of an OFB cipher object accept data of any length (i.e. padding is not needed).

The OFB cipher object has a read-only attribute `iv` (*bytes*), holding the Initialization Vector.

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_OFB)
>>> ct_bytes = cipher.encrypt(data)
>>> iv = b64encode(cipher.iv).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> result = json.dumps({'iv':iv, 'ciphertext':ct})
>>> print(result)
{"iv": "NUuRJbL0UMp8+UMCk2/vQA==", "ciphertext": "XGVGc1Gw"}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     nonce = b64decode(b64['nonce'])
>>>     ct = b64decode(b64['ciphertext'])
>>>     cipher = AES.new(key, AES.MODE_OFB, iv=iv)
>>>     pt = cipher.decrypt(ct)
>>>     print("The message was: ", pt)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

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, an OpenPGP 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** (or **IV** - Initialization Vector), AEAD modes require the additional delivery of a **MAC tag**.

This is the state machine for a cipher object:

Beside the usual `encrypt()` and `decrypt()` already available for classic modes of operation, several other methods are present:

`update`(*data*)
Authenticate those parts of the message that get delivered as is, without any encryption (like headers). It is similar to the `update()` method of a MAC object. Note that all data passed to `encrypt()` and `decrypt()` get automatically authenticated already.

Parameters **`data`** (*bytes*) – the extra data to authenticate

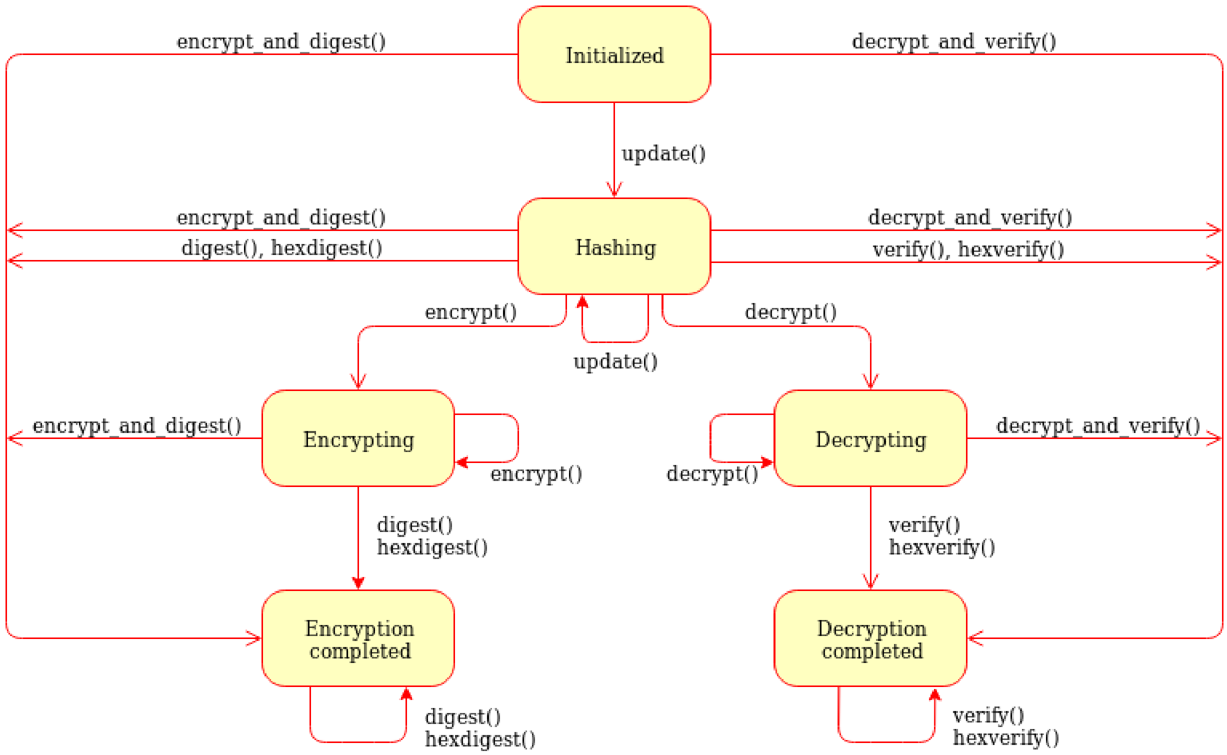


Fig. 5.3: Generic state diagram for a AEAD cipher mode

digest ()
Create the final authentication tag (MAC tag) for a message.

Return bytes the MAC tag

hexdigest ()
Equivalent to `digest()`, with the output encoded in hexadecimal.

Return str the MAC tag as a hexadecimal string

verify (*mac_tag*)
Check if the provided authentication tag (MAC tag) is valid, that is, if the message has been decrypted using the right key and if no modification has taken place in transit.

Parameters **mac_tag** (*bytes*) – the MAC tag

Raises **ValueError** – if the MAC tag is not valid, that is, if the entire message should not be trusted.

hexverify (*mac_tag_hex*)
Same as `verify()` but accepts the MAC tag encoded as an hexadecimal string.

Parameters **mac_tag_hex** (*str*) – the MAC tag as a hexadecimal string

Raises **ValueError** – if the MAC tag is not valid, that is, if the entire message should not be trusted.

encrypt_and_digest (*plaintext*, *output=None*)
Perform `encrypt()` and `digest()` in one go.

Parameters **plaintext** (*bytes*) – the last piece of plaintext to encrypt

Keyword Arguments **output** (*bytes/bytearray/memoryview*) – the pre-allocated buffer where the ciphertext must be stored (as opposed to being returned).

Returns

a tuple with two items

- the ciphertext, as `bytes`
- the MAC tag, as `bytes`

The first item becomes `None` when the `output` parameter specified a location for the result.

decrypt_and_verify(*ciphertext*, *mac_tag*, *output=None*)
Perform `decrypt()` and `verify()` in one go.

Parameters **ciphertext** (*bytes*) – the last piece of ciphertext to decrypt

Keyword Arguments **output** (*bytes/bytearray/memoryview*) – the pre-allocated buffer where the plaintext must be stored (as opposed to being returned).

Raises **ValueError** – if the MAC tag is not valid, that is, if the entire message should not be trusted.

CCM mode

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 at the module level under `Crypto.Cipher` instantiates a new CCM cipher object for the relevant base algorithm. In the following definition, `<algorithm>` can only be `AES` today:

Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None, mac_len=None, msg_len=None, assoc_l
Create a new CCM object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_CCM`
- **nonce** (*bytes*) – the value of the fixed nonce. It must be unique for the combination message/key. For AES, its length varies from 7 to 13 bytes. The longer the nonce, the smaller the allowed message size (with a nonce of 13 bytes, the message cannot exceed 64KB). If not present, the library creates a 11 bytes random nonce (the maximum message size is 8GB).
- **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.

Returns a CTR cipher object

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

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CCM)
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in cipher.nonce, header, ciphertext,
↪ tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "p6ffzcKw+6xopVQ=", "header": "aGVhZGVy", "ciphertext": "860kZo/G", "tag":
↪"Ck5YpVCM6fdWnFkFxw8K6A=="}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>>     jv = {k:b64decode(b64[k]) for k in json_k}
>>>
>>>     cipher = AES.new(key, AES.MODE_CCM, nonce=jv['nonce'])
>>>     cipher.update(jv['header'])
>>>     plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
>>>     print("The message was: " + plaintext)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

EAX mode

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

The `new()` function at the module level under `Crypto.Cipher` instantiates a new EAX cipher object for the relevant base algorithm.

`Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None, mac_len=None)`

Create a new EAX object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_EAX`
- **nonce** (*bytes*) – the value of the fixed nonce. It must be unique for the combination message/key. If not present, the library creates a random nonce (16 bytes long for AES).

- **mac_len** (*integer*) – the desired length of the MAC tag (default if not present: the cipher's block size, 16 bytes for AES).

Returns an EAX cipher object

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

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_EAX)
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in cipher.nonce, header, ciphertext,
↪ tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "CSIJ+e8KP7HJo+hC4RXIyQ==", "header": "aGVhZGVy", "ciphertext": "9YYjuAn6",
↪"tag": "kXHrs9ZwYmjDkmfEJx7Clg=="}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>>     jv = {k:b64decode(b64[k]) for k in json_k}
>>>
>>>     cipher = AES.new(key, AES.MODE_EAX, nonce=jv['nonce'])
>>>     cipher.update(jv['header'])
>>>     plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
>>>     print("The message was: " + plaintext)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

GCM mode

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

The `new()` function at the module level under `Crypto.Cipher` instantiates a new GCM cipher object for the relevant base algorithm.

`Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None, mac_len=None)`
Create a new GCM object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_GCM`
- **nonce** (*bytes*) – the value of the fixed nonce. It must be unique for the combination message/key. If not present, the library creates a random nonce (16 bytes long for AES).
- **mac_len** (*integer*) – the desired length of the MAC tag, from 4 to 16 bytes (default: 16).

Returns a GCM cipher object

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

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_GCM)
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in cipher.nonce, header, ciphertext,
↪ tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "DpOK8NIOuSOQlTq+BphKWw==", "header": "aGVhZGVy", "ciphertext": "CZVqyacc",
↪"tag": "B2tBgICbyw+Wji9KpLVa8w=="}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import unpad
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>>     jv = {k:b64decode(b64[k]) for k in json_k}
>>>
>>>     cipher = AES.new(key, AES.MODE_GCM, nonce=jv['nonce'])
>>>     cipher.update(jv['header'])
>>>     plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
>>>     print("The message was: " + plaintext)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

SIV mode

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

Although less efficient than other modes, SIV is *nonce misuse-resistant*: accidental reuse of the nonce does not jeopardize the security as it happens with CCM or GCM. As a matter of fact, operating **without** a nonce is not an error per se: the cipher simply becomes **deterministic**. In other words, a message gets always encrypted into the same ciphertext.

The `new()` function at the module level under `Crypto.Cipher` instantiates a new SIV cipher object for the relevant base algorithm.

`Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None)`

Create a new SIV object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key; it must be twice the size of the key required by the underlying cipher (e.g. 32 bytes for AES-128).
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_SIV`
- **nonce** (*bytes*) – the value of the fixed nonce. It must be unique for the combination message/key. If not present, the encryption will be deterministic.

Returns a SIV cipher object

If the *nonce* parameter was provided to `new()`, the resulting cipher object has a read-only attribute `nonce`.

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16 * 2)
>>> nonce = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_SIV, nonce=nonce)     # Without nonce, the␣
↪encryption
>>>                                                      # becomes deterministic
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in nonce, header, ciphertext, tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "zMiifAVvDpMS8hnGK/z+iw==", "header": "aGVhZGVy", "ciphertext": "Q7lReEAF",
↪"tag": "KgdnBVbCee6B/wGmMf/wQA=="}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # We assume that the key was securely shared beforehand
>>> try:
>>>     b64 = json.loads(json_input)
>>>     json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>>     jv = {k:b64decode(b64[k]) for k in json_k}
>>>
>>>     cipher = AES.new(key, AES.MODE_SIV, nonce=jv['nonce'])
>>>     cipher.update(jv['header'])
>>>     plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
>>>     print("The message was: " + plaintext)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

One side-effect is that encryption (or decryption) must take place in one go with the method `encrypt_and_digest()` (or `decrypt_and_verify()`). You cannot use `encrypt()` or `decrypt()`. The state diagram is therefore:

