PyRIC v0.1.5: User Manual

Dale V. Patterson
wraith.wireless@yandex.com

July 24, 2016

Contents

1 About PyRIC

PyRIC (is a Linux only) library providing wireless developers and pentesters the ability to identify, enumerate and manipulate their system's wireless cards programmatically in Python. Pentesting applications and scripts written in Python have increased dramatically in recent years. However, these tools still rely on Linux command lines tools to setup and prepare and restore the system for use. Until now. Why use subprocess.Popen, regular expressions and str.find to interact with your wireless cards? PyRIC puts iw, ifconfig, rfkill, udevadm, airmon-ng and macchanger in your hands (or your program).

PyRIC is designed with Python 2.7 in mind but has now been made compatible with Python 3.5. It will also work on Python 3.0 but you will have to hard code the command line options in the two examples as Python 3.0 does not include the module argparse
PyRIC is:

1. **Pythonic**: No ctypes, SWIG etc. PyRIC redefines C header files as Python and uses sockets to communicate with kernel.

2. **Self-sufficient**: No third-party files used, PyRIC is completely self- contained

3. **Fast**: (relatively speaking) PyRIC is faster than using iw through subprocess.Popen

4. **Parseless**: Get the output you without parsing output from iw. Never worry about iw updates and rewriting your parsers.

5. **Easy**: If you can use iw, you can use PyRIC

At it's heart, PyRIC is a Python port of (a subset of) iw and by extension, a Python port of Netlink w.r.t nl80211 functionality. The original goal of PyRIC was to provide a simple interface to the underlying nl80211 kernel support, handling the complex operations of Netlink seamlessy while maintaining a minimum of "code walking" to understand, modify and extend. But, why stop there? Since it's initial inception, PyRIC has grown to include ioctl support to replicate features of ifconfig such as getting or setting the mac address and has recently implemented rkill support to soft block or unblock wireless cards.

While users can utilize libnl.py to communicate directly with the kernel, the true utility of PyRIC is pyw.py. Like iw, pyw provides an interface/buffer between the caller and the kernel, handling all message construction, parsing and data transfer transparently and without requiring any Netlink knowledge or experience.

At this time, PyRIC can:

- enumerate interfaces and wireless interfaces,
- identify a cards driver, chipset and manufacturer,
- get/set hardware address,
- get/set ip4 address, netmask and or broadcast,
- turn card on/off,
- get supported standards, commands or modes,
- get if info,
- get dev info,
- get phy info,
- get link info,
- get STA (connected AP) info,
- get/set regulatory domain,
- get/set mode,
- get/set coverage class, RTS threshold, Fragmentation threshold and retry limits,
- add/delete interfaces,
- determine if a card is connected,
- get link info for a connected card,
- enumerate ISM and UNII channels,

- block/unblock rfkill devices.

And, through libnl.py and libio.py, users can extend the above functionality by creating additional commands.

1.1 Background

PyRIC arose out of a need in Wraith (https://github.com/wraith-wireless/wraith) for Python nl80211/netlink and ioctl functionality. Originally, Wraith used ifconfig, iwconfig and iw via subprocess.Popen and parsed the output. There are obvious shortfalls with this method, especially in terms of iw that is actively changing (revisions break the parser) and I started looking for an open source alternative. There are several open source projects out there such as pyroute, pymnl (and the python files included in the libnl source) but they generally have either not been maintained recently or come with warnings. I desired a simple interface to the underlying nl80211 kernel support that handles the complex operations of netlink seamlessy while maintaining a minimum of "code walking" to understand, modify and extend. I decided to write my own because I do not need complete netlink functionality, only that provided by generic netlink and within the nl80221 family. Additionally, for Wraith, I do not need a full blown port of iw et. al. functionality to Python but only require the ability to turn a wireless nic on/off, get/set the hwaddr, get/set the channel, determine some properties of the card and add/delete interfaces.
So, why did I do this and why is it done "this" way? When I first started to explore the idea of moving away from iw output parsing, I looked at the source for iw, and existing Python ports. Just to figure out how to get the family id for nl80211 required reading through five different source files with no comments. To that extent, I have attempted to keep subclassing to a minimum, the total number of classes to a minimum, combine files where possible and where it makes since and keep the number of files required to be open simultaneously in order to understand the methodology and follow the program to a minimum. One can understand the PyRIC program flow with only two files open at any time namely, pyw and libnl. In fact, only an understanding of pyw is required to add additional commands although an understanding of libnl.py is helpful especially, if for example, the code is to be extended to handle multicast or callbacks.

