PEP272 Encryption Documentation

Simon Biewald

Oct 02, 2020

Contents:

Documentation

To prevent reinventing the wheel while creating a PEP-272 interface for a new block cipher encryption, this library aims to create an extensible framework for new libraries.

Currently following modes of operation are supported:

- ECB
- CBC
- CFB
- OFB
- CTR

The PGP mode of operation is not supported. It may be added in the future.

CHAPTER 1

Example

In this example `encrypt_aes(key, block)` will encrypt one block of AES while `decrypt_aes(key, block)` will decrypt one.

```
>>> from pep272_encryption import PEP272Cipher, MODE_ECB
>>> class AESCipher:
...     """
...     PEP-272 cipher class for AES
...     """
...     block_size = 16
...
...     def encrypt_block(self, key, block, **kwargs):
...         return encrypt_aes(key, block)
...
...     def decrypt_block(self, key, block, **kwargs):
...         return decrypt_aes(key, block)
...
>>> cipher = AESCipher(b'\00'*16, MODE_ECB)
>>> cipher.encrypt(b'\00'*16)
b'f\xe9K\xd4\xef\x8a,;\x88L\xfaY\xca4+.'
```

CHAPTER 2

License

CHAPTER 3

Structure

This documentation is structured.

It starts with the *Installation* instructions. In the second chapter there are examples and guides. Look up in the *Library reference*. The *Discussion* section explains in-depth knowledge of topics important in the context of the library to get a deeper understanding of the software.

3.1 Installation

Like the most python libraries, `pep272-encryption` can be installed with pip from PyPi:

```
$ pip install pep272_encryption
```

3.1.1 Development version

To get the newest features, it is possible to directly install the latest version from GitHub:

```
$ pip install git+https://github.com/Varbin/pep272-encryption
```

3.1.2 Supported platforms

`pep272-encryption` is pure Python with an optional C extension. They are pre-compiled for various operating systems and cpu architectures.

Python 2.7 and 3.5+ are supported and tested. The module should work on 3.3 and 3.4, but there are not automated tests for those versions.

3.2 Examples

Note: This library is for creation of *block cipher interfaces* only! It does not contain or implement any ciphers by itself.

3.2.1 Implementing a block cipher

0. The TEA block cipher

In this example, the a PEP-272 interface for the TEA block cipher is created. TEA uses 128-bit (16 bytes) keys and 64-bit (8 bytes) blocks.

In Python, an unoptimized version of TEA can be written like following:

```
import struct

def encrypt_tea(value, key, endian="!", rounds=64):
    v0, v1 = struct.unpack(endian + "2L", value)
    k = struct.unpack(endian + "4L", key)

    # mask is an uint32 helper
    delta, mask, sum = 0x9e3779b9, 0xffffffff, 0

    for i in range(rounds//2):
        sum = (sum + delta) & mask
        v0 = v0 + ( ((v1<<4) + k[0]) ^ (v1 + sum) ^ ((v1>>5) + k[1]) ) & mask
        v1 = v1 + ( ((v0<<4) + k[2]) ^ (v0 + sum) ^ ((v0>>5) + k[3]) ) & mask

    return struct.pack(endian + "2L", v0, v1)

def decrypt_tea(value, key, endian="!", rounds=64):
    v0, v1 = struct.unpack(endian + "2L", value)
    k = struct.unpack(endian + "4L", key)

    # mask is an uint32 helper
    delta, mask = 0x9e3779b9, 0xffffffff
    sum = (delta * rounds // 2) & mask

    for i in range(rounds//2):
        v1 = v1 - ( ((v0<<4) + k[2]) ^ (v0 + sum) ^ ((v0>>5) + k[3]) ) & mask
        v0 = v0 - ( ((v1<<4) + k[0]) ^ (v1 + sum) ^ ((v1>>5) + k[1]) ) & mask
        sum = (sum - delta) & mask

    return struct.pack(endian + "2L", v0, v1)
```

1. Constant definition

Import the pep272-encryption module and set module level constants, as defined per PEP-272:

```
from pep272_encryption import PEP272Cipher, MODE_ECB, MODE_CBC, \
                                            MODE_CFB, MODE_OFB, \
                                            MODE_CTR
```