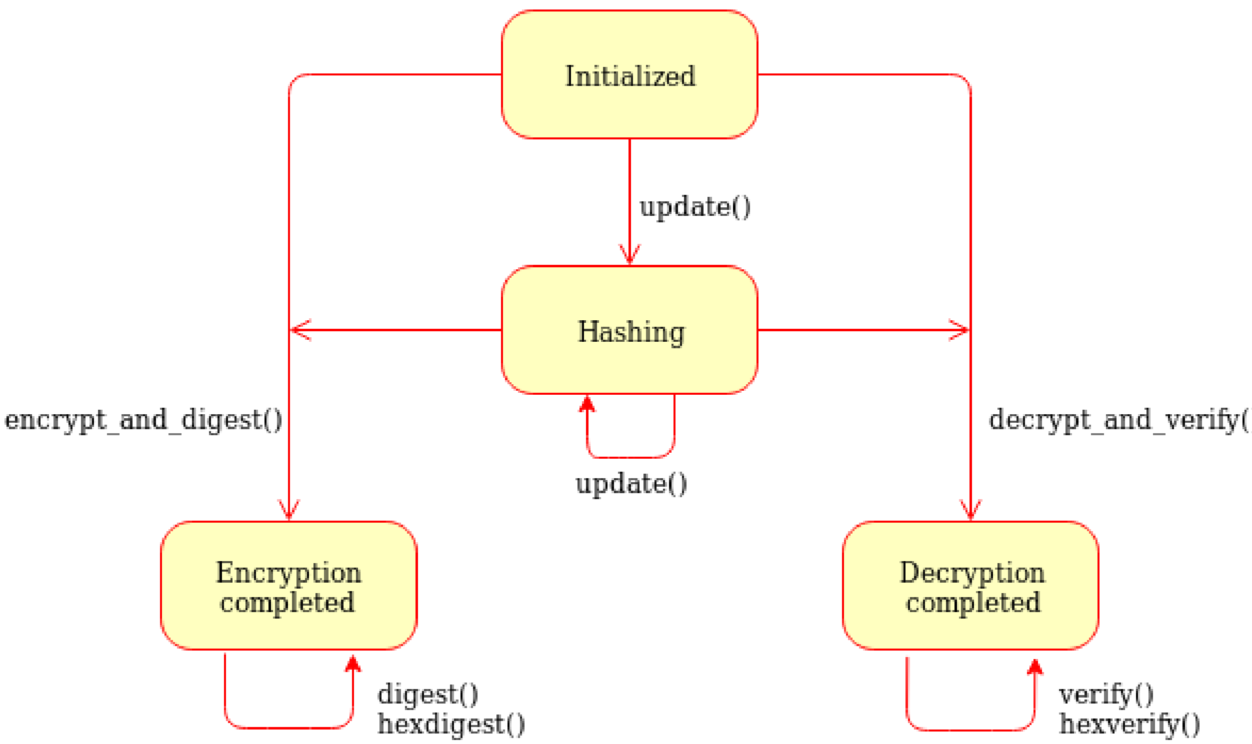


Fig. 5.4: State diagram for the SIV cipher mode

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 full 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")
```

OCB mode

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 at the module level under `Crypto.Cipher` instantiates a new OCB cipher object for the relevant base algorithm.

`Crypto.Cipher.<algorithm>.new(key, mode, *, nonce=None, mac_len=None)`

Create a new OCB object, using <algorithm> as the base block cipher.

Parameters

- **key** (*bytes*) – the cryptographic key
- **mode** – the constant `Crypto.Cipher.<algorithm>.MODE_OCB`
- **nonce** (*bytes*) – the value of the fixed nonce, wuth length between 1 and 15 bytes. It must be unique for the combination message/key. If not present, the library creates a 15 bytes random nonce.
- **mac_len** (*integer*) – the desired length of the MAC tag (default if not present: 16 bytes).

Returns an OCB cipher object

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

Example (encryption):

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_OCB)
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in cipher.nonce, header, ciphertext,
↪ tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "I7E6PKxHNYo2i9sz8W98", "header": "aGVhZGVy", "ciphertext": "nYJnJ8jC", "tag
↪": "0UbFcmO9lqGknCIDWRLALA=="}
```

Example (decryption):

```
>>> import json
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>>
>>> # We assume that the key was securely shared beforehand
>>> try:
```

(continues on next page)

(continued from previous page)

```
>>>     b64 = json.loads(json_input)
>>>     json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>>     jv = {k:b64decode(b64[k]) for k in json_k}
>>>
>>>     cipher = AES.new(key, AES.MODE_OCB, nonce=jv['nonce'])
>>>     cipher.update(jv['header'])
>>>     plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
>>>     print("The message was: " + plaintext)
>>> except ValueError, KeyError:
>>>     print("Incorrect decryption")
```

5.1.4 Legacy 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)

5.2 `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*.

5.2.1 Signing a message

1. Instantiate a new signer object for the desired algorithm, for instance with `Crypto.Signature.pkcs1_15.new()`. The first parameter is the key object (*private* key) obtained via the `Crypto.PublicKey` module.
2. Instantiate a cryptographic hash object, for instance with `Crypto.Hash.SHA384.new()`. Then, process the message with its *`update()`* method.
3. Invoke the `sign()` method on the signer with the hash object as parameter. The output is the signature of the message (a byte string).

5.2.2 Verifying a signature

1. Instantiate a new verifier object for the desired algorithm, for instance with `Crypto.Signature.pkcs1_15.new()`. The first parameter is the key object (*public* key) obtained via the `Crypto.PublicKey` module.

2. Instantiate a cryptographic hash object, for instance with `Crypto.Hash.SHA384.new()`. Then, process the message with its `update()` method.
3. Invoke the `verify()` method on the verifier, with the hash object and the incoming signature as parameters. If the message is not authentic, an `ValueError` is raised.

5.2.3 Available mechanisms

- pkcs1_v1_5
- pkcs1_pss
- dsa

5.3 `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.

5.3.1 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')
```

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

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:

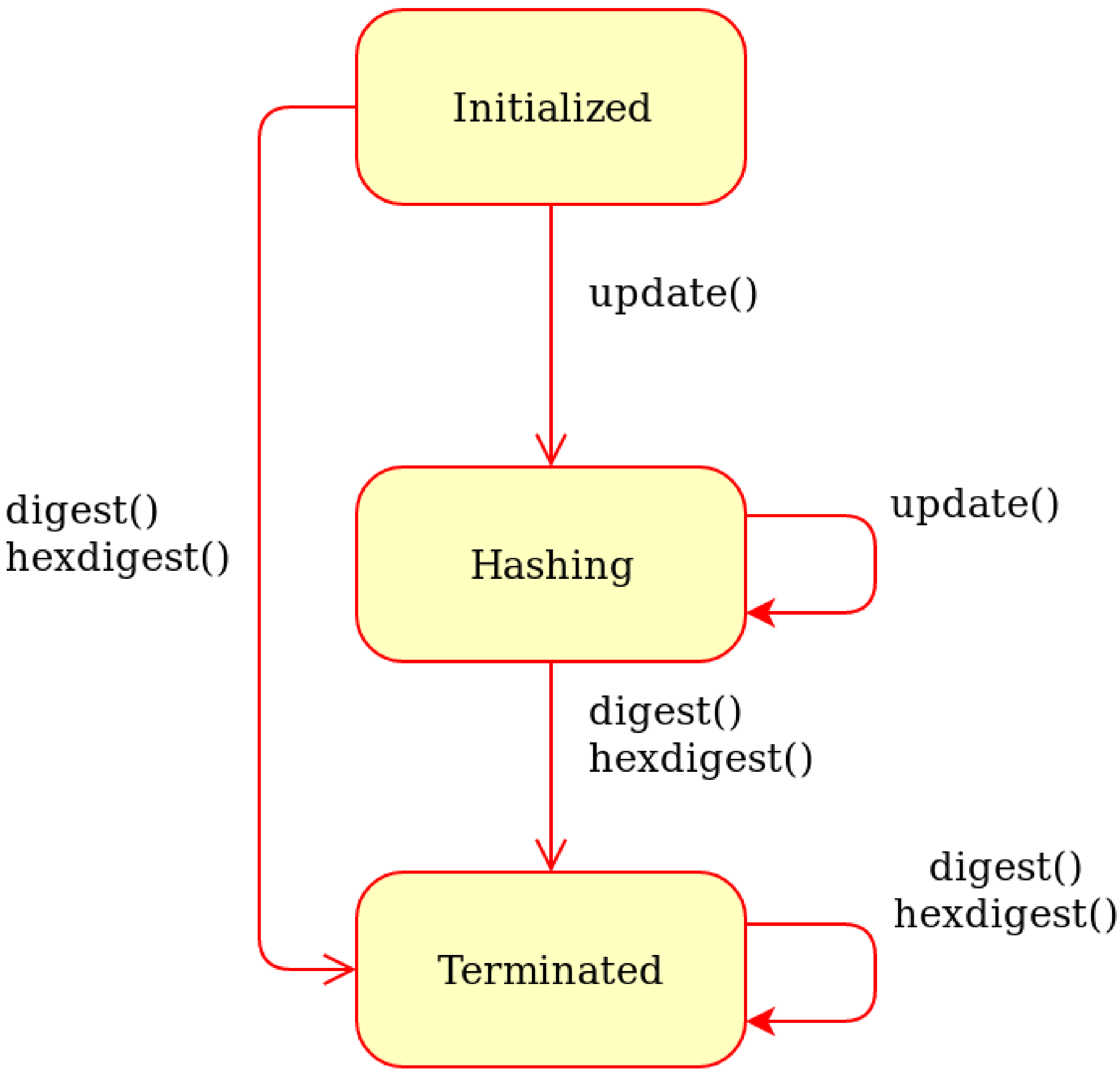


Fig. 5.5: Generic state diagram for a hash object

