

---

# **rtrlib-python Documentation**

***Release 0.1***

**Marcel Röthke**

**Jun 09, 2017**



---

## Contents

---

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	Python Requirements . . . . .	3
1.3	Download and Installation . . . . .	3
<b>2</b>	<b>Indices and tables</b>	<b>15</b>
	<b>Python Module Index</b>	<b>17</b>



`rtrlib-python` is a cffi based binding for `RTRlib`.

The RTRlib implements the client-side of the RPKI-RTR protocol (RFC 6810) and BGP Prefix Origin Validation (RFC 6811). This release also supports Internet-Draft draft-ietf-sidr-rpki-rtr-rfc6810-bis, which enables the maintenance of router keys. Router keys are required to deploy BGPSEC.

Currently only basic validation against one cache is supported.



### Requirements

- python 2.7 or python 3
- C Compiler
- [RTRlib](#)

### Python Requirements

If you are using virtualenv these are installed automatically during the install step, otherwise you have to use your platforms package management tool or just run `pip install -r requirements.txt`.

- `cffi>=1.4.0`
- `enum34`
- `six`

### Download and Installation

- `git clone https://github.com/rtrlib/python-binding.git`
- `cd python-binding`
- `python setup.py build`
- `python setup.py install`

For usage examples see here [here](#) or in the [tools](#) dir of the repository.

Contents:

## API Documentation

rtrlib-python - a cffi based rtrlib wrapper

**license** MIT, see LICENSE for more details.

### rtrlib.rtr\_manager

```
class rtrlib.rtr_manager.RTRManager(host, port, refresh_interval=3600, expire_interval=7200,
                                     retry_interval=600, status_callback=None, sta-
                                     tus_callback_data=None)
```

Wrapper around rtr\_manager.

#### Parameters

- **host** (*str*) – Hostname or IP of rpki cache server
- **port** (*int*) – Port number
- **refresh\_interval** (*int*) – Interval in seconds between serial queries that are sent to the server. Must be  $\geq 1$  and  $\leq 86400$ s (one day).
- **expire\_interval** (*int*) – Stored validation records will be deleted if cache was unable to refresh data for this period. The value should be twice the refresh\_interval. The value must be  $\geq 600$ s (ten minutes) and  $\leq 172800$ s (two days).
- **retry\_interval** (*int*) – This parameter specifies how long to wait (in seconds) before retrying a failed Query. The value must be  $\geq 1$ s and  $\leq 7200$ s (two hours).
- **status\_callback** (*function*) – status callback, called on status changes of the rtr manager
- **status\_callback\_data** (*object*) – arbitrary data object passed to the callback.

Raises *RTRInitError* –

**for\_each\_ipv4\_record** (*callback*, *data*)

Iterate over all ipv4 records of the pfx table.

callback must take two arguments, the pfx\_record and the data object.

For a more pythonic alternative see *ipv4\_records()*

#### Parameters

- **callback** (*callable*) – called for every record in the pfx table
- **data** (*object*) – arbitrary data object that is passed to the callback function

**for\_each\_ipv6\_record** (*callback*, *data*)

Iterate over all ipv6 records of the pfx table.

callback must take two arguments, the pfx\_record and the data object.

For a more pythonic alternative see *ipv6\_records()*

#### Parameters

- **callback** (*callable*) – called for every record in the pfx table
- **data** (*object*) – arbitrary data object that is passed to the callback function



**ipv4\_records()**

Return iterator over all ipv4 records in the pfx table.

This iterator utilises threads to execute retrieve the records. If that is a problem for you take a look at [\*for\\_each\\_ipv4\\_record\(\)\*](#).

**Return type** Iterator

**ipv6\_records()**

Return iterator over all ipv6 records in the pfx table.

This iterator utilises threads to execute retrieve the records. If that is a problem for you take a look at [\*for\\_each\\_ipv6\\_record\(\)\*](#).

**Return type** Iterator

**is\_synced()**

Check if RTRManager is fully synchronized.

**Return type** bool

**start** (*wait=True, timeout=5*)

Start RTRManager.

