fastecdsa Documentation

Release 2.0.0

Anton Kueltz

Feb 20, 2020

Contents:

CHAPTER 1

Installation

The only actively supported operating systems at this time are most Linux distros and OS X.

You can use pip: `$ pip install fastecdsa` or clone the repo and use `$ python setup.py install`. Note that you need to have a C compiler (you can check this via e.g. `$ which gcc` or `$ which clang`). You also need to have GMP on your system as the underlying C code in this package includes the `gmp.h` header (and links against gmp via the `-lgmp` flag).

1.1 Installing Dependencies

1.1.1 Ubuntu / Debian

```
$ sudo apt-get install gcc python-dev libgmp3-dev
```

1.1.2 RHEL / CentOS

```
$ sudo yum install gcc python-devel gmp-devel
```

CHAPTER 2

fastecdsa

2.1 fastecdsa.curve

class fastecdsa.curve.**Curve**(*name: str*, *p: int*, *a: int*, *b: int*, *q: int*, *gx: int*, *gy: int*, *oid: bytes = None*)

Bases: `object`

Representation of an elliptic curve.

Defines a group for the arithmetic operations of point addition and scalar multiplication. Currently only curves defined via the equation $y^2 \equiv x^3 + ax + b \pmod{p}$ are supported.

Attributes:

name (str): The name of the curve

p (int): The value of p in the curve equation.

a (int): The value of a in the curve equation.

b (int): The value of b in the curve equation.

q (int): The order of the base point of the curve.

oid (bytes): The object identifier of the curve.

G

The base point of the curve.

For the purposes of ECDSA this point is multiplied by a private key to obtain the corresponding public key. Make a property to avoid cyclic dependency of Point on Curve (a point lies on a curve) and Curve on Point (curves have a base point).

__init__(*name: str*, *p: int*, *a: int*, *b: int*, *q: int*, *gx: int*, *gy: int*, *oid: bytes = None*)

Initialize the parameters of an elliptic curve.

WARNING: Do not generate your own parameters unless you know what you are doing or you could generate a curve severely less secure than you think. Even then, consider using a standardized curve for the sake of interoperability.

Currently only curves defined via the equation $y^2 \equiv x^3 + ax + b \pmod{p}$ are supported.

Args:

name (string): The name of the curve
p (int): The value of p in the curve equation.
a (int): The value of a in the curve equation.
b (int): The value of b in the curve equation.
q (int): The order of the base point of the curve.
gx (int): The x coordinate of the base point of the curve.
gy (int): The y coordinate of the base point of the curve.
oid (bytes): The object identifier of the curve.

__repr__() → str
Return repr(self).

__weakref__
list of weak references to the object (if defined)

evaluate(*x: int*) → int
Evaluate the elliptic curve polynomial at 'x'

Args: x (int): The position to evaluate the polynomial at

Returns: int: the value of $(x^3 + ax + b) mod p$

classmethod get_curve_by_oid(*oid: bytes*)
Get a curve via it's object identifier.

is_point_on_curve(*P*) → bool
Check if a point lies on this curve.

The check is done by evaluating the curve equation $y^2 \equiv x^3 + ax + b \pmod{p}$ at the given point (x, y) with this curve's domain parameters (a, b, p). If the congruence holds, then the point lies on this curve.

Args: P (long, long): A tuple representing the point P as an (x, y) coordinate pair.

Returns: bool: `True` if the point lies on this curve, otherwise `False`.

2.2 fastecdsa.ecdsa

exception fastecdsa.ecdsa.**EcdsaError**(*msg*)
Bases: `Exception`

fastecdsa.ecdsa.**sign**(*msg: MsgTypes, d: int, curve: fastecdsa.curve.Curve = P256, hashfunc=<built-in function openssl_sha256>, prehashed: bool = False*)
Sign a message using the elliptic curve digital signature algorithm.

The elliptic curve signature algorithm is described in full in FIPS 186-4 Section 6. Please refer to http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf for more information.

Args:

msg (str|bytes|bytearray): A message to be signed.
d (int): The ECDSA private key of the signer.
curve (fastecdsa.curve.Curve): The curve to be used to sign the message.
hashfunc (_hashlib.HASH): The hash function used to compress the message.

