ECPy Documentation

Release 0.8

Cédric Mesnil

September 27, 2016

Contents

1 Status **1**

2 Install **3**

3 Overview **5**
3.1 Supported Curves & Signature 5
3.2 Types 6

4 API **7**
4.1 curves module 7
4.2 keys module 11
4.3 ECDSA module 12
4.4 EDDSA module 12
4.5 ECSchnorr module 13
4.6 Borromean module 15
4.7 ecrand module 16
4.8 formatters module 17

5 Indices and tables **19**

Python Module Index **21**

Status

ECPy is in beta stage but already used in some internal tooling.
Any constructive comment is welcome.

Version 0.8

Authors Cedric Mesnil, <cedric.mesnil@ubinity.com>

CHAPTER 2

Install

ECPy is originally coded for Python 3, but run under python 2.7 (and maybe 2.6) by using *future*. If you run Python 2, please install the future into the present:

    pip install future

Then install ECPy:

- **Rebuild from git clone:**
  - python3 setup.py sdist
  - cd dist
  - tar xzvf ECPy-M.m.tar.gz
  - python3 setup install
- **Install from dist package:**
  - Download last dist tarball
  - tar xzvf ECPy-M.m.tar.gz
  - python3 setup.py install

Overview

ECPy (pronounced ekpy), is a pure python Elliptic Curve library. It provides ECDSA, EDDSA, ECSchnorr signature as well as Point operation.

ECDSA sample

```
from ecpy.curves     import Curve,Point
from ecpy.keys       import ECPublicKey, ECPrivateKey
from ecpy.ecdsa      import ECDSA

cv     = Curve.get_curve('secp256k1')
pu_key = ECPublicKey(Point(0x65d5b8bf9ab1801c9f168d4815994ad35f1dcb6ae6c7a1a303966b677b813b00,

                           0xe6b865e529b8ecbf71cf966e900477d49ced5846d7662dd2dd11ccd55c0aff7f,
                           cv))
pv_key = ECPrivateKey(0xfb26a4e75eec75544c0f44e937dcf5ee6355c7176600b9688c667e5c283b43c5,
                      cv)


signer = ECDSA()
sig    = signer.sign(b'01234567890123456789012345678912',pv_key)
assert(signer.verify(b'01234567890123456789012345678912',sig,pu_key))
```

Point sample

```
from ecpy.curves     import Curve,Point

cv = Curve.get_curve('secp256k1')
P  = Point(0x65d5b8bf9ab1801c9f168d4815994ad35f1dcb6ae6c7a1a303966b677b813b00,
           0xe6b865e529b8ecbf71cf966e900477d49ced5846d7662dd2dd11ccd55c0aff7f,
           cv)
k  = 0xfb26a4e75eec75544c0f44e937dcf5ee6355c7176600b9688c667e5c283b43c5
Q  = k*P
R  = P+Q
```

3.1 Supported Curves & Signature

ECPy support the following curves

- Short Weierstrass form: $y^2=x^3+a*x+b$
- Twisted Edward $a*x^2+y2=1+d*x^2*y^2$

See pyec.Curve. get_curve_names

ECPy supports the following

3.2 Types

ECPY use binary *bytes* and *int* as primary types.

int are used when scalar is required, as for point coordinate, scalar multiplication, ...
bytes are used when data is required, as hash value, message, ...

Other main types are `Point`, `Curve`, `Key`, `ECDSA`, `EDDSA`, `ECSchnorr`. `Borromean`.

See API details...

CHAPTER 4

API

4.1 curves module

Elliptic Curve and Point manipulation

class `ecpy.curves.`**`Curve`**(*parameters*)

Bases: `object`

Elliptic Curve abstraction

You should not directly create such Object. Use *get_curve* to get the predefined curve or create a well-know type of curve with your parameters

Supported well know elliptic curve are:

- Short Weierstrass form: $y^2=x^3+a*x+b$
- Twisted Edward $a*x^2+y2=1+d*x^2*y^2$

`name`

str

curve name, the one given to get_curve or return by get_curve_names

`size`

int

bit size of curve

`a`

int

first curve parameter

`b d`

int

second curve parameter

`field`

int

curve field

`generator`

Point

curve point generator

order
int

order of generator

add_point (*P*, *Q*)
Returns the sum of P and Q

This function ignores the default curve attach to P and Q, and assumes P and Q are on this curve.