(continues on next page)

(continued from previous page)

```
block_size = 8   # 64-bit blocks
key_size = 16    # 128-bit keys
```

2. The TEACipher class

Subclass the PEP272Cipher class, setting the block size parameter and override *encrypt_block* and *decrypt_block* methods:

```
class TEACipher(PEP272Cipher):
    block_size = block_size

    def encrypt_block(self, key, block, **kwargs):
        return encrypt_tea(block, key,
                           kwargs.get('endian', '!'),
                           kwargs.get('rounds', 64))

    def decrypt_block(self, key, block, **kwargs):
        return decrypt_tea(block, key,
                           kwargs.get('endian', '!'),
                           kwargs.get('rounds', 64))


def new(*args, **kwargs):
    return TEACipher(*args, **kwargs)
```

3. Complete example

Below is the full example code of all snippets combined: A PEP-272 compliant implementation of the TEA block cipher in pure python, that is interchangeable with other ciphers.

```
import struct

from pep272_encryption import PEP272Cipher
from pep272_encryption import MODE_ECB, MODE_CBC, MODE_CFB, MODE_OFB, \
                              MODE_CTR

block_size = 8   # 64-bit blocks
key_size = 16    # 128-bit keys


def new(*args, **kwargs):
    return TEACipher(*args, **kwargs)


class TEACipher(PEP272Cipher):
    block_size = block_size

    def encrypt_block(self, key, block, **kwargs):
        return encrypt_tea(block, key,
                           kwargs.get('endian', '!'),
                           kwargs.get('rounds', 64))

```

(continues on next page)

(continued from previous page)

```
    def decrypt_block(self, key, block, **kwargs):
        return decrypt_tea(block, key,
                           kwargs.get('endian', '!'),
                           kwargs.get('rounds', 64))


def encrypt_tea(value, key, endian="!", rounds=64):
    v0, v1 = struct.unpack(endian + "2L", value)
    k = struct.unpack(endian + "4L", key)

    # mask is an uint32 helper
    delta, mask, sum = 0x9e3779b9, 0xffffffff, 0

    for i in range(rounds//2):
        sum = (sum + delta) & mask
        v0 = v0 + ( ((v1<<4) + k[0]) ^ (v1 + sum) ^ ((v1>>5) + k[1]) ) & mask
        v1 = v1 + ( ((v0<<4) + k[2]) ^ (v0 + sum) ^ ((v0>>5) + k[3]) ) & mask

    return struct.pack(endian + "2L", v0, v1)


def decrypt_tea(value, key, endian="!", rounds=64):
    v0, v1 = struct.unpack(endian + "2L", value)
    k = struct.unpack(endian + "4L", key)

    # mask is an uint32 helper
    delta, mask = 0x9e3779b9, 0xffffffff
    sum = (delta * rounds // 2) & mask

    for i in range(rounds//2):
        v1 = v1 - ( ((v0<<4) + k[2]) ^ (v0 + sum) ^ ((v0>>5) + k[3]) ) & mask
        v0 = v0 - ( ((v1<<4) + k[0]) ^ (v1 + sum) ^ ((v1>>5) + k[1]) ) & mask
        sum = (sum - delta) & mask

    return struct.pack(endian + "2L", v0, v1)
```

This library can then be used easily:

```
>>> import tea
>>> cipher = tea.new(b'16-bytes key 123', mode=tea.MODE_OFB, IV=b'\00'*8)
>>> cipher.encrypt(b'123456'*6)
b"k\xaf F\xfb*\xeb\x00
↪'kP\x9c\xc9M\xb99\x1cy\xda\x99\xb1\xf0H\x14\x9c\xae@\xddxe`\x01\x85\xc9p\x85"
```

3.2.2 Implementing a custom mode of operation

In this example a "xor-encrypt-xor" mode is implemented. It takes two additional keys, that are XORed with the plain- and ciphertext. Otherwise it works like ECB.

The mode of operation is passed to the object's constructor. In this example *500* is assigned to the *MODE_XEX* constant.