1.2 Naming Conventions

The terms interface, device and radio are all used interchangeably throughout to refer to a network interface controller (NIC). The following terms will always have one meaning:

- **dev** - the device name i.e. wlan0 or eth0 of a NIC,
- **phy** - the physical index of a NIC i.e. the 0 in phy0,
- **ifindex** - the interface index of a NIC,
- **card** or **Card** - a NIC abstraction, an object used in pyw functions see the following section for a description.

1.3 Cards

A Card is merely a wrapper around a tuple t = (phy index,device name,ifindex). Since the underlying Netlink calls sometimes require the physical index, sometimes the device name, and sometimes the

chset	Total	Avg	Longest	Shortest
Popen(iw)	588.3059	0.0588	0.0682	0.0021
one-time	560.3559	0.0560	0.0645	0.0003
persistent	257.8293	0.0257	0.0354	0.0004

Table 1: Benchmark: Popen(iw) vs pyw

ifindex, pyw functions[1] take a Card object which doesn't require callers to know which identifier to use for each function. There are four primary methods to creating a Card:

1. **pyw.getcard** returns a Card object from a given dev,
2. **pyw.devinfo** returns the dict info where info['card'] is the Card object. This function will take either a card or a dev
3. **pyw.devadd** returns a new Card object,
4. **pyw.ifaces** returns a list of tuples t = (Card,mode) sharing the same phy as a given dev.

A side affect of using Cards is that many of the netlink calls require the ifindex. The ifindex is found through the use of ioctl, meaning two sockets have to be created and two messages have to be sent, received and parsed in order to execute the command. With Cards, the ifindex is requested for only once.

Keep in mind that any identifier (phy, dev, ifindex) can be invalidated outside of your control. Another program can rename your interface, that is change the dev without your knowledge. Depending on what functions are being used this may not be noticed right away as the phy will remain the same. Also for usb devices, (if the usb is disconnected and reconnected) will have the same dev but the phy and ifindex will be different.

1.4 Benchmarks

PyRIC makes use of several "extensions" to speed up pyw functions:

1. **Persistent sockets**: pyw provides the caller with functions and the ability to pass their own netlink (or ioctl socket) to pyw functions
2. **One-time request for the nl80211 family id**: pyw stores the family id in a global variable
3. **Consolidation** different "reference" values are consolidated in one class (see the previous section)

While small, these changes can improve the performance of any programs using pyw. Table 1 shows benchmarks for hop time on an Alfa AWUS036NH conducted 10000 times. Note that we are not implying that PyRIC is faster than iw. Rather, the table shows that PyRIC is faster than using Popen to execute iw. Using one-time sockets, there is a difference of 28 seconds over Popen and iw with a small decrease in the average hoptime. Not a big difference. However, the performance increased dramatically when persistent netlink sockets are used with the total time and average hop time nearly halved.

[1]Not all functions accept a Card, devinfo() accept either a Card or a dev, devadd accepts either a Card or a ifindex and phyadd accepts a card or a physical index

Source	Stability	Recency	Installation
pip	5	3	5
PyPI	5	3	4
PyRIC Web	4	4	4
Github	3	5	3

Table 2: Stability vs Recency vs Installation

2 Installing PyRIC

The easiest way to install PyRIC is through PyPI:

```
sudo pip install PyRIC
```

You can also install PyRIC from source. The tarball can be downloaded from:

- PyPi: https://pypi.python.org/pypi/PyRIC,
- PyRIC Web: http://wraith-wireless.github.io/PyRIC, or
- Github: https://github.com/wraith-wireless/PyRIC.

After downloading, extract and run:

```
sudo python setup.py install
```

If you just want to test PyRIC out, download your choice from above. After extraction, move the pyric folder (the package directory) to your location of choice and from there start Python and import pyw. It is very important that you do not try and run it from PyRIC which is the distribution directory. This will break the imports pyw.py uses.

You will only be able to test PyRIC from the pyric directory but, if you want to, you can add it to your Python path and run it from any program or any location. To do so, assume you untared PyRIC to /home/bob/PyRIC. Create a text file named pyric.pth with one line

/home/bob/PyRIC

and save this file to /usr/lib/python2.7/dist-packages (or /usr/lib/python3/dist-packages if you want to try it in Python 3).