Parameters

- **P** (`Point`) – first point to add
- **Q** (`Point`) – second point to add

Returns A new Point R = P+Q

Return type *Point*

Raises `ECPyException` – with "Point not on curve", if Point R is not on curve, thus meaning either P or Q was not on.

decode_point (*eP*)
decode/decompress a point according to its curve

encode_point (*P*)
encode/compress a point according to its curve

static **get_curve** (*name*)
Return a Curve object according to its name

Parameters **name** (*str*) – curve name to retrieve

Returns Curve object

Return type *Curve*

static **get_curve_names** ()
Returns all known curve names

Returns list of names as str

Return type tuple

is_on_curve (*P*)
Check if P is on this curve

This function ignores the default curve attach to P

Parameters **P** (`Point`) – Point to check

Returns True if P is on curve, False else

Return type bool

mul_point (*k*, *P*)
Returns the scalar multiplication P with k.

This function ignores the default curve attach to P and Q, and assumes P and Q are on this curve.

Parameters

- **P** (`Point`) – point to mul_point
- **k** (*int*) – scalar to multiply

Returns A new Point R = k*Q

Return type *Point*

Raises

- `ECPyException` – with "Point not on curve", if Point R is not
- on curve, thus meaning P was not on.

sub_point (*P*, *Q*)

Returns the difference of P and Q

This function ignores the default curve attach to P and Q, and assumes P and Q are on this curve.

Parameters

- **P** (`Point`) – first point to subtract with
- **Q** (`Point`) – second point to subtract to

Returns A new Point R = P-Q

Return type *Point*

Raises `ECPyException` – with "Point not on curve", if Point R is not on curve, thus meaning either P or Q was not on.

class `ecpy.curves.`**Point** (*x*, *y*, *curve*, *check=True*)

Bases: `object`

Immutable Elliptic Curve Point.

A Point support the following operator:

- \+ : Point Addition, with automatic doubling support.
- * : Scalar multiplication, can write as k*P or P*k, with P :class:Point and k :class:int
- ==: Point comparison

x

int

Affine x coordinate

y

int

Affine y coordinate

curve

Curve

Curve on which the point is define

Parameters

- **x** (*int*) – x coordinate
- **y** (*int*) – y coordinate
- **check** (*bool*) – if True enforce x,y is on curve

Raises `ECPyException` – if check=True and x,y is not on curve

class `ecpy.curves.`**TwistedEdwardCurve** (*domain*)

Bases: *ecpy.curves.Curve*

An elliptic curve defined by the equation: $a*x^2+y^2=1+d*x^2*y^2$

The given domain must be a dictionary providing the following keys/values:

- name (str) : curve unique name
- size (int) : bit size
- a (int) : *a* equation coefficient
- d (int) : *b* equation coefficient
- field (inf) : field value
- generator (int[2]) : x,y coordinate of generator
- order (int) : order of generator

Parameters **domain** (`dict`) – a dictionary providing curve domain parameters

add_point (*P*, *Q*)
See `Curve.add_point()`

decode_point (*eP*)
Decodes a point P according to *draft_irtf-cfrg-eddsa-04*.

Parameters

- **eP** (`bytes`) – encoded point
- **curve** (`Curve`) – curve on witch point is

Returns Point : decoded point

encode_point (*P*)
Encodes a point P according to *draft_irtf-cfrg-eddsa-04*.