`PEP272Cipher` is subclassed with the parameters *key1* and *key2*:

```python
from abc import ABC

from pep272_encryption import PEP272Cipher
from pep272_encryption.util import split_blocks, xor_strings

MODE_XEX = 500


class XEXCipher(ABC, PEP272Cipher):
    def __init__(self, key, mode, **kwargs):
        self.key1 = kwargs.pop("key1", b'\x00' * self.block_size)
        self.key2 = kwargs.pop("key2", b'\x00' * self.block_size)

        PEP272Cipher.__init__(self, key, mode, **kwargs)

    def encrypt(self, string):
        if self.mode == MODE_XEX:
            out = []
            for block in split_blocks(string, self.block_size):
                inner = xor_strings(block, self.key1)
                encrypted = self.encrypt_block(self.key, inner, **self.kwargs)
                outer = xor_strings(encrypted, self.key2)
                out.append(outer)
            return b"".join(out)
        else:
            PEP272Cipher.encrypt(self, string)

    def decrypt(self, string):
        if self.mode == MODE_XEX:
            out = []
            for block in split_blocks(string, self.block_size):
                encrypted = xor_strings(block, self.key2)
                inner = self.decrypt_block(self.key, encrypted, **self.kwargs)
                plain = xor_strings(inner, self.key1)
                out.append(plain)
            return b"".join(out)
        else:
            PEP272Cipher.decrypt(self, string)
```

3.2.3 Implementing a stream cipher

While the PEP272 Interface and this library is designed for block ciphers, it may also be used for implementing stream ciphers.

Below is an implementation of the RC4 cipher.

```python
from pep272_encryption import PEP272Cipher, MODE_ECB

block_size = 1
key_size = 0


def new(*args, **kwargs):
    return RC4Cipher(*args, **kwargs)


```

(continues on next page)

(continued from previous page)

```
class RC4Cipher(PEP272Cipher):
    block_size = 1
    key_size = 0

    def __init__(self, key, mode=MODE_ECB, **kwargs):
        if mode != MODE_ECB:
            raise ValueError("Stream ciphers only support ECB mode")

        self.S = list(range(256))
        j = 0
        for i in range(256):
            j = (j + self.S[i] + key[i % len(key)]) % 256
            self.S[i], self.S[j] = self.S[j], self.S[i]

        self.i = self.j = 0

        PEP272Cipher.__init__(self, key, mode, **kwargs)

    def encrypt_block(self, key, block, **kwargs):
        self.i = (self.i + 1) % 256
        self.j = (self.j + self.S[self.i]) % 256
        self.S[self.i], self.S[self.j] = self.S[self.j], self.S[self.i]

        K = self.S[(self.S[self.i] + self.S[self.j]) % 256]
        return bytes([block[0] ^ K])

    def decrypt_block(self, key, block, **kwargs):
        return self.encrypt_block(key, block, **kwargs)


assert RC4Cipher(b'\x01\x02\x03\x04\x05').encrypt(b'\x00'*16) \
       == b"\xb29c\x05\xf0=\xc0'\xcc\xc3RJ\n\x11\x18\xa8"
```

3.3 Library reference

This module creates PEP-272 cipher object classes for building block ciphers in python.

To use, inherit the PEP272Cipher and overwrite `encrypt_block` or `decrypt_block(self, key, string, **kwargs)` methods and set the block_size attribute.

Example:

```
class YourCipher(PEP272Cipher):
    block_size=8

    def encrypt_block(self, key, string, **kwargs):
        ...

    def decrypt_block(self, key_string, **kwargs):
        ...
```

3.3.1 The PEP272Cipher class

Subclass and overwrite the PEP272Cipher.encrypt_block and PEP272Cipher.decrypt_block methods and the block_size attribute.

class pep272_encryption.**PEP272Cipher**(*key*, *mode*, *IV=None*, ***kwargs*)

A cipher class as defined in PEP-272.

Parameters

- **key** (*bytes*) – The symmetric key to use for encryption.
- **mode** (*int*) – The mode of operation to use. For valid values see Reference/*Block cipher mode of operation*.
- **IV** (*bytes*) – A unique bytestring with once the block size in length. For security reasons it should be unpredictable and must never be used twice for the same key. Required for *CBC*, *CFB* and *OFB* mode of operation.
- ****kwargs** – See below.