```
>>> 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
```

5.3.2 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.

5.3.3 Modern hash algorithms

- SHA-2 family
  - sha224
  - sha256
  - sha384
  - sha512
- SHA-3 family
  - sha3_224
  - sha3_256
  - sha3_384
  - sha3_512
- BLAKE2
  - blake2s
  - blake2b

5.3.4 Extensible-Output Functions (XOF)

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

5.3.5 Message Authentication Code (MAC) algorithms

- hmac
- cmac
- poly1305

5.3.6 Historich hash algorithms

The following algorithms should not be used in new designs:

- sha1
- md2
- md5
- ripemd160
- keccak

5.4 `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.

5.4.1 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 `export_key()` 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).

5.4.2 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','wb')
>>> f.write(key.export_key('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:

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

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}mod 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!

export_key (*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().export_key())
>>> f.close()
>>>
>>> # Sign a message
>>> message = b"Hello"
>>> hash_obj = SHA256.new(message)
>>> signer = DSS.new(key, 'fips-186-3')
>>> signature = signer.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())
>>> verifier = DSS.new(pub_key, 'fips-186-3')
>>>
>>> # Verify the authenticity of the message
>>> try:
>>>     verifier.verify(hash_obj, signature)
>>>     print "The message is authentic."
>>> except ValueError:
>>>     print "The message is not authentic."
```

`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!

export_key (*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 to perform a key exchange.

Compared to traditional algorithms like RSA, an ECC key is significantly smaller at the same security level. For instance, a 3072-bit RSA key takes 768 bytes whereas the equally strong NIST P-256 private key only takes 32 bytes (that is, 256 bits).

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

Curve	Possible identifiers
NIST P-256	'NIST P-256', 'p256', 'P-256', 'prime256v1', 'secp256r1'
NIST P-384	'NIST P-384', 'p384', 'P-384', 'prime384v1', 'secp384r1'
NIST P-521	'NIST P-521', 'p521', 'P-521', 'prime521v1', 'secp521r1'

For more information about each NIST curve see FIPS 186-4, Section D.1.2.

The following example demonstrates how to generate a new ECC 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()
>>> f.close()
...
>>> f = open('myprivatekey.pem','rt')
>>> key = ECC.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 as defined in Table 5.4.2.
- **pointQ** (*EccPoint*) – an ECC point representating the public component
- **d** (*integer*) – A scalar representating the private component

export_key(***kwargs*)

Export this ECC key.

Parameters

- **format** (*string*) – The format to use for encoding the key:
  - `'DER'`. The key will be encoded in ASN.1 DER format (binary). For a public key, the ASN.1 `subjectPublicKeyInfo` structure defined in RFC5480 will be used. For a private key, the ASN.1 `ECPrivateKey` structure defined in RFC5915 is used instead (possibly within a PKCS#8 envelope, see the `use_pkcs8` flag below).
  - `'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*) – Only relevant for private keys.

  If `True` (default and recommended), the PKCS#8 representation will be used.

  If `False`, the much weaker 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`.
- **compress** (*boolean*) – If `True`, a more compact representation of the public key with the X-coordinate only is used.

  If `False` (default), the full public key will be exported.

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 to `export_key()` will be 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*, *curve='p256'*)

A class to abstract a point over an Elliptic Curve.

The class support special methods for:

- Adding two points: `R = S + T`
- In-place addition: `S += T`
- Negating a point: `R = -T`
- Comparing two points: `if S == T: ...`
- Multiplying a point by a scalar: `R = S*k`
- In-place multiplication by a scalar: `T *= k`

Variables

- **x** (*integer*) – The affine X-coordinate of the ECC point
- **y** (*integer*) – The affine Y-coordinate of the ECC point
- **xy** – The tuple with X- and Y- coordinates

copy()

Return a copy of this point.

double()

Double this point (in-place operation).

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

is_point_at_infinity()
True if this is the point-at-infinity.

point_at_infinity()
Return the point-at-infinity for the curve this point is on.

size_in_bits()
Size of each coordinate, in bits.

size_in_bytes()
Size of each coordinate, in bytes.

exception Crypto.PublicKey.ECC.**UnsupportedEccFeature**

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 a curve name defined in Table 5.4.2.
- **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 a curve name defined in Table 5.4.2.
- **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*

5.4.3 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:

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

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*

5.5 `Crypto.Protocol` package

5.5.1 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*, *hmac_hash_module=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 or byte string*) – The secret password to generate the key from.
- **salt** (*string or byte string*) – A (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. 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.
- **hmac_hash_module** (*module*) – A module from *Crypto.Hash* implementing a Merkle-Damgard cryptographic hash, which PBKDF2 must use in combination with HMAC. This parameter is mutually exclusive with `prf`.

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.

5.5.2 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:
```

(continues on next page)

(continued from previous page)

```
>>>     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*

5.6 `Crypto.IO` package

Modules for reading and writing cryptographic data.

- *PEM*
- *PKCS#8*

5.6.1 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.

5.6.2 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

5.7 `Crypto.Random` package

`Crypto.Random.`**`get_random_bytes`**(*N*)

Return a random byte string of length *N*.

5.7.1 `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*.

5.8 `Crypto.Util` package

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

5.8.1 `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*, *strict=False*`)`
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
```

(continues on next page)

(continued from previous page)

```
>>> except ValueError:
>>>   print "Not a valid DER INTEGER"
```

the output will be 9.

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

decode (*der_encoded*, *strict=False*)
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')
```

(continues on next page)

(continued from previous page)

```
>>> seq_der = DerSequence([4])
>>> seq_der.append(9)
>>> seq_der.append(obj_der.encode())
>>> print hexlify(seq_der.encode())
```

which will show `3009020104020109070102`, the DER encoding of the sequence containing `4`, `9`, and the object with payload `02`.

For decoding:

```
>>> s = unhexlify(b'3009020104020109070102')
>>> 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*, *strict=False*, *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.
- **strict** (*boolean*) – Whether decoding must check for strict DER compliancy.

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*, *strict=False*)
Decode a complete DER OBJECT ID, and re-initializes this object with it.

Parameters

- **der_encoded** (*byte string*) – A complete DER OBJECT ID.
- **strict** (*boolean*) – Whether decoding must check for strict DER compliancy.

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*, *strict=False*)
Decode a complete DER BIT STRING, and re-initializes this object with it.

Parameters

- **der_encoded** (*byte string*) – a complete DER BIT STRING.
- **strict** (*boolean*) – Whether decoding must check for strict DER compliancy.

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*, *strict=False*)
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.
- **strict** (*boolean*) – Whether decoding must check for strict DER compliancy.

Raises `ValueError` – in case of parsing errors.

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

5.8.2 Crypto.Util.Padding module

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