Parameters **P** – point to encode

Returns bytes : encoded point

is_on_curve (*P*)
See `Curve.is_on_curve()`

mul_point (*k*, *P*)
See `Curve.add_point()`

x_recover (*y*, *sign=0*)
Retrieves the x coordinate according to the y one, such that point (x,y) is on curve.

Parameters

- **y** (`int`) – y coordinate
- **sign** (`int`) – sign of x

Returns the computed x coordinate

Return type int

class ecpy.curves.**WeierstrassCurve** (*domain*)
Bases: `ecpy.curves.Curve`

An elliptic curve defined by the equation: $y^2=x^3+a*x+b$.

The given domain must be a dictionary providing the following keys/values:

- name (str) : curve unique name
- size (int) : bit size
- a (int) : *a* equation coefficient
- b (int) : *b* equation coefficient
- field (inf) : field value
- generator (int[2]) : x,y coordinate of generator
- order (int) : order of generator
- cofactor (int) : cofactor

Parameters **domain** (*dict*) – a dictionary providing curve parameters

add_point (*P*, *Q*)
See *Curve.add_point()*

is_on_curve (*P*)
See *Curve.is_on_curve()*

mul_point (*k*, *P*)
See *Curve.mul_point()*

4.2 keys module

class ecpy.keys.**ECPrivateKey** (*d*, *curve*)
Bases: object

Public EC key.

Can be used for both ECDSA and EDDSA signature

Attributes d (int) : private key scalar curve (Curve) : curve

Parameters

- **d** (*int*) – private key value
- **curve** (Curve) – curve

get_public_key ()
Returns the public key corresponding to this private key

Returns public key

Return type *ECPublicKey*

class ecpy.keys.**ECPublicKey** (*W*)
Bases: object

Public EC key.

Can be used for both ECDSA and EDDSA signature

W

Point

public key point

Parameters **W** (Point) – public key value

4.3 ECDSA module

class ecpy.ecdsa.**ECDSA** (*fmt='DER'*)

Bases: object

ECDSA signer.

Parameters **fmt** (*str*) – in/out signature format. See *ecpy.formatters*

sign (*msg*, *pv_key*)

Signs a message hash.

Parameters

- **msg** (*bytes*) – the message hash to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing

sign_k (*msg*, *pv_key*, *k*)

Signs a message hash with provided random

Parameters

- **msg** (*bytes*) – the hash of message to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing
- **k** (ecpy.keys.ECPrivateKey) – random to use for signing

sign_rfc6979 (*msg*, *pv_key*, *hasher*)

Signs a message hash according to RFC6979

Parameters

- **msg** (*bytes*) – the message hash to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing
- **hasher** (*hashlib*) – hasher conform to hashlib interface

verify (*msg*, *sig*, *pu_key*)

Verifies a message signature.

Parameters

- **msg** (*bytes*) – the message hash to verify the signature
- **sig** (*bytes*) – signature to verify
- **pu_key** (ecpy.keys.ECPublicKey) – key to use for verifying

4.4 EDDSA module

class ecpy.eddsa.**EDDSA** (*hasher*, *fmt='EDDSA'*)

Bases: object

EDDSA signer implemenation according to:

- IETF draft-irtf-cfrg-eddsa-05.

Parameters

- **hasher** (*hashlib*) – callable constructor returning an object with update(), digest() interface. Example: hashlib.sha256, hashlib.sha512...
- **fmt** (*str*) – in/out signature format. See *ecpy.formatters*.

sign (*msg*, *pv_key*)
Signs a message.

Parameters

- **msg** (*bytes*) – the message to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing

verify (*msg*, *sig*, *pu_key*)
Verifies a message signature.

