fastecdsa Documentation

Release 1.6.4

Anton Kueltz

Apr 09, 2019

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.asn1

2.2 fastecdsa.curve

class fastecdsa.curve.**Curve**(*name*, *p*, *a*, *b*, *q*, *gx*, *gy*, *oid=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 (string): The name of the curve
p (long): The value of p in the curve equation.
a (long): The value of a in the curve equation.
b (long): The value of b in the curve equation.
q (long): The order of the base point of the curve.
oid (long): 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*, *p*, *a*, *b*, *q*, *gx*, *gy*, *oid=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 (long): The value of p in the curve equation.
a (long): The value of a in the curve equation.
b (long): The value of b in the curve equation.
q (long): The order of the base point of the curve.
gx (long): The x coordinate of the base point of the curve.
gy (long): The y coordinate of the base point of the curve.
oid (str): The object identifier of the curve.

`__repr__`()
Return repr(self).

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

`evaluate`(*x*)
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*)
Get a curve via it's object identifier.

`is_point_on_curve`(*P*)
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`.

2.3 fastecdsa.ecdsa

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

`fastecdsa.ecdsa.`**`sign`**(*msg*, *d*, *curve=P256*, *hashfunc=<built-in function openssl_sha256>*)
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 (long): 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*, *msg*, *Q*, *curve=P256*, *hashfunc=<built-in function openssl_sha256>*)
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 (long, long): 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.

2.4 fastecdsa.keys

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

Export a public or private EC key in PEM format.

Args:

key (fastecdsa.point.Point | long): 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 (fastecdsa.encoding.KeyEncoder): The class used to encode the key

`fastecdsa.keys.`**`gen_keypair`**(*curve*)

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

`fastecdsa.keys.`**`gen_private_key`**(*curve*, *randfunc=<built-in function urandom>*)

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

`fastecdsa.keys.`**`get_public_key`**(*d*, *curve*)

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*, *msg*, *curve=P256*, *hashfunc=<built-in function openssl_sha256>*)

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

Args:

sig (long, long): 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*, *curve=None*, *public=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.

2.5 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*, *y*, *curve=P256*)

Bases: `object`

Representation of a point on an elliptic curve.

Attributes:

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

y (long): 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*)

Return self==value.

`__init__`(*x*, *y*, *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 (long): The x coordinate of the point.

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

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

`__mul__`(*scalar*)

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 $R = P * d$

`__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__`()

Return repr(self).

`__rmul__`(*scalar*)

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)

2.6 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 (long): An ECDSA private key.
q (long): 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*, *p*)
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 (long): The value whose root to take.
p (long): The prime whose field to perform the square root in.

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

`fastecdsa.util.`**`msg_bytes`**(*msg*)
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

E

F

G

I

M

P

R

S

V