```
>>> from Crypto.Util.Padding import pad, unpad
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b'Unaligned'   # 9 bytes
>>> key = get_random_bytes(32)
>>> iv = get_random_bytes(16)
>>>
>>> cipher1 = AES.new(key, AES.MODE_CBC, iv)
>>> ct = cipher1.encrypt(pad(data, 16))
>>>
>>> cipher2 = AES.new(key, AES.MODE_CBC, iv)
>>> pt = unpad(cipher2.decrypt(ct), 16)
>>> assert(data == pt)
```

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.

5.8.3 `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.

5.8.4 `Crypto.Util.strxor` module

Fast XOR for byte strings.

`Crypto.Util.strxor.`**`strxor`**(*term1*, *term2*, *output=None*)

XOR two byte strings.

Parameters

- **term1** (*bytes/bytearray/memoryview*) – The first term of the XOR operation.
- **term2** (*bytes/bytearray/memoryview*) – The second term of the XOR operation.
- **output** (*bytearray/memoryview*) – The location where the result must be written to. If `None`, the result is returned.

Return If `output` is `None`, a new `bytes` string with the result. Otherwise `None`.

`Crypto.Util.strxor.`**`strxor_c`**(*term*, *c*, *output=None*)

XOR a byte string with a repeated sequence of characters.

Parameters

- **term** (*bytes/bytearray/memoryview*) – The first term of the XOR operation.
- **c** (*bytes*) – The byte that makes up the second term of the XOR operation.

- **output** (*None or bytearray/memoryview*) – If not None, the location where the result is stored into.

Returns If output is None, a new bytes string with the result. Otherwise None.

5.8.5 Crypto.Util.Counter module

Richer counter functions for CTR cipher mode.

CTR is a mode of operation for block ciphers.

The plaintext is broken up in blocks and each block is XOR-ed with a *keystream* to obtain the ciphertext. The *keystream* is produced by the encryption of a sequence of *counter blocks*, which all need to be different to avoid repetitions in the keystream. Counter blocks don't need to be secret.

The most straightforward approach is to include a counter field, and increment it by one within each subsequent counter block.

The new() function at the module level under Crypto.Cipher instantiates a new CTR cipher object for the relevant base algorithm. Its parameters allow you define a counter block with a fixed structure:

- an optional, fixed prefix
- the counter field encoded in big endian mode

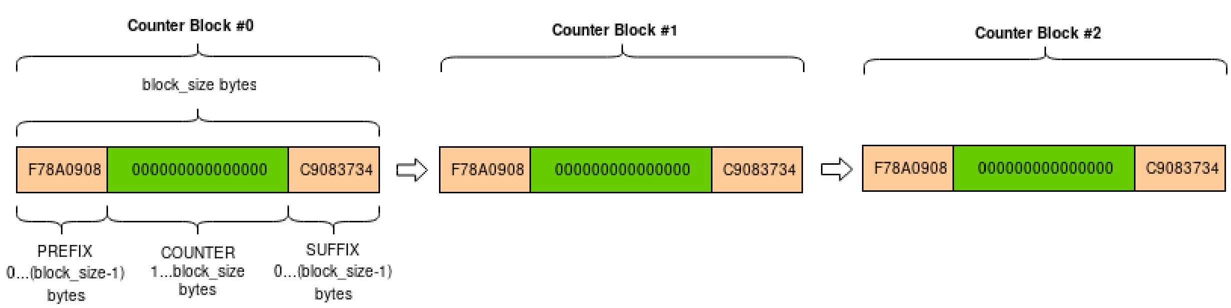
The length of the two components can vary, but together they must be as large as the block size (e.g. 16 bytes for AES).

Alternatively, the counter parameter can be used to pass a counter block object (created in advance with the function *Crypto.Util.Counter.new()*) for a more complex composition:

- an optional, fixed prefix
- the counter field, encoded in big endian or little endian mode
- an optional, fixed suffix

As before, the total length must match the block size.

The counter blocks with a big endian counter will look like this:



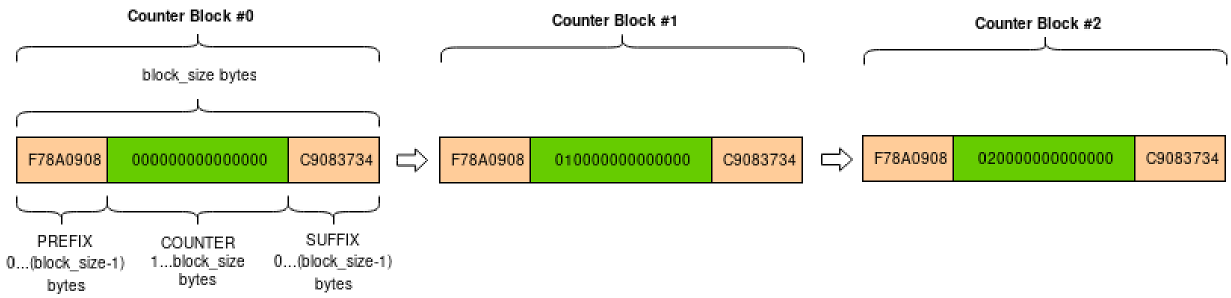
The counter blocks with a little endian counter will look like this:

Example of AES-CTR encryption with custom counter:

```
from Crypto.Cipher import AES
from Crypto.Util import Counter
from Crypto import Random

nonce = Random.get_random_bytes(4)
```

(continues on next page)



(continued from previous page)

```
ctr = Counter.new(64, prefix=nonce, suffix=b'ABCD', little_endian=True, initial_
↪value=10)
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.

5.8.6 `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.`**`ceil_div`**(*n*, *d*)

Return ceil(n/d), that is, the smallest integer r such that r*d >= n

`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

6.1 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)
```

6.2 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.export_key(passphrase=secret_code, pkcs=8,
                              protection="scryptAndAES128-CBC")

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

print(key.publickey().export_key())
```

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().export_key())
```

6.3 Generate public key and private key

The following code generates public key stored in `receiver.pem` and private key stored in `private.pem`. These files will be used in the examples below. Every time, it generates different public key and private key pair.

```
from Crypto.PublicKey import RSA

key = RSA.generate(2048)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)

public_key = key.publickey().export_key()
file_out = open("receiver.pem", "wb")
file_out.write(public_key)
```

6.4 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

data = "I met aliens in UFO. Here is the map.".encode("utf-8")
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)
enc_session_key = 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 (enc_session_key, cipher_aes.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 private 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_aes.decrypt_and_verify(ciphertext, tag)
print(data.decode("utf-8"))
```

CHAPTER 7

Frequently Asked Questions

7.1 Is CTR cipher mode compatible with Java?

Yes. When you instantiate your AES cipher in Java:

```
Cipher  cipher = Cipher.getInstance("AES/CTR/NoPadding");

SecretKeySpec keySpec = new SecretKeySpec(new byte[16], "AES");
IvParameterSpec ivSpec = new IvParameterSpec(new byte[16]);

cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec);
```

You are effectively using *CTR mode* without a fixed nonce and with a 128-bit big endian counter starting at 0. The counter will wrap around only after 2^{128} blocks.

You can replicate the same keystream in PyCryptodome with:

```
ivSpec = b'\x00' * 16
ctr = AES.new(keySpec, AES.MODE_CTR, initial_value=ivSpec)
```

7.2 Are RSASSA-PSS signatures compatible with Java or OpenSSL?

Yes. For Java, you must consider that by default the mask is generated by MGF1 with SHA-1 (regardless of how you hash the message) and the salt is 20 bytes long.