`fastecdsa.ecdsa.verify`(*sig: (<class 'int'>, <class 'int'>), msg: MsgTypes, Q: fastecdsa.point.Point, curve: fastecdsa.curve.Curve = P256, hashfunc=<built-in function openssl_sha256>*) → bool

Verify a message signature using the elliptic curve digital signature algorithm.

The elliptic curve signature algorithm is described in full in FIPS 186-4 Section 6. Please refer to http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf for more information.

Args:

sig (int, int): The signature for the message.

msg (str|bytes|bytearray): A message to be signed.

Q (fastecdsa.point.Point): The ECDSA public key of the signer.

curve (fastecdsa.curve.Curve): The curve to be used to sign the message.

hashfunc (_hashlib.HASH): The hash function used to compress the message.

Returns: bool: True if the signature is valid, False otherwise.

Raises:

fastecdsa.ecdsa.EcdsaError: If the signature or public key are invalid. Invalid signature in this case means that it has values less than 1 or greater than the curve order.

2.3 fastecdsa.encoding

class `fastecdsa.encoding.KeyEncoder`

Bases: `object`

Base class that any encoding class for EC keys should derive from.

All overriding methods should be static.

class `fastecdsa.encoding.SigEncoder`

Bases: `object`

Base class that any encoding class for EC signatures should derive from.

All overriding methods should be static.

2.4 fastecdsa.encoding.der

class `fastecdsa.encoding.der.DEREncoder`

Bases: *`fastecdsa.encoding.SigEncoder`*

static `decode_signature`(*sig: bytes*) -> (<class 'int'>, <class 'int'>)

Decode an EC signature from serialized DER format as described in https://tools.ietf.org/html/rfc2459 (section 7.2.2) and as detailed by bip-0066

Returns (r,s)

static `encode_signature`(*r: int, s: int*) → bytes

Encode an EC signature in serialized DER format as described in https://tools.ietf.org/html/rfc2459 (section 7.2.2) and as detailed by bip-0066

Args: r, s

Returns: bytes: The DER encoded signature

exception fastecdsa.encoding.der.**InvalidDerSignature**
Bases: Exception

2.5 fastecdsa.encoding.pem

class fastecdsa.encoding.pem.**PEMEncoder**
Bases: *fastecdsa.encoding.KeyEncoder*

static decode_private_key(*pemdata: str) -> (<class 'int'>, <class 'fastecdsa.point.Point'>*)
Decode an EC key as described in RFC 5915 and RFC 5480.

Args: pemdata (bytes): A sequence of bytes representing an encoded EC key.

Returns: (long, fastecdsa.point.Point): A private key, public key tuple. If the encoded key was a public key the first entry in the tuple is None.

static decode_public_key(*pemdata: str, curve: fastecdsa.curve.Curve = None*) → fastecdsa.point.Point
Delegate to private key decoding but return only the public key

static encode_private_key(*d: int, Q: fastecdsa.point.Point = None, curve: fastecdsa.curve.Curve = None*) → str
Encode an EC keypair as described in RFC 5915.

Args:

d (long): An ECDSA private key.

Q (fastecdsa.point.Point): The ECDSA public key.

curve (fastecdsa.curve.Curve): The curve that the private key is for.

Returns: str: The ASCII armored encoded EC keypair.

static encode_public_key(*Q: fastecdsa.point.Point*) → str
Encode an EC public key as described in RFC 5480.

Returns: str: The ASCII armored encoded EC public key.

2.6 fastecdsa.encoding.sec1

exception fastecdsa.encoding.sec1.**InvalidSEC1PublicKey**
Bases: Exception

class fastecdsa.encoding.sec1.**SEC1Encoder**
Bases: *fastecdsa.encoding.KeyEncoder*

static decode_public_key(*key: bytes, curve: fastecdsa.curve.Curve*) → fastecdsa.point.Point

Decode a public key as described in http://www.secg.org/SEC1-Ver-1.0.pdf in sections 2.3.3/2.3.4

compressed: 04 + x_bytes + y_bytes uncompressed: 02 or 03 + x_bytes

Args: key (bytes): public key encoded using the SEC1 format curve (fastecdsa.curve.Curve): Curve to use when decoding the public key