Depending on the blockcipher mode of operation one or multiple of the following arguments must be passed depending on the mode of operation.

Keyword arguments

- **iv** (*bytes*) – Alternative name for **IV**
- **segment_size** (*int*): The segment size for one encryption "segment" of CFB mode in bits. It must be multiple of 8 (only byte-sized operations are allowed) and the maximum size is the block size * 8. Required for *CFB* mode of operation.
- **counter** (*callable*): A callable object returning *block size* bytes or a counter from `Crypto.Util.Counter`. For security reasons the counter output must **never** repeat. Required for *CTR* mode.
- Additional keyword arguments are passed to the underlying block cipher implementation as kwargs.

Changed in version 0.4: *IV* can be used as a positional argument.

Changed in version 0.4: PyCryptodome counters are accepted for *counter* in addition to to callables.

decrypt(*string*)

Decrypt data with the key and the parameters set at initialization.

The cipher object is stateful; decryption of a long block of data can be broken up in two or more calls to *decrypt()*. That is, the statement:

```
>>> c.decrypt(a) + c.decrypt(b)
```

is always equivalent to:

```
>>> c.decrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to decrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter in CTR returns data of the wrong length.

Returns The decrypted data, as a byte string. It is as long as *string*.

Return type bytes

decrypt_block(*key*, *block*, ***kwargs*)

Dummy function for the decryption of a single block. Overwrite with 'real' deryption function.

Parameters

- **key** (*bytes*) – The symmetric encryption key.
- **block** (*bytes*) – A single ciphertext block to encrypt.
- ****kwargs** – Additional parameters passed to *__init__*.

Raises **NotImplementedError** – This method is to be overridden.

Returns plaintext block

Return type bytes

encrypt(*string*)

Encrypt data with the key and the parameters set at initialization.

The cipher object is stateful; encryption of a long block of data can be broken up in two or more calls to *encrypt()*. That is, the statement:

```
>>> c.encrypt(a) + c.encrypt(b)
```

is always equivalent to:

```
>>> c.encrypt(a+b)
```

That also means that you cannot reuse an object for encrypting or decrypting other data with the same key.

This function does not perform any padding.

- For *MODE_ECB*, *MODE_CBC* *string* length (in bytes) must be a multiple of *block_size*.
- For *MODE_CFB*, *string* length (in bytes) must be a multiple of *segment_size*/8.
- For *MODE_CTR* and *MODE_OFB*, *string* can be of any length.

Parameters **string** (*bytes*) – The piece of data to encrypt.

Raises

- **ValueError** – When a mode of operation has be requested this code cannot handle.
- **ValueError** – When len(string) has a wrong length, as described above.
- **TypeError** – When the counter callable in CTR returns data with the wrong length.

Returns The encrypted data, as a byte string. It is as long as *string*.

Return type bytes

encrypt_block(*key*, *block*, ***kwargs*)

Dummy function for the encryption of a single block. Overwrite with 'real' encryption function.

Parameters

- **key** (*bytes*) – The symmetric encryption key.
- **block** (*bytes*) – A single plaintext block to encrypt.
- ****kwargs** – Additional parameters passed to *__init__*.

Raises **NotImplementedError** – This method is to be overridden.

Returns ciphertext block

Return type bytes

3.3.2 Block cipher mode of operation

Note: For mote details about different modes of operation, see Discussions/*Block cipher mode of operation*.

Block ciphers can be used in different modes of operation. The mode of operation can be set by passing one of the constants to the cipher object. Different modes of operation may require to pass extra arguments to the constructor.

Below is an example from the mostly PEP-272 compliant PyCryptodome.