If you want to use another algorithm or another salt length, you must instantiate a `PSSParameterSpec` object, for instance:

```
Signature ss = Signature.getInstance("SHA256withRSA/PSS");
AlgorithmParameters pss1 = ss.getParameters();
```

(continues on next page)

(continued from previous page)

```
PSSParameterSpec pssParameterSpec = new PSSParameterSpec("SHA-256", "MGF1", new␣
↪MGF1ParameterSpec("SHA-256"), 32, 0xBC);
ss.setParameter(spec1);
```

On the other hand, a quirk of OpenSSL (and of a few other libraries, especially if they are wrappers to OpenSSL) is that the default salt length is maximized, and it does not match in size the digest applied to the message, as recommended in RFC8017. In PyCryptodome, you maximize the salt length with:

```
key = RSA.import_key(open('privkey.der').read())
h = SHA256.new(message)
salt_bytes = key.size_in_bytes() - h.digest_size - 2
signature = pss.new(key, salt_bytes=salt_bytes).sign(h)
```

7.3 Why do I get the error `No module named Crypto` on Windows?

Check the directory where Python packages are installed, like:

```
/path/to/python/Lib/site-packages/
```

You might find a directory named `crypto`, with all the PyCryptodome files in it.

The most likely cause is described here and you can fix the problem with:

```
pip uninstall crypto
pip uninstall pycryptodome
pip install pycryptodome
```

The root cause is that, in the past, you most likely have installed an unrelated but similarly named package called crypto, which happens to operate under the namespace `crypto`.

The Windows filesystem is **case-insensitive** so `crypto` and `Crypto` are effectively considered the same thing. When you subsequently install `pycryptodome`, `pip` finds that a directory named with the target namespace already exists (under the rules of the underlying filesystem), and therefore installs all the sub-packages of `pycryptodome` in it. This is probably a reasonable behavior, if it wansn't that pip does not issue any warning even if it could detect the issue.

CHAPTER 8

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, highlight it and include a justification.
- Ensure that your code complies to PEP8 and PEP257.
- If you add or modify a public interface, make sure the relevant type stubs remain up to date.
- Ensure that your code does not use constructs or includes modules not present in Python 2.6.
- 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 9

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
- 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:**
  - 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
- Run lint on the C code
- Add (minimal) support for PGP
- Add (minimal) support for PKIX / X.509

CHAPTER 10

Changelog

10.1 3.8.1 (4 April 2019)

10.1.1 New features

- Add support for loading PEM files encrypted with AES192-CBC, AES256-CBC, and AES256-GCM.
- When importing ECC keys, ignore EC PARAMS section that was included by some openssl commands.

10.1.2 Resolved issues

- repr() did not work for `ECC.EccKey`.
- Fix installation in development mode.
- Minimal length for Blowfish cipher is 32 bits, not 40 bits.
- Various updates to docs.

10.2 3.8.0 (23 March 2019)

10.2.1 New features

- Speed-up ECC performance. ECDSA is 33 times faster on the NIST P-256 curve.
- Added support for NIST P-384 and P-521 curves.
- `EccKey` has new methods `size_in_bits()` and `size_in_bytes()`.
- Support HMAC-SHA224, HMAC-SHA256, HMAC-SHA384, and HMAC-SHA512 in PBE2/PBKDF2.

10.2.2 Resolved issues

- DER objects were not rejected if their length field had a leading zero.
- Allow legacy RC2 ciphers to have 40-bit keys.
- ASN.1 Object IDs did not allow the value 0 in the path.

10.2.3 Breaks in compatibility

- `point_at_infinity()` becomes an instance method for `Crypto.PublicKey.ECC.EccKey`, from a static one.

10.3 3.7.3 (19 January 2019)

10.3.1 Resolved issues

- GH#258: False positive on PSS signatures when externally provided salt is too long.
- Include type stub files for `Crypto.IO` and `Crypto.Util`.

10.4 3.7.2 (26 November 2018)

10.4.1 Resolved issues

- GH#242: Fixed compilation problem on ARM platforms.

10.5 3.7.1 (25 November 2018)

10.5.1 New features

- Added type stubs to enable static type checking with mypy. Thanks to Michael Nix.
- New `update_after_digest` flag for CMAC.

10.5.2 Resolved issues

- GH#232: Fixed problem with gcc 4.x when compiling `ghash_clmul.c`.
- GH#238: Incorrect digest value produced by CMAC after cloning the object.
- Method `update()` of an EAX cipher object was returning the underlying CMAC object, instead of the EAX object itself.
- Method `update()` of a CMAC object was not throwing an exception after the digest was computed (with `digest()` or `verify()`).

10.6 3.7.0 (27 October 2018)

10.6.1 New features

- Added support for Poly1305 MAC (with AES and ChaCha20 ciphers for key derivation).
- Added support for ChaCha20-Poly1305 AEAD cipher.
- New parameter `output` for `Crypto.Util.strxor.strxor`, `Crypto.Util.strxor.strxor_c`, `encrypt` and `decrypt` methods in symmetric ciphers (`Crypto.Cipher` package). `output` is a pre-allocated buffer (a `bytearray` or a writeable `memoryview`) where the result must be stored. This requires less memory for very large payloads; it is also more efficient when encrypting (or decrypting) several small payloads.

10.6.2 Resolved issues

- GH#266: AES-GCM hangs when processing more than 4GB at a time on x86 with PCLMULQDQ instruction.

10.6.3 Breaks in compatibility

- Drop support for Python 3.3.
- Remove `Crypto.Util.py3compat.unhexlify` and `Crypto.Util.py3compat.hexlify`.
- With the old Python 2.6, use only `ctypes` (and not `cffi`) to interface to native code.

10.7 3.6.6 (17 August 2018)

10.7.1 Resolved issues

- GH#198: Fix vulnerability on AESNI ECB with payloads smaller than 16 bytes (CVE-2018-15560).

10.8 3.6.5 (12 August 2018)

10.8.1 Resolved issues

- GH#187: Fixed incorrect AES encryption/decryption with AES acceleration on x86 due to gcc's optimization and strict aliasing rules.
- GH#188: More prime number candidates than necessary where discarded as composite due to the limited way D values were searched in the Lucas test.
- Fixed ResouceWarnings and DeprecationWarnings.
- Workaround for Python 3.7.0 bug on Windows (https://bugs.python.org/issue34108).

10.9 3.6.4 (10 July 2018)

10.9.1 New features

- Build Python 3.7 wheels on Linux, Windows and Mac.

10.9.2 Resolved issues

- GH#178: Rename `_cpuid` module to make upgrades more robust.
- More meaningful exceptions in case of mismatch in IV length (CBC/OFB/CFB modes).
- Fix compilation issues on Solaris 10/11.

10.10 3.6.3 (21 June 2018)

10.10.1 Resolved issues

- GH#175: Fixed incorrect results for CTR encryption/decryption with more than 8 blocks.

10.11 3.6.2 (19 June 2018)

10.11.1 New features

- ChaCha20 accepts 96 bit nonces (in addition to 64 bit nonces) as defined in RFC7539.
- Accelerate AES-GCM on x86 using PCLMULQDQ instruction.
- Accelerate AES-ECB and AES-CTR on x86 by pipelining AESNI instructions.
- As result of the two improvements above, on x86 (Broadwell):
  - AES-ECB and AES-CTR are 3x faster
  - AES-GCM is 9x faster

10.11.2 Resolved issues

- On Windows, MPIR library was stilled pulled in if renamed to `gmp.dll`.