**Parameters**

- **wait** (*bool*) – Wait for the manager to finish sync
- **timeout** (*int*) –

**Raises** [\*SyncTimeout\*](#) – Raised if timeout is reached, this does not mean that the sync failed, only that it did not finish in time.

**stop()**

Stop RTRManager.

**validate** (*asn, prefix, mask\_len*)

Validate BGP prefix and returns state as PfxvState enum.

**Parameters**

- **asn** (*int*) – autonomous system number
- **prefix** (*str*) – ip address
- **mask\_len** (*int*) – length of the subnet mask

**Return type** [\*ValidationResult\*](#)

**wait\_for\_sync** (*timeout=5*)

Wait until RTRManager is synchronized.

**Parameters** **timeout** (*int*) –

**Raises** [\*SyncTimeout\*](#) – Raise if timeout is reached, this does not mean that the sync failed, only that it did not finish in time.

**class** `rtrlib.rtr_manager.PfxvState`

Wrapper for the pfxv\_state enum.

**invalid** = <class 'sphinx.ext.autodoc.BGP\_PFXV\_STATE\_INVALID'>

One or more records that match the input prefix exists in the pfx\_table, but the prefix max\_len or ASN doesn't match.

**not\_found** = <class 'sphinx.ext.autodoc.BGP\_PFXV\_STATE\_NOT\_FOUND'>

No certificate for the route exists

**valid** = <class 'sphinx.ext.autodoc.BGP\_PFXV\_STATE\_VALID'>

A valid certificate for the pfx\_record exists

**class** rtrlib.rtr\_manager.**ValidationResult** (*prefix, prefix\_length, asn, state, reason\_records=None, reason\_len=0*)

Wrapper class for validation result.

#### Parameters

- **prefix** (*str*) – The prefix that was validated
- **prefix\_length** (*int*) – The length of the prefix
- **asn** – The ASN the prefix is supposed to be in.
- **asn** – int
- **state** (*enum pfxv\_state \**) – Validation state
- **reason\_records** (*struct pfx\_record \*\**) – Array of PFXRecords the decision is based on
- **reason\_len** (*int*) – Length of reason\_records

**as\_invalid**

True if at least one matching record has a different as number and state is invalid.

**as\_valid**

True if any one matching record has been found.

**is\_invalid**

Return true if prefix is invalid.

**is\_valid**

True if prefix is valid.

**length\_invalid**

True if at least one matching record has a miss matching prefix length and state is invalid.

**length\_valid**

True if any one matching record was found

**not\_found**

True if prefix could not be found.

**reason**

List of *Reason*.

**state**

Validation state.

**class** rtrlib.rtr\_manager.**Reason** (*prefix\_length, asn, record*)

A Reason upon which a validation decision was made.

#### Parameters

- **prefix\_length** (*int*) – Length of the validated prefix
- **asn** (*As number of the validated prefix*) – As number of the validated prefix
- **record** (*PFXRecord*) – PFXRecord

**as\_invalid**

True is as is invalid.

**as\_valid**  
True if as is valid.

**length\_invalid**  
True if prefix length is invalid.

**length\_valid**  
True if prefix length is valid.

## rtrlib.rtr\_socket

**class** `rtrlib.rtr_socket.RTRSocket` (*socket*)

Wrapper around the `rtr_socket` struct

**Parameters** `socket` (*cdata*) – `rtr_socket` struct

**expire\_interval**  
Time period in seconds. Received records are deleted if the client was unable to refresh data for this time period. If 0 is specified, the `expire_interval` is twice the `refresh_interval`.

**has\_recieved\_pdus**  
True, if this socket has already received PDUs

**last\_update**  
Timestamp of the last validation record update. Is 0 if the `pfx_table` doesn't stores any validation records from this `rtr_socket`.

**refresh\_interval**  
Time period in seconds. Tells the router how long to wait before next attempting to poll the cache, using a Serial Query or Reset Query PDU.

**retry\_interval**  
Time period in seconds between a failed query and the next attempt.

**state**  
Current state of the socket.

**version**  
Protocol version used by this socket

**class** `rtrlib.rtr_socket.RTRSocketList` (*sockets, length*)

List of `RTRSocket`s. Can be accessed like any other list.