Parameters

- **msg** (*bytes*) – the message to verify the signature
- **sig** (*bytes*) – signature to verify
- **pu_key** (ecpy.keys.ECPublicKey) – key to use for verifying

4.5 ECSchnorr module

class ecpy.ecschnorr.**ECSchnorr** (*hasher*, *option='ISO'*, *fmt='DER'*)
Bases: object

ECSchnorr signer implementation according to:

- BSI:TR03111
- ISO/IEC:14888-3
- bitcoin-core:libsecp256k1

In order to select the specification to be conform to, choose the corresponding string option: "BSI", "ISO", "ISOx", "LIBSECP"

Signature:

- **"BSI": compute r,s according to to BSI :**
  1. k = RNG(1:n-1)
  2. Q = [k]G
  3. r = H(M ||Qx) If r = 0 mod n, goto 1.
  4. s = k - r.d mod n If s = 0 goto 1.
  5. Output (r, s)
- **"ISO": compute r,s according to ISO :**
  1. k = RNG(1:n-1)
  2. Q = [k]G If r = 0 mod n, goto 1.
  3. r = H(Qx||Qy||M).
  4. s = (k + r.d) mod n If s = 0 goto 1.

5. Output (r, s)

- **"ISOx": compute r,s according to optimized ISO variant:**
  1. k = RNG(1:n-1)
  2. Q = [k]G If r = 0 mod n, goto 1.
  3. r = H(Qx||Qy||M).
  4. s = (k + r.d) mod n If s = 0 goto 1.
  5. Output (r, s)
- **"LIBSECP": compute r,s according to bitcoin lib:**
  1. k = RNG(1:n-1)
  2. Q = [k]G if Qy is odd, negate k and goto 2
  3. r = Qx % n
  4. h = H(r || m). if h == 0 or h >= order goto 1
  5. s = k - h.d.
  6. Output (r, s)

Verification

- **"BSI": verify r,s according to to BSI :**
  1. Verify that r in {0, . . . , 2**t - 1} and s in {1, 2, . . . , n - 1}. If the check fails, output False and terminate.
  2. Q = [s]G + [r]W If Q = 0, output Error and terminate.
  3. v = H(M||Qx)
  4. Output True if v = r, and False otherwise.
- **"ISO": verify r,s according to ISO :**
  1. check...
  2. Q = [s]G - [r]W If Q = 0, output Error and terminate.
  3. v = H(Qx||Qy||M).
  4. Output True if v = r, and False otherwise.
- **"ISOx": verify r,s according to optimized ISO variant:**
  1. check...
  2. Q = [s]G - [r]W If Q = 0, output Error and terminate.
  3. v = H(Qx||M).
  4. Output True if v = r, and False otherwise.
- **"LIBSECP":**
  1. Signature is invalid if s >= order. Signature is invalid if r >= p.
  2. h = H(r || m). Signature is invalid if h == 0 or h >= order.
  3. R = [h]Q + [s]G. Signature is invalid if R is infinity or R's y coordinate is odd.
  4. Signature is valid if the serialization of R's x coordinate equals r.

Default is "ISO"

Parameters

- **hasher** (*hashlib*) – callable constructor returning an object with update(), digest() interface. Example: hashlib.sha256, hashlib.sha512...
- **option** (*str*) – one of "BSI","ISO","ISOx","LIBSECP"
- **fmt** (*str*) – in/out signature format. See *ecpy.formatters*

sign (*msg*, *pv_key*)
Signs a message hash.

Parameters

- **hash_msg** (*bytes*) – the message hash to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing

sign_k (*msg*, *pv_key*, *k*)
Signs a message hash with provided random

Parameters

- **hash_msg** (*bytes*) – the message hash to sign
- **pv_key** (ecpy.keys.ECPrivateKey) – key to use for signing
- **k** (ecpy.keys.ECPrivateKey) – random to use for signing

verify (*msg*, *sig*, *pu_key*)
Verifies a message signature.