10.11.3 Breaks in compatibility

- In `Crypto.Util.number`, functions `floor_div` and `exact_div` have been removed. Also, `ceil_div` is limited to non-negative terms only.

10.12 3.6.1 (15 April 2018)

10.12.1 New features

- Added Google Wycheproof tests (https://github.com/google/wycheproof) for RSA, DSA, ECDSA, GCM, SIV, EAX, CMAC.
- New parameter `mac_len` (length of MAC tag) for CMAC.

10.12.2 Resolved issues

- In certain circumstances (at counter wrapping, which happens on average after 32 GB) AES GCM produced wrong ciphertexts.
- Method `encrypt()` of AES SIV cipher could be still called, whereas only `encrypt_and_digest()` is allowed.

10.13 3.6.0 (8 April 2018)

10.13.1 New features

- Introduced `export_key` and deprecated `exportKey` for DSA and RSA key objects.
- Ciphers and hash functions accept `memoryview` objects in input.
- Added support for SHA-512/224 and SHA-512/256.

10.13.2 Resolved issues

- Reintroduced `Crypto.__version__` variable as in PyCrypto.
- Fixed compilation problem with MinGW.

10.14 3.5.1 (8 March 2018)

10.14.1 Resolved issues

- GH#142. Fix mismatch with declaration and definition of addmul128.

10.15 3.5.0 (7 March 2018)

10.15.1 New features

- Import and export of ECC curves in compressed form.
- The initial counter for a cipher in CTR mode can be a byte string (in addition to an integer).
- Faster PBKDF2 for HMAC-based PRFs (at least 20x for short passwords, more for longer passwords). Thanks to Christian Heimes for pointing out the implementation was under-optimized.

- The salt for PBKDF2 can be either a string or bytes (GH#67).
- Ciphers and hash functions accept data as *bytearray*, not just binary strings.
- The old SHA-1 and MD5 hash functions are available even when Python's own *hashlib* does not include them.

10.15.2 Resolved issues

- Without libgmp, modular exponentiation (since v3.4.8) crashed on 32-bit big-endian systems.

10.15.3 Breaks in compatibility

- Removed support for Python < 2.6.

10.16 3.4.12 (5 February 2018)

10.16.1 Resolved issues

- GH#129. pycryptodomex could only be installed via wheels.

10.17 3.4.11 (5 February 2018)

10.17.1 Resolved issues

- GH#121. the record list was still not correct due to PEP3147 and __pycache__ directories. Thanks again to John O'Brien.

10.18 3.4.10 (2 February 2018)

10.18.1 Resolved issues

- When creating ElGamal keys, the generator wasn't a square residue: ElGamal encryption done with those keys cannot be secure under the DDH assumption. Thanks to Weikeng Chen.

10.19 3.4.9 (1 February 2018)

10.19.1 New features

- More meaningful error messages while importing an ECC key.

10.19.2 Resolved issues

- GH#123 and #125. The SSE2 command line switch was not always passed on 32-bit x86 platforms.
- GH#121. The record list (–record) was not always correctly filled for the pycryptodomex package. Thanks to John W. O'Brien.

10.20 3.4.8 (27 January 2018)

10.20.1 New features

- Added a native extension in pure C for modular exponentiation, optimized for SSE2 on x86. In the process, we drop support for the arbitrary arithmetic library MPIR on Windows, which is painful to compile and deploy. The custom modular exponentiation is 130% (160%) slower on an Intel CPU in 32-bit (64-bit) mode, compared to MPIR. Still, that is much faster that CPython's own *pow()* function which is 900% (855%) slower than MPIR. Support for the GMP library on Unix remains.
- Added support for *manylinux* wheels.
- Support for Python 3.7.

10.20.2 Resolved issues

- The DSA parameter 'p' prime was created with 255 bits cleared (but still with the correct strength).
- GH#106. Not all docs were included in the tar ball. Thanks to Christopher Hoskin.
- GH#109. ECDSA verification failed for DER encoded signatures. Thanks to Alastair Houghton.
- Human-friendly messages for padding errors with ECB and CBC.

10.21 3.4.7 (26 August 2017)

10.21.1 New features

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

10.21.2 Resolved issues

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

10.22 3.4.6 (18 May 2017)

10.22.1 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.

10.23 3.4.5 (6 February 2017)

10.23.1 Resolved issues

- The library can also be compiled using MinGW.

10.24 3.4.4 (1 February 2017)

10.24.1 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.

10.25 3.4.3 (17 October 2016)

10.25.1 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

10.26 3.4.2 (8 March 2016)

10.26.1 Resolved issues

- Fix renaming of package for `install` command.

10.27 3.4.1 (21 February 2016)

10.27.1 New features

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

10.28 3.4 (7 February 2016)

10.28.1 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`.

10.28.2 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`.

10.28.3 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`.

10.29 3.3.1 (1 November 2015)

10.29.1 New features

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

10.29.2 Resolved issues

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

10.29.3 Breaks in compatibility

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

10.30 3.3 (29 October 2015)

10.30.1 New features

- Windows wheels bundle the MPIR library
- Detection of faults occurring 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)

10.30.2 Resolved issues

- GH#3: gcc 4.4.7 unhappy about double typedef

10.30.3 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()`

10.31 3.2.1 (9 September 2015)

10.31.1 New features

- Windows wheels are automatically built on Appveyor

10.32 3.2 (6 September 2015)

10.32.1 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.

10.32.2 Resolved issues

- tomcrypt_macros.h asm error #1

10.32.3 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`.

10.33 3.1 (15 March 2015)

10.33.1 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).

10.33.2 Resolved issues

- None

10.33.3 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.

10.34 3.0 (24 June 2014)

10.34.1 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`.

10.34.2 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).

10.34.3 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`.

10.34.4 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 11

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.

11.1 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>

11.2 BSD license

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

11.3 OCB license

The OCB cipher mode is patented in the US under patent numbers 7,949,129 and 8,321,675. 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.

11.4 Apache 2.0 license (Wycheproof)

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works

that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Python Module Index

C

Crypto.IO.PEM, 57
Crypto.IO.PKCS8, 58
Crypto.Protocol.KDF, 53
Crypto.Protocol.SecretSharing, 55
Crypto.PublicKey.DSA, 44
Crypto.PublicKey.ECC, 48
Crypto.PublicKey.ElGamal, 52
Crypto.PublicKey.RSA, 39
Crypto.Util.asn1, 60
Crypto.Util.Counter, 68
Crypto.Util.number, 68
Crypto.Util.Padding, 65
Crypto.Util.RFC1751, 66
Crypto.Util.strxor, 66