Read Only.

**class** `rtrlib.rtr_socket.RTRSocketState`

States of the RTR socket

**CONNECTING** = <class 'sphinx.ext.autodoc.RTR\_CONNECTING'>

Socket is establishing the transport connection

**ERROR\_FATAL** = <class 'sphinx.ext.autodoc.RTR\_ERROR\_FATAL'>

Fatal protocol error occurred

**ERROR\_NO\_DATA\_AVAILABLE** = <class 'sphinx.ext.autodoc.RTR\_ERROR\_NO\_DATA\_AVAIL'>

No validation records are available on the RTR server

**ERROR\_NO\_INCREMENTAL\_UPDATE\_AVAILABLE** = <class 'sphinx.ext.autodoc.RTR\_ERROR\_NO\_INCR\_UPDATE\_AVAIL'>

Server was unable to answer the last serial or reset query

**ERROR\_TRANSPORT** = <class 'sphinx.ext.autodoc.RTR\_ERROR\_TRANSPORT'>

Error on the transport socket occurred

**ESTABLISHED** = <class 'sphinx.ext.autodoc.RTR\_ESTABLISHED'>

Connection is established and socket is waiting for a Serial Notify or expiration of the refresh\_interval timer.

**FAST\_RECONNECT** = <class 'sphinx.ext.autodoc.RTR\_FAST\_RECONNECT'>

Reconnect without any waiting period

**RESET** = <class 'sphinx.ext.autodoc.RTR\_RESET'>

Resetting RTR connection

**SHUTDOWN** = <class 'sphinx.ext.autodoc.RTR\_SHUTDOWN'>

RTR Socket is stopped

**SYNC** = <class 'sphinx.ext.autodoc.RTR\_SYNC'>

Receiving validation records from the RTR server

## rtrlib.records

Collection of wrappers for \*record structs of rtrlib

**class** `rtrlib.records.PFXRecord(record)`

Wrapper around the pfx\_record struct.

**asn**

Origin AS number.

**max\_len**

Maximum prefix length.

**min\_len**

Minimum prefix length.

**prefix**

IP prefix.

**socket**

*RTRSocket* this record was received in.

**class** `rtrlib.records.SPKIRecord(record)`

Wrapper around the spki\_record struct.

**asn**

Origin AS number.

**ski**

Subject Key Identifier.

**socket**

*RTRSocket* this record was received in.

**spki**

Subject public key info.

`rtrlib.records.copy_pfx_record(record)`

Copy a pfx record.

**Parameters** `record` (*PFXRecord*) – The record that should be copied

**Return type** *PFXRecord*

## rtrlib.manager\_group

**class** `rtrlib.manager_group.ManagerGroup` (*group*)

Wrapper around the `rtr_mgr_group` struct

**Parameters** `group` (*cdata*) – A `rtr_mgr_group` struct

**preference**

The preference value of the group

**sockets**

The socket list as `RTRSocketList`

**sockets\_len**

The `sockets_len` value of the group

**status**

The group status as `enum34`

**class** `rtrlib.manager_group.ManagerGroupStatus`

Wrapper around the C enum `rtr_mgr_status`.

**CLOSED** = <class ‘`sphinx.ext.autodoc.RTR_MGR_CLOSED`’>

RTR sockets are disconnected

**CONNECTING** = <class ‘`sphinx.ext.autodoc.RTR_MGR_CONNECTING`’>

RTR sockets trying to establish a connection

**ERROR** = <class ‘`sphinx.ext.autodoc.RTR_MGR_ERROR`’>

Error occurred on at least one RTR socket

**ESTABLISHED** = <class ‘`sphinx.ext.autodoc.RTR_MGR_ESTABLISHED`’>

All RTR sockets of the group are synchronized with the rtr servers

## rtrlib.exceptions

Module for all custom exceptions

**exception** `rtrlib.exceptions.IpConversionException`

An Error during str to address conversion or the reverse occurred.

**exception** `rtrlib.exceptions.PFXException`

An error during validation occurred.

**exception** `rtrlib.exceptions.RTRInitError`

An error during initialization of the RTR manager occurred.