Parameters

- **msg** (*bytes*) – the message hash to verify the signature
- **sig** (*bytes*) – signature to verify
- **pu_key** (ecpy.keys.ECPublicKey) – key to use for verifying

4.6 Borromean module

class ecpy.borromean.**Borromean** (*fmt='BTUPLE'*)
Bases: object

Borromean Ring signer implementation according to:

https://github.com/Blockstream/borromean_paper/blob/master/borromean_draft_0.01_9ade1e49.pdf

https://github.com/ElementsProject/secp256k1-zkp/blob/secp256k1-zkp/src/modules/rangeproof/borromean_impl.h

ElementsProject implementation has some tweaks compared to PDF. This implementation is ElementsProject compliant.

For now, only secp256k1+sha256 is supported. This constraint will be release soon.

Parameters **fmt** (*str*) – in/out signature format. See *ecpy.formatters*. IGNORED.

sign (*msg*, *rings*, *pv_keys*, *pv_keys_index*)
Signs a message hash.

The public *rings* argument is a tuple of public key array. In other words each element of the ring tuple is an array containing the public keys list of that ring

A Private key must be given for each provided ring. For each private key, the corresponding public key is specified by its index in the ring.

Exemple: let r1 be the first ring with 2 keys: pu11, pu12 let 21 be the second ring with 3 keys: pu21,pu22,pu23 let say we want to produce a signature with sec12 and sec21 *sign* should be called as:

```
borromean.sign(m,
               ([pu11,pu12],[pu21,pu22,pu23]),
               [sec12, sec21], [1,0])
```

The return value is a tuple (e0, [s0,s1....]). Each value is encoded as binary (bytes).

Parameters

- **msg** (*bytes*) – the message hash to sign
- **rings** (*tuple of (ecpy.keys.ECPublicKey[]*) – public key rings
- **pv_keys** (*ecpy.keys.ECPrivateKey[]*) – key to use for signing
- **pv_keys_index** (*int[]*) –

Returns signature

Return type (e0, [s0,s1....])

verify(*msg*, *sig*, *rings*)
Verifies a message signature.

Parameters

- **msg** (*bytes*) – the message hash to verify the signature
- **sig** (*bytes*) – signature to verify
- **rings** (*key.ECPublicKey*) – key to use for verifying

Returns True if signature is verified, False else

Return type boolean

4.7 ecrand module

ecpy.ecrand.**rnd**(*q*)
Returns a random number less than q, with the same bits length than q

Parameters **q** (*int*) – field/modulo

Returns random

Return type int

ecpy.ecrand.**rnd_rfc6979**(*hashmsg*, *secret*, *q*, *hasher*, *V=None*)
Generates a deterministic *value* according to RF6979.

See https://tools.ietf.org/html/rfc6979#section-3.2

if V == None, this is the first try, so compute the initial value for V. Else it means the previous value has been rejected by the caller, so generate the next one!

Warning: the *hashmsg* parameter is the message hash, not the message itself. In other words, *hashmsg* is equal to *h1* in the *rfc6979, section-3.2, step a*.

Parameters

- **hasher** (*hashlib*) – hasher
- **hashmsg** (*bytes*) – message hash
- **secret** (*int*) – secret
- **q** (*int*) – field/modulo
- **V** – previous value for continuation

The function returns a couple *(k,V)* with *k* the expected value and *V* is the continuation value to pass to next cal if k is rejected.

Returns (k,V)

Return type tuple

4.8 formatters module

`ecpy.formatters.`**`decode_sig`**(*sig*, *fmt='DER'*)
encore signature according format

Parameters

- **rs** (*bytes, ints, tuple*) – r,s value
- **fmt** (*str*) – 'DER'|'BTUPLE'|'ITUPLES'|'RAW'|'EDDSA

Returns (r,s)

Return type ints

`ecpy.formatters.`**`encode_sig`**(*r*, *s*, *fmt='DER'*, *size=0*)
encore signature according format

Parameters

- **r** (*int*) – r value
- **s** (*int*) – s value
- **fmt** (*str*) – 'DER'|'BTUPLE'|'ITUPLE'|'RAW'|'EDDSA