Returns: Point: The decoded public key

Raises: InvalidSEC1PublicKey

static encode_public_key(*point: fastecdsa.point.Point, compressed: bool = True*) → bytes

Encode a public key as described in http://www.secg.org/SEC1-Ver-1.0.pdf

in sections 2.3.3/2.3.4 compressed: 04 + x_bytes + y_bytes uncompressed: 02 or 03 + x_bytes

Args: point (fastecdsa.point.Point): Public key to encode compressed (bool): Set to False if you want an uncompressed format

Returns: bytes: The SEC1 encoded public key

2.7 fastecdsa.keys

fastecdsa.keys.**export_key**(*key*, *curve: fastecdsa.curve.Curve = None*, *filepath: str = None*, *encoder=<class 'fastecdsa.encoding.pem.PEMEncoder'>*)

Export a public or private EC key in PEM format.

Args:

key (fastecdsa.point.Point | int): A public or private EC key

curve (fastecdsa.curve.Curve): The curve corresponding to the key (required if the key is a private key)

filepath (str): Where to save the exported key. If None the key is simply printed.

encoder (type): The class used to encode the key

fastecdsa.keys.**gen_keypair**(*curve: fastecdsa.curve.Curve) -> (<class 'int'>, <class 'fastecdsa.point.Point'>*)

Generate a keypair that consists of a private key and a public key.

The private key d is an integer generated via a cryptographically secure random number generator that lies in the range $[1, n)$, where n is the curve order. The public key Q is a point on the curve calculated as $Q = dG$, where G is the curve's base point.

Args: curve (fastecdsa.curve.Curve): The curve over which the keypair will be calulated.

Returns: int, fastecdsa.point.Point: Returns a tuple with the private key first and public key second.

fastecdsa.keys.**gen_private_key**(*curve: fastecdsa.curve.Curve*, *randfunc=<built-in function urandom>*) → int

Generate a private key to sign data with.

The private key d is an integer generated via a cryptographically secure random number generator that lies in the range $[1, n)$, where n is the curve order. The default random number generator used is /dev/urandom.

Args:

curve (fastecdsa.curve.Curve): The curve over which the key will be calulated.

randfunc (function): A function taking one argument 'n' and returning a bytestring of n random bytes suitable for cryptographic use. The default is "os.urandom"

Returns: int: Returns a positive integer smaller than the curve order.

fastecdsa.keys.**get_public_key**(*d: int*, *curve: fastecdsa.curve.Curve*) → fastecdsa.point.Point

Generate a public key from a private key.

The public key Q is a point on the curve calculated as $Q = dG$, where d is the private key and G is the curve's base point.

Args:

d (long): An integer representing the private key.

curve (fastecdsa.curve.Curve): The curve over which the key will be calulated.

Returns: fastecdsa.point.Point: The public key, a point on the given curve.

`fastecdsa.keys.`**`get_public_keys_from_sig`**(*sig: (<class 'int'>, <class 'int'>), msg, curve: fastecdsa.curve.Curve = P256, hashfunc=<built-in function openssl_sha256>)* -> *(<class 'fastecdsa.point.Point'>, <class 'fastecdsa.point.Point'>)*

Recover the public keys that can verify a signature / message pair.

Args:

sig (int, int): A ECDSA signature.

msg (str|bytes|bytearray): The message corresponding to the signature.

curve (fastecdsa.curve.Curve): The curve used to sign the message.

hashfunc (_hashlib.HASH): The hash function used to compress the message.

Returns:

(fastecdsa.point.Point, fastecdsa.point.Point): The public keys that can verify the signature for the message.

`fastecdsa.keys.`**`import_key`**(*filepath: str, curve: fastecdsa.curve.Curve = None, public: bool = False, decoder=<class 'fastecdsa.encoding.pem.PEMEncoder'>*)

Import a public or private EC key in PEM format.

Args:

filepath (str): The location of the key file

public (bool): Indicates if the key file is a public key

decoder (fastecdsa.encoding.KeyEncoder): The class used to parse the key

Returns: (long, fastecdsa.point.Point): A (private key, public key) tuple. If a public key was imported then the first value will be None.