**exception** `rtrlib.exceptions.RTRlibException`

rtrlib exception base class.

**exception** `rtrlib.exceptions.SyncTimeout`

The timeout was reached while waiting for sync.

## Callbacks

Rtrlib provides 3 callbacks one for updates on the manager status, one for `pfx_table` and one for `spki_table` updates.

## RTR Manager Status Callback

This callback is called when the RTR Managers status is changed. The callback function must take 4 arguments.

**manager\_status\_callback** (*rtr\_mgr\_group*, *group\_status*, *rtr\_socket*, *data*)

### Parameters

- **rtr\_mgr\_group** – socket group where the status change originates
- **group\_status** – the new status
- **rtr\_socket** – the socket where the change originates
- **data** (*object* or *None*) – Data Object, if defined at manager initialization

This callback is registered at manager initialization using `status_callback` parameter. The data object may be passed with the `status_callback_data` parameter.

## PFX iteration callback

This callback can be used to iterate over the entire pfx table.

**pfx\_for\_each** (*pfx\_record*, *data*)

*pfx\_record* is only guaranteed to be valid during this function call. If you want to store it somewhere e.g. in *data* than you have to copy it. you can use `rtrlib.records.copy_pfx_record()` for this.

### Parameters

- **pfx\_record** (*PFXRecord*) – Pfx record from the iterated pfx table
- **data** (*object* or *None*) – Arbitrary data object provided by the user

**Warning:** You should **not** register more than one function per callback for the following callbacks, it will **not** work and result in undefined behaviour

## PFX update callback

This callback is called for any change to the Prefix validation table, it takes two arguments.

**pfx\_update\_callback** (*pfx\_record*, *added*) :

### Parameters

- **pfx\_record** (*rtrlib.records.PFXRecord*) – The affected pfx record
- **added** (*bool*) – Indicates whether the record was added or removed

This callback can be registered using the `rtrlib.register_pfx_update_callback()` function

**rtrlib.register\_pfx\_update\_callback** (*func*)

Register RTR manager `pfx_update_callback`

**Parameters** **func** (*function*) – Callback function

## SPKI update callback

This callback is called for any change to the Subject Public Key Info table, it takes two arguments.

**spki\_update\_callback(spki\_record, added) :**

### Parameters

- **spki\_record** (*rtrlib.records.PFXRecord*) – The affected spki record
- **added** (*bool*) – Indicates whether the record was added or removed

This callback can be registered using the *rtrlib.register\_spki\_update\_callback()* function

**rtrlib.register\_spki\_update\_callback(func)**

Register RTR manager spki update callback

**Warning:** This callback is untested, because at the time of writing no caching server with spki records was available. It should work but might cause you computer to explode.

**Parameters** **func** (*function*) – Callback function

## Usage Examples

### Validation

```
from rtrlib import RTRManager, PfxvState

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)

if result == PfxvState.valid:
    print('Prefix Valid')
elif result == PfxvState.invalid:
    print('Prefix Invalid')
elif result == PfxvState.not_found:
    print('Prefix not found')
else:
    print('Invalid response')

mgr.stop()
```

### PFX Table iteration (with iterator)

```
from rtrlib import RTRManager, PfxvState

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)

for recordv4 in mgr.ipv4_records():
    print(recordv4)

mgr.stop()
```

### PFX Table iteration (with callback)

```
from rtrlib import RTRManager, PfxvState

def callback(pfx_record, data):
    print(pfx_record)

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()
result = mgr.validate(12345, '10.10.0.0', 24)

mgr.for_each_ipv4_record(callback, None)

mgr.stop()
```

### Print PFX updates

```
from rtrlib import RTRManager, register_pfx_update_callback

def callback(pfx_record, added):
    print('%s %s' % ('+' if added else '-', pfx_record))

register_pfx_update_callback(callback)

mgr = RTRManager('rpki-validator.realmv6.org', 8282)
mgr.start()

mgr.stop()
```

### Advanced Usage

---

**Note:** This is by no means supposed to be a reference on cffi itself, if you want to do something like this please read the [cffi Documentation](#).

---

In case you want to do something that is not (yet) supported by the binding you can access the c functions directly.