```
>>> from Crypto.Cipher import AES
>>> iv = b'random 16 bytes!'
>>> key = b'0123456789abcdef'
>>> cipher = AES.new(key, mode=AES.MODE_CBC, IV=iv)
>>> cipher.encrypt(b'\00'*16)
b'j\xa2\xb5\x80\xf7\xbd\xb4I\xda\xea\x9aN\x9d\xb5\x9a\x17'
```

This library supports following modes of operation:

- Electronic code book (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

The CFB variant of PGP is not supported.

Planned modes are (extending to PEP-272):

- Propagating Cipher Block Chaining (PCBC), used in older Kerberos versions
- Infinite Garble Extension (IGE), used by Telegram.
- OpenPGP mode, compatible to PyCrypto or PyCryptodome.

Authenticated encryption (AE) or authenticated encryption with associated data (AEAD) are currently not supported, as they would require additional methods to finalize the encryption and sometimes have special requirements.

Constant	Number	Source	Implemented	Specification
MODE_ECB	1	PEP-272	Yes	NIST.SP.800-38A
MODE_CBC	2	PEP-272	Yes	NIST.SP.800-38A
MODE_CFB	3	PEP-272	Yes	NIST.SP.800-38A
MODE_PGP	4	PEP-272	No	RFC 4880
MODE_OFB	5	PEP-272	Yes	NIST.SP.800-38A
MODE_CTR	6	PEP-272	Yes	NIST.SP.800-38A
MODE_OPENPGP	7	PyCrypto	No	RFC 4880
MODE_XTS	**8**	PyCryptoPlus	No	IEEE P1619 and NIST.SP.800-38E
MODE_CCM	**8**	PyCrypto (unreleased) / PyCryptodome	No	NIST.SP.800-38C
MODE_EAX	9	PyCrypto (unreleased) / PyCryptodome	No	The EAX Mode of Operation
MODE_SIV	10	PyCrypto (unreleased) / PyCryptodome	No	RFC 5297
MODE_GCM	11	PyCrypto (unreleased) / PyCryptodome	No	NIST.SP.800-38D
MODE_OCB	12	PyCrypto (unreleased) / PyCryptodome	No	RFC 7253

3.3.3 Utility functions

Utility library for compatibility with Python 2 and 3. A counter to use with CTR is also included.

There are versions of `chr` and `ord`-methods to work with bytes with Python 3 and strings with Python 2.

class `pep272_encryption.util.`**`Counter`**(*nonce=None*, *initial_value=0*, *suffix=None*, *iv=None*, *IV=None*, *block_size=None*, *endian='big'*, *wrap_around=False*)

Counter for usage in CTR mode.

Big endian is as assumed for all counter operations by default.

Parameters

- **nonce** (*bytes*) – Prefix for counter operations.
- **initial_value** (*int*) – Initial integer value.
- **suffix** (*bytes*) – Suffix to add after output.
- **iv** (*bytes*) – Counter output to resume. The usage of *iv* prohibits the use of *nonce*, *initial_value*, *block_size* and *suffix*.
- **IV** (*bytes*) – Alternative for *iv*.
- **block_size** (*int*) – Size of counter in-/output
- **endian** (*str*) – Endian for number/byte conversions.
- **wrap_around** (*bool*) – If an exception should not be raised if the counter returns the same value twice. For security reasons, setting this value to true is not recommended.

The counter is not thread safe.

Without arguments, it generates a random nonce, with the counter starts at 0:

```
>>> c = Counter()  # random nonce
>>> c().endswith(b"\x00")
True
>>> c().endswith(b"\x01")
True
>>> c().endswith(b"\x02")
True
```

Alternatively, a nonce and an initial value can be set:

```
>>> c = Counter(nonce=b'\x00\x01\x02', initial_value=0xff01,
...                   block_size=8)
>>> c()
b'\x00\x01\x02\x00\x00\x00\xff\x01'
>>> c()
b'\x00\x01\x02\x00\x00\x00\xff\x02'
```

The third alternative is to give a full start string. Counter length is determined by the IV length:

```
>>> c = Counter(IV=b'\x00'*4, endian="little")
>>> c()
b'\x00\x00\x00\x00'
>>> c()
b'\x01\x00\x00\x00'
```

block_size = 16

Used by many algorithms

pep272_encryption.util.**b_chr**(*ordinal*)

Return a byte string of one character with 0 <= ordinal <= 255.

Parameters **ordinal** (*int*) – The Unicode code point of a single char

Returns The byte string representation of the ordinal

Return type bytes

pep272_encryption.util.**b_ord**(*byte*)

Return the Unicode code point for a byte or iteration product of a byte string alike object (e.g. bytearray).

Parameters **byte** (*bytes or str or int*) – The single byte or iteration product of a byte string to convert

Returns Unicode code point

Return type int

pep272_encryption.util.**from_bytes**(*bytestring*, *byteorder*)

Convert a bytestring to an interger.

Parameters

- **bytestring** (*bytes*) – The byte string to convert.
- **byteorder** (*str*) – either 'big' or 'little'

Return type int

pep272_encryption.util.**split_blocks**(*bytestring*, *block_size*)

Splits bytestring in block_size-sized blocks.

Raises an error if len(string) % blocksize != 0.

`pep272_encryption.util.`**`to_bytes`**(*integer*, *length*, *byteorder*)

Convert an integer to a bytestring.

Parameters

- **integer** (*int*) – The integer to convert.
- **length** (*int*) – Length of the created byte string.
- **byteorder** (*str*) – either 'big' or 'little'.

Return type bytes

`pep272_encryption.util.`**`xor_strings`**(*one*, *two*)

xor two bytestrings together.

Parameters

- **one** (*bytes*) – First string
- **two** (*bytes*) – Second string

Returns The xored strings

Return type bytes

3.4 Discussion

This section provides in-depth knowledge of topics important in the context of the library.

3.4.1 Block cipher mode of operation

Block ciphers can be used in different modes of operation. The mode of operation can be set by passing one of the constants to the cipher object. Different modes of operation may require to pass extra arguments to the constructor.

PEP-272 requires libraries to provide at least the most common modes: `ECB`, `CBC`, `CFB`, `OFB` and `CTR`, all of those are supported by this library.

Each mode of operation is different and has different requirements.

> **Warning:** All of the supported modes of operating do not protect the integrity of encrypted data!

Electronic Code Book Mode (ECB)

The ECB mode of operation is the simplest one - each plaintext block is independently encrypted. The resulting problem is that the same plaintext leads to the same ciphertext, every time they occur. This means ECB mode is not semantically secure.

Plain- / ciphertexts must be multiple of blocksize in length.

The formulae for the ECB mode are:

$$C_i = E_K(P_i)$$
$$P_i = D_K(C_i)$$

Fig. 1: ECB encryption

Fig. 2: ECB decryption

Attacks against ECB mode

Because all plaintext blocks are encrypted independently, an encryption of the same block results in the same ciphertext block each time.

This means by having multiple ciphertexts in can be concluded whether the correspondent plaintexts are the same or not.

The multiple repetition of plaintext blocks may result in visible repetitions in the ciphertext, e.g. in images.

Fig. 3: Plain Tux image

Cipher Block Chaining Mode (CBC)

To solve the problems of the ECB mode, a plaintext block is xored to the previous ciphertext block. For the very "first" ciphertext an **initialization vector (IV)** is used. The IV can be considered public information.

Plain- / ciphertexts must be multiple of blocksize in length.

Having an incorrect block or IV will result in an incorrect decryption of the direct adjectant block, but the remaining blocks will remain intact.

The formulae for en- and decryption are:

$$\begin{aligned} C_i &= E_K(P_i \oplus C_{i-1}) \\ P_i &= D_K(C_i) \oplus C_{i-1} \\ C_0 &= \mathrm{IV} \end{aligned}$$