Returns TLV for DER encoding

Return type bytes

Returns (r,s) for BTUPLE encoding

Return type bytes

Returns (r,s) for ITUPLE encoding

Return type ints

Returns r|s for RAW encoding

Return type bytes

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

e

ecpy.borromean, 15
ecpy.curves, 7
ecpy.ecdsa, 12
ecpy.ecrand, 16
ecpy.ecschnorr, 13
ecpy.eddsa, 12
ecpy.formatters, 17
ecpy.keys, 11

A

a (ecpy.curves.Curve attribute), 7
add_point() (ecpy.curves.Curve method), 8
add_point() (ecpy.curves.TwistedEdwardCurve method), 10
add_point() (ecpy.curves.WeierstrassCurve method), 11

B

Borromean (class in ecpy.borromean), 15

C

Curve (class in ecpy.curves), 7
curve (ecpy.curves.Point attribute), 9

D

decode_point() (ecpy.curves.Curve method), 8
decode_point() (ecpy.curves.TwistedEdwardCurve method), 10
decode_sig() (in module ecpy.formatters), 17

E

ECDSA (class in ecpy.ecdsa), 12
ECPrivateKey (class in ecpy.keys), 11
ECPublicKey (class in ecpy.keys), 11
ecpy.borromean (module), 15
ecpy.curves (module), 7
ecpy.ecdsa (module), 12
ecpy.ecrand (module), 16
ecpy.ecschnorr (module), 13
ecpy.eddsa (module), 12
ecpy.formatters (module), 17
ecpy.keys (module), 11
ECSchnorr (class in ecpy.ecschnorr), 13
EDDSA (class in ecpy.eddsa), 12
encode_point() (ecpy.curves.Curve method), 8
encode_point() (ecpy.curves.TwistedEdwardCurve method), 10
encode_sig() (in module ecpy.formatters), 17

F

field (ecpy.curves.Curve attribute), 7

G

generator (ecpy.curves.Curve attribute), 7
get_curve() (ecpy.curves.Curve static method), 8
get_curve_names() (ecpy.curves.Curve static method), 8
get_public_key() (ecpy.keys.ECPrivateKey method), 11

I

is_on_curve() (ecpy.curves.Curve method), 8
is_on_curve() (ecpy.curves.TwistedEdwardCurve method), 10
is_on_curve() (ecpy.curves.WeierstrassCurve method), 11

M

mul_point() (ecpy.curves.Curve method), 8
mul_point() (ecpy.curves.TwistedEdwardCurve method), 10
mul_point() (ecpy.curves.WeierstrassCurve method), 11

N

name (ecpy.curves.Curve attribute), 7

O

order (ecpy.curves.Curve attribute), 7

P

Point (class in ecpy.curves), 9

R

rnd() (in module ecpy.ecrand), 16
rnd_rfc6979() (in module ecpy.ecrand), 16

S

sign() (ecpy.borromean.Borromean method), 15
sign() (ecpy.ecdsa.ECDSA method), 12
sign() (ecpy.ecschnorr.ECSchnorr method), 15
sign() (ecpy.eddsa.EDDSA method), 13

sign_k() (ecpy.ecdsa.ECDSA method), 12
sign_k() (ecpy.ecschnorr.ECSchnorr method), 15
sign_rfc6979() (ecpy.ecdsa.ECDSA method), 12
size (ecpy.curves.Curve attribute), 7
sub_point() (ecpy.curves.Curve method), 9

T

TwistedEdwardCurve (class in ecpy.curves), 9

V

verify() (ecpy.borromean.Borromean method), 16
verify() (ecpy.ecdsa.ECDSA method), 12
verify() (ecpy.ecschnorr.ECSchnorr method), 15
verify() (ecpy.eddsa.EDDSA method), 13

W

W (ecpy.keys.ECPublicKey attribute), 11
WeierstrassCurve (class in ecpy.curves), 10

X

x (ecpy.curves.Point attribute), 9
x_recover() (ecpy.curves.TwistedEdwardCurve method), 10

Y

y (ecpy.curves.Point attribute), 9