Let's say you implemented RFC6810 yourself but still want to use rtrlib's pfxtable.

```
# _rtrlib is the cffi object, it contains the actual bindings in lib
# and helper functions for allocation and
# other stuff that is not native to python
from _rtrlib import lib, ffi

# only imported for the pfx_table_callback
import rtrlib

# allocate pfx_table
pfx_table = ffi.new('struct pfx_table *')

# initialize it
```



```

lib.pfx_table_init(pfx_table, ffi.NULL)

def add_record(asn, ip, prefixmin, prefixmax):
    record = ffi.new('struct pfx_record *')
    prefix = ffi.new('struct lrtr_ip_addr *')
    lib.lrtr_ip_str_to_addr(ip.encode('ascii'), prefix)

    record.asn = asn
    record.min_len = prefixmin
    record.max_len = prefixmax
    record.socket = ffi.NULL
    record.prefix = prefix[0]

    lib.pfx_table_add(pfx_table, record)

# add records
records = ((234, '22.45.66.0', 24, 24),
           (545, '9..0.0', 8, 8),
           (4545, '223.4.66.0', 24, 24),
           (5454, '120.6.47.0', 24, 24))

for record in records:
    asn, ip, min_len, max_len = record
    add_record(asn, ip, min_len, max_len)

# iterate over pfx_table to demonstrate it's content

# since the callback from the rtrlib module is used record
# is automatically wrapped in a python class
def callback(record, notused):
    print(record)

# necessary because cffi new style callbacks are used,
# lib.pfx_table_callback is a wrapper that calls the actual callback
handle = ffi.new_handle((callback, None))

lib.pfx_table_for_each_ipv4_record(pfx_table, lib.pfx_table_callback, handle)

lib.pfx_table_free(pfx_table)

```



## CHAPTER 2

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### **r**

- `rtrlib`, 4
- `rtrlib.exceptions`, 9
- `rtrlib.manager_group`, 8
- `rtrlib.records`, 8
- `rtrlib.rtr_manager`, 4
- `rtrlib.rtr_socket`, 7



## A

as\_invalid (rtrlib.rtr\_manager.Reason attribute), 6  
as\_invalid (rtrlib.rtr\_manager.ValidationResult attribute), 6  
as\_valid (rtrlib.rtr\_manager.Reason attribute), 6  
as\_valid (rtrlib.rtr\_manager.ValidationResult attribute), 6  
asn (rtrlib.records.PFXRecord attribute), 8  
asn (rtrlib.records.SPKIRecord attribute), 8

## C

CLOSED (rtrlib.manager\_group.ManagerGroupStatus attribute), 9  
CONNECTING (rtrlib.manager\_group.ManagerGroupStatus attribute), 9  
CONNECTING (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
copy\_pfx\_record() (in module rtrlib.records), 8

## E

ERROR (rtrlib.manager\_group.ManagerGroupStatus attribute), 9  
ERROR\_FATAL (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
ERROR\_NO\_DATA\_AVAILABLE (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
ERROR\_NO\_INCREMENTAL\_UPDATE\_AVAILABLE (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
ERROR\_TRANSPORT (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
ESTABLISHED (rtrlib.manager\_group.ManagerGroupStatus attribute), 9  
ESTABLISHED (rtrlib.rtr\_socket.RTRSocketState attribute), 7  
expire\_interval (rtrlib.rtr\_socket.RTRSocket attribute), 7

## F

FAST\_RECONNECT (rtrlib.rtr\_socket.RTRSocketState attribute), 8

for\_each\_ipv4\_record() (rtrlib.rtr\_manager.RTRManager method), 4  
for\_each\_ipv6\_record() (rtrlib.rtr\_manager.RTRManager method), 4

## H

has\_recieved\_pdus (rtrlib.rtr\_socket.RTRSocket attribute), 7

## I

invalid (rtrlib.rtr\_manager.PfxvState attribute), 5  
IpConversionException, 9  
ipv4\_records() (rtrlib.rtr\_manager.RTRManager method), 4  
ipv6\_records() (rtrlib.rtr\_manager.RTRManager method), 5  
is\_invalid (rtrlib.rtr\_manager.ValidationResult attribute), 6  
is\_synced() (rtrlib.rtr\_manager.RTRManager method), 5  
is\_valid (rtrlib.rtr\_manager.ValidationResult attribute), 6