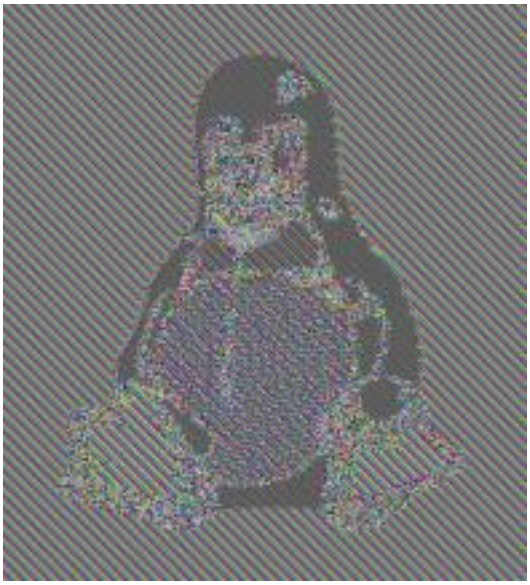

Fig. 4: Encrypted Tux image in ECB mode

Fig. 5: CBC encryption

Attacks against CBC mode

A one-bit change to the ciphertext causes complete corruption of the corresponding block of plaintext, and the inversion of the corresponding bit in the next block while leaving the rest iof the blocks intact. This can lead to padding oracle attacks such as POODLE (it is the consequence solely of the choice of CBC mode but other design choices, too).

Watermarking attacks are possible with predictable IVs.

Cipher Feedback Mode (CFB)

The CFB mode of operation makes a stream cipher out of the block cipher. The block size of the cipher is reduced to `segment_size`.

Plain- and ciphertext must be a multiple of `segment_size` in length.

The formulae describing CFB mode are:

$$\begin{aligned} C_i &= E_K(C_{i-1}) \oplus P_i \\ P_i &= E_K(C_{i-1}) \oplus C_i \\ C_0 &= \mathrm{IV} \end{aligned}$$

Fig. 6: CBC decryption

Fig. 7: CFB encryption

Fig. 8: CFB decryption

Output Feedback (OFB)

OFB mode creates a stream cipher by XORing the plain text with a keystream generated by encrypting a stream of null bytes in CBC mode. Encryption and decryption are the same, data of arbitrary length can be processed.

The initialization vector **must** be unique, or there will be a catastrophic cryptographic failure.

The formulae describing OFB mode of operation are:

$$\begin{aligned} C_i &= P_i \oplus O_i \\ P_i &= C_i \oplus O_i \\ O_i &= E_K(O_{i-1} \oplus O_i) \\ O_0 &= \mathrm{IV} \end{aligned}$$

Fig. 9: OFB encryption

Counter mode of operation

CTR mode creates a stream cipher by XORing the plain text with a keystream generated by encrypting a counter. Encryption and decryption are the same, data of arbitrary length can be processed.

The counter **must never** repeat, or there will be a catastrophic cryptographic failure.

CTR can be described with those formulae:

$$\begin{aligned} C_i &= P_i \oplus O_i \\ P_i &= C_i \oplus O_i \end{aligned}$$

where O_i are the return values of the counter.

3.4.2 Speed

As a large part of this library is written in pure python, it is slower than C implementations, like PyCrypto and its successor PyCryptodome.

The presence of the optional C extension significantly improves the speed of this library.

3.5 Acknowledgements and Sources

3.5.1 Images

Fig. 10: OFB decryption

Fig. 11: CTR encryption

Fig. 12: CTR decryption

Fig. 13: Tux image

Tux

Tux image (and derivates) by Larry Erwing made with The GIMP under following conditions: "*Permission to use and/or modify this image is granted provided you acknowledge me lewing@isc.tamu.edu and The GIMP if someone asks.*"

Graphic representations of block cipher modes of operation

Fig. 14: One of the schemata used

Those can be found on Wikipedia/Block cipher mode of operation and are made by different authors in public domain.

3.5.2 Sources

Block cipher mode of operation: Wikipedia/Block cipher mode of operation

"TEA, a Tiny Encryption Algorithm": Original Paper

CHAPTER 4

Changelog

4.1 0.4

4.1.1 New

- Type hints
- Build manylinux wheels, universal wheel is limited to Python 2 and PyPy
- In addition to callables, the *counter* argument for CTR mode now accepts counters from PyCryptodome now

4.1.2 Changed

- Extension module is in pure C, instead of being written in Cython
- *__init__* signature is slightly different: *IV* can be given as a positional argument

4.2 0.3 - 2019-06-14

4.2.1 Added

- `PEP272Cipher` is a new style class on Python 2.
- Documentation
- Optional extensions module for more speed. CBC and CFB are now two times faster!

4.2.2 Changed

- `PEP272Cipher.IV` does not change in mode using a CBC

4.3 0.1 - 2019-03-03

- Initial release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index