2.8 fastecdsa.point

exception `fastecdsa.point.`**`CurveMismatchError`**(*curve1, curve2*)

Bases: `Exception`

`__init__`(*curve1, curve2*)

Initialize self. See help(type(self)) for accurate signature.

`__weakref__`

list of weak references to the object (if defined)

class `fastecdsa.point.`**`Point`**(*x: int, y: int, curve=P256*)

Bases: `object`

Representation of a point on an elliptic curve.

Attributes:

x (int): The x coordinate of the point.

y (int): The y coordinate of the point.

curve (`Curve`): The curve that the point lies on.

`__add__`(*other*)

Add two points on the same elliptic curve.

Args:

self (`Point`): a point P on the curve other (`Point`): a point Q on the curve

Returns: `Point`: A point R such that $R = P + Q$

__eq__(*other*) → bool

Return self==value.

__init__(*x: int*, *y: int*, *curve=P256*)

Initialize a point on an elliptic curve.

The x and y parameters must satisfy the equation $y^2 \equiv x^3 + ax + b \pmod{p}$, where a, b, and p are attributes of the curve parameter.

Args:

x (int): The x coordinate of the point.

y (int): The y coordinate of the point.

curve (`Curve`): The curve that the point lies on.

__mul__(*scalar: int*)

Multiply a `Point` on an elliptic curve by an integer.

Args:

self (`Point`): a point P on the curve

other (int): an integer $d \in \mathbb{Z}_q$ where q is the order of the curve that P is on

Returns: `Point`: A point R such that $R = P * d$

__neg__()

Return the negation of a `Point` i.e. the points reflection over the x-axis.

Args:

self (`Point`): a point P on the curve

Returns: `Point`: A point $R = (P_x, -P_y)$

__radd__(*other*)

Add two points on the same elliptic curve.

Args:

self (`Point`): a point P on the curve

other (`Point`): a point Q on the curve

Returns: `Point`: A point R such that $R = P + Q$

__repr__() → str

Return repr(self).

__rmul__(*scalar: int*)

Multiply a `Point` on an elliptic curve by an integer.

Args:

self (`Point`): a point P on the curve

other (long): an integer $d \in \mathbb{Z}_q$ where q is the order of the curve that P is on

Returns: `Point`: A point R such that $R = d * P$

__str__()

Return str(self).

__sub__(*other*)

Subtract two points on the same elliptic curve.

Args:

self (`Point`): a point P on the curve

other (`Point`): a point Q on the curve

Returns: `Point`: A point R such that $R = P - Q$

`__weakref__`
list of weak references to the object (if defined)

2.9 fastecdsa.util

class `fastecdsa.util.`**`RFC6979`**(*msg*, *x*, *q*, *hashfunc*)
Bases: `object`

Generate a nonce per RFC6979.

In order to avoid reusing a nonce with the same key when signing two different messages (which leaks the private key) RFC6979 provides a deterministic method for generating nonces. This is based on using a pseudo-random function (HMAC) to derive a nonce from the message and private key. More info here: http://tools.ietf.org/html/rfc6979.

Attributes:

msg (string): A message being signed.

x (int): An ECDSA private key.

q (int): The order of the generator point of the curve being used to sign the message.

hashfunc (_hashlib.HASH): The hash function used to compress the message.

`gen_nonce`()
http://tools.ietf.org/html/rfc6979#section-3.2

`fastecdsa.util.`**`mod_sqrt`**(*a: int*, *p: int) -> (<class 'int'>*, *<class 'int'>*)
Compute the square root of $a \pmod p$

In other words, find a value x such that $x^2 \equiv a \pmod p$.

Args:

a (int): The value whose root to take.

p (int): The prime whose field to perform the square root in.

Returns: (int, int): the two values of x satisfying $x^2 \equiv a \pmod p$.

`fastecdsa.util.`**`msg_bytes`**(*msg*) → bytes
Return bytes in a consistent way for a given message.

The message is expected to be either a string, bytes, or an array of bytes.

Args:

msg (str|bytes|bytearray): The data to transform.

Returns: bytes: The byte encoded data.

Raises: ValueError: If the data cannot be encoded as bytes.

Python Module Index

f

Index

Symbols

C

D

E

F

G

I

K

M

P

R

S

V