## L

last\_update (rtrlib.rtr\_socket.RTRSocket attribute), 7  
length\_invalid (rtrlib.rtr\_manager.Reason attribute), 7  
length\_invalid (rtrlib.rtr\_manager.ValidationResult attribute), 6  
length\_valid (rtrlib.rtr\_manager.Reason attribute), 7  
length\_valid (rtrlib.rtr\_manager.ValidationResult attribute), 6

## M

manager\_status\_callback() (built-in function), 10  
ManagerGroup (class in rtrlib.manager\_group), 9  
ManagerGroupStatus (class in rtrlib.manager\_group), 9  
max\_len (rtrlib.records.PFXRecord attribute), 8  
min\_len (rtrlib.records.PFXRecord attribute), 8

## N

not\_found (rtrlib.rtr\_manager.PfxvState attribute), 5

not\_found (rtrlib.rtr\_manager.ValidationResult attribute), 6

## P

pfx\_for\_each() (built-in function), 10  
 PFXException, 9  
 PFXRecord (class in rtrlib.records), 8  
 PfxvState (class in rtrlib.rtr\_manager), 5  
 preference (rtrlib.manager\_group.ManagerGroup attribute), 9  
 prefix (rtrlib.records.PFXRecord attribute), 8

## R

Reason (class in rtrlib.rtr\_manager), 6  
 reason (rtrlib.rtr\_manager.ValidationResult attribute), 6  
 refresh\_interval (rtrlib.rtr\_socket.RTRSocket attribute), 7  
 register\_pfx\_update\_callback() (in module rtrlib), 10  
 register\_spki\_update\_callback() (in module rtrlib), 11  
 RESET (rtrlib.rtr\_socket.RTRSocketState attribute), 8  
 retry\_interval (rtrlib.rtr\_socket.RTRSocket attribute), 7  
 RTRInitError, 9  
 rtrlib (module), 4  
 rtrlib.exceptions (module), 9  
 rtrlib.manager\_group (module), 8  
 rtrlib.records (module), 8  
 rtrlib.rtr\_manager (module), 4  
 rtrlib.rtr\_socket (module), 7  
 RTRlibException, 9  
 RTRManager (class in rtrlib.rtr\_manager), 4  
 RTRSocket (class in rtrlib.rtr\_socket), 7  
 RTRSocketList (class in rtrlib.rtr\_socket), 7  
 RTRSocketState (class in rtrlib.rtr\_socket), 7

## S

SHUTDOWN (rtrlib.rtr\_socket.RTRSocketState attribute), 8  
 ski (rtrlib.records.SPkiRecord attribute), 8  
 socket (rtrlib.records.PFXRecord attribute), 8  
 socket (rtrlib.records.SPkiRecord attribute), 8  
 sockets (rtrlib.manager\_group.ManagerGroup attribute), 9  
 sockets\_len (rtrlib.manager\_group.ManagerGroup attribute), 9  
 spki (rtrlib.records.SPkiRecord attribute), 8  
 SPkiRecord (class in rtrlib.records), 8  
 start() (rtrlib.rtr\_manager.RTRManager method), 5  
 state (rtrlib.rtr\_manager.ValidationResult attribute), 6  
 state (rtrlib.rtr\_socket.RTRSocket attribute), 7  
 status (rtrlib.manager\_group.ManagerGroup attribute), 9  
 stop() (rtrlib.rtr\_manager.RTRManager method), 5  
 SYNC (rtrlib.rtr\_socket.RTRSocketState attribute), 8  
 SyncTimeout, 9

## V

valid (rtrlib.rtr\_manager.PfxvState attribute), 5  
 validate() (rtrlib.rtr\_manager.RTRManager method), 5  
 ValidationResult (class in rtrlib.rtr\_manager), 6  
 version (rtrlib.rtr\_socket.RTRSocket attribute), 7

## W

wait\_for\_sync() (rtrlib.rtr\_manager.RTRManager method), 5