Index

A

add() (*Crypto.Util.asn1.DerSetOf method*), 64

B

bytes_to_long() (*in module Crypto.Util.number*), 68

C

ceil_div() (*in module Crypto.Util.number*), 69
combine() (*Crypto.Protocol.SecretSharing.Shamir static method*), 56
construct() (*in module Crypto.PublicKey.DSA*), 44
construct() (*in module Crypto.PublicKey.ECC*), 50
construct() (*in module Crypto.PublicKey.ElGamal*), 52
construct() (*in module Crypto.PublicKey.RSA*), 39
copy() (*Crypto.PublicKey.ECC.EccPoint method*), 49
Crypto.IO.PEM (*module*), 57
Crypto.IO.PKCS8 (*module*), 58
Crypto.Protocol.KDF (*module*), 53
Crypto.Protocol.SecretSharing (*module*), 55
Crypto.PublicKey.DSA (*module*), 44
Crypto.PublicKey.ECC (*module*), 48
Crypto.PublicKey.ElGamal (*module*), 52
Crypto.PublicKey.RSA (*module*), 39
Crypto.Random.get_random_bytes() (*built-in function*), 59
Crypto.Random.random.choice() (*built-in function*), 60
Crypto.Random.random.getrandbits() (*built-in function*), 59
Crypto.Random.random.randint() (*built-in function*), 60
Crypto.Random.random.randrange() (*built-in function*), 59
Crypto.Random.random.sample() (*built-in function*), 60
Crypto.Random.random.shuffle() (*built-in function*), 60
Crypto.Util.asn1 (*module*), 60
Crypto.Util.Counter (*module*), 68
Crypto.Util.number (*module*), 68
Crypto.Util.Padding (*module*), 65
Crypto.Util.RFC1751 (*module*), 66
Crypto.Util.strxor (*module*), 66

D

decode() (*Crypto.Util.asn1.DerBitString method*), 64
decode() (*Crypto.Util.asn1.DerInteger method*), 61
decode() (*Crypto.Util.asn1.DerObject method*), 60
decode() (*Crypto.Util.asn1.DerObjectId method*), 63
decode() (*Crypto.Util.asn1.DerSequence method*), 62
decode() (*Crypto.Util.asn1.DerSetOf method*), 64
decode() (*in module Crypto.IO.PEM*), 57
decrypt_and_verify(), 27
DerBitString (*class in Crypto.Util.asn1*), 63
DerInteger (*class in Crypto.Util.asn1*), 60
DerNull (*class in Crypto.Util.asn1*), 61
DerObject (*class in Crypto.Util.asn1*), 60
DerObjectId (*class in Crypto.Util.asn1*), 63
DerOctetString (*class in Crypto.Util.asn1*), 61
DerSequence (*class in Crypto.Util.asn1*), 61
DerSetOf (*class in Crypto.Util.asn1*), 64
digest(), 25
domain() (*Crypto.PublicKey.DSA.DsaKey method*), 45
double() (*Crypto.PublicKey.ECC.EccPoint method*), 49
DsaKey (*class in Crypto.PublicKey.DSA*), 45

E

EccKey (*class in Crypto.PublicKey.ECC*), 48
EccPoint (*class in Crypto.PublicKey.ECC*), 49
ElGamalKey (*class in Crypto.PublicKey.ElGamal*), 52
encode() (*Crypto.Util.asn1.DerBitString method*), 64
encode() (*Crypto.Util.asn1.DerInteger method*), 61
encode() (*Crypto.Util.asn1.DerObject method*), 60
encode() (*Crypto.Util.asn1.DerObjectId method*), 63
encode() (*Crypto.Util.asn1.DerSequence method*), 62

encode() (*Crypto.Util.asn1.DerSetOf method*), 65
encode() (*in module Crypto.IO.PEM*), 57
encrypt_and_digest(), 26
english_to_key() (*in module Crypto.Util.RFC1751*), 66
export_key() (*Crypto.PublicKey.DSA.DsaKey method*), 46
export_key() (*Crypto.PublicKey.ECC.EccKey method*), 48
export_key() (*Crypto.PublicKey.RSA.RsaKey method*), 42
exportKey() (*Crypto.PublicKey.DSA.DsaKey method*), 45
exportKey() (*Crypto.PublicKey.RSA.RsaKey method*), 41

G

GCD() (*in module Crypto.Util.number*), 68
generate() (*in module Crypto.PublicKey.DSA*), 44
generate() (*in module Crypto.PublicKey.ECC*), 50
generate() (*in module Crypto.PublicKey.ElGamal*), 52
generate() (*in module Crypto.PublicKey.RSA*), 39
getPrime() (*in module Crypto.Util.number*), 69
getRandomInteger() (*in module Crypto.Util.number*), 69
getRandomNBitInteger() (*in module Crypto.Util.number*), 69
getRandomRange() (*in module Crypto.Util.number*), 69
getStrongPrime() (*in module Crypto.Util.number*), 69

H

has_private() (*Crypto.PublicKey.DSA.DsaKey method*), 47
has_private() (*Crypto.PublicKey.ECC.EccKey method*), 49
has_private() (*Crypto.PublicKey.ElGamal.ElGamalKey method*), 53
has_private() (*Crypto.PublicKey.RSA.RsaKey method*), 43
hasInts() (*Crypto.Util.asn1.DerSequence method*), 62
hasOnlyInts() (*Crypto.Util.asn1.DerSequence method*), 62
hexdigest(), 26
hexverify(), 26
HKDF() (*in module Crypto.Protocol.KDF*), 53

I

import_key() (*in module Crypto.PublicKey.DSA*), 47
import_key() (*in module Crypto.PublicKey.ECC*), 50
import_key() (*in module Crypto.PublicKey.RSA*), 40
inverse() (*in module Crypto.Util.number*), 70
is_point_at_infinity() (*Crypto.PublicKey.ECC.EccPoint method*), 49
isPrime() (*in module Crypto.Util.number*), 70

K

key_to_english() (*in module Crypto.Util.RFC1751*), 66

L

long_to_bytes() (*in module Crypto.Util.number*), 70

N

new() (*in module Crypto.Util.Counter*), 68

O

oid (*in module Crypto.PublicKey.RSA*), 43

P

pad() (*in module Crypto.Util.Padding*), 65
PBKDF1() (*in module Crypto.Protocol.KDF*), 53
PBKDF2() (*in module Crypto.Protocol.KDF*), 54
point_at_infinity() (*Crypto.PublicKey.ECC.EccPoint method*), 50
public_key() (*Crypto.PublicKey.ECC.EccKey method*), 49
publickey() (*Crypto.PublicKey.DSA.DsaKey method*), 47
publickey() (*Crypto.PublicKey.ElGamal.ElGamalKey method*), 53
publickey() (*Crypto.PublicKey.RSA.RsaKey method*), 43

R

RsaKey (*class in Crypto.PublicKey.RSA*), 41

S

scrypt() (*in module Crypto.Protocol.KDF*), 54
Shamir (*class in Crypto.Protocol.SecretSharing*), 55
size() (*in module Crypto.Util.number*), 70
size_in_bits() (*Crypto.PublicKey.ECC.EccPoint method*), 50
size_in_bits() (*Crypto.PublicKey.RSA.RsaKey method*), 43
size_in_bytes() (*Crypto.PublicKey.ECC.EccPoint method*), 50
size_in_bytes() (*Crypto.PublicKey.RSA.RsaKey method*), 43
split() (*Crypto.Protocol.SecretSharing.Shamir static method*), 56

strxor() (*in module Crypto.Util.strxor*), 66
strxor_c() (*in module Crypto.Util.strxor*), 66

U

unpad() (*in module Crypto.Util.Padding*), 65
UnsupportedEccFeature, 50
unwrap() (*in module Crypto.IO.PKCS8*), 59
update(), 25

V

verify(), 26

W

wrap() (*in module Crypto.IO.PKCS8*), 58