3 Using PyRIC

As stated previously, PyRIC provides a set of functions to interact with your system's radio(s) and the ability to interact directly with the kernel through netlink and ioctl sockets.

3.1 Interacting with the Wireless Core and Wireless NICs: pyw.py

If you can use iw, you can use pyw. The easist way to explain how to use pyw is with an example. Imagine your wireless network, on ch 6, has been experiencing difficulties lately and you want to capture some traffic to analyse it. Listing 1 shows how to set up a wireless pentest environment.

```
 1: import pyric                                  # pyric error (and ecode EUNDEF)
 2: from pyric import pyw                         # for iw functionality
 3: import pyric.utils.hardware as hw             # for chipset/driver
 4: from pyric.utils.channels import rf2ch        # rf to channel conversion
 5:
 6: dev = 'wlan0'
 7: ifaces = pyw.interfaces()
 8: wifaces = pyw.winterfaces()
 9: if dev not in ifaces:
10:     print "Device {0} is not valid, use one of {1}".format(dev,ifaces)
11:     return
12: elif dev not in wifaces:
13:     print "Device {0} is not wireless, use one of {1}".format(dev,wifaces)
14:
15: print "Regulatory Domain currently: ", pyw.regget()
16: dinfo = pyw.devinfo(dev)
17: card = dinfo['card']
18: pinfo = pyw.phyinfo(card)
19: driver = hw.ifdriver(card.dev)
20: chipset = hw.ifchipset(driver)
21:
22: pyw.down(card)
23: pyw.macset(card,'00:03:93:57:54:46')
24:
25: msg = "Using {0} currently in mode: {1}\n".format(card,dinfo['mode'])
26: msg += "\tDriver: {0} Chipset: {1}\n".format(driver,chipset)
27: if dinfo['mode'] == 'managed':
28:     msg += "\ton channel {0} width {1}\n".format(rf2ch(dinfo['RF']),
29:                                                  dinfo['CHW'])
30: msg += "\tSupports modes {0}\n".format(pinfo['modes'])
31: msg += "\tSupports commands {0}".format(pinfo['commands'])
32: msg += "\thw addr {0}".format(pyw.macget(card))
33: print msg
34:
35: pdev = 'pent0'
36: for iface in pyw.ifaces(card):
37:     pyw.devdel(iface[0])
38: pcard = pyw.devadd(card, pdev, 'monitor')
39: pyw.up(pcard)
40: pyw.chset(pcard,6,None)
41:
42: # DO STUFF HERE
43:
44: pyw.devdel(pcard)
45:
46: card = pyw.devadd(card,card.dev,dinfo['mode'])
47: pyw.macset(card,dinfo['mac'])
48: pyw.up(card)
```

Listing 1: Setting up a Wireless Pentest Environment

Listing 1 attempts to show most of the available pyw functions in use and is the basic shell used in another project, Wraith[4], to instantiate a wireless (802.11) sensor - (for a full listing of all pyw functions see Appendix A) - with scanning capabilities. Lines 1 and 2 should always be included as they import the pyric error and pyw functions. Line 3 imports hardware which provides the ifchipset and ifdriver functions and Line 4 imports the rf2ch conversion function.

In lines 6 through 13, the device wlan0 is confirmed wireless and lines 16 through 20 a Card object for 'wlan0' is created and details about the interface are printed. Next, the mac address of wlan0 is changed on lines 23. Note, the device is brought down first.

More information on the device is printed in lines 25 through 33. Starting on line 35, a device named 'pent0' is created in monitor mode. First in lines 36 and 37, all interfaces on the same phy are deleted [2] before creating the new device, bringing the card up and setting it to channel 6 NOHT.

Restoring the device starts on line 45, where the virtual interface is deleted, the previous interface is restored and the mac address is reset.

3.1.1 One-time vs Persistent Sockets

The example in Listing 1 uses one-time sockets (netlink and ioctl). When using iw, there are several things that occur prior to the actual command or request being submitted. First, iw creates a netlink socket. Then, iw will request the family id for nl80211. The relative time spent doing this is neglible but, it is redundant and it may become noticeable in programs that repeatedly use the Netlink service. Once comlete, iw closes the socket. In some cases, the ifindex of the device is needed and iw will also initiate an ioctl call to retrieve it. PyRIC eliminates these redundancies by using a global variable in pyw that stores the family id after the first time it is requested and by providing callers the option to use persistent sockets.

- **One-time Sockets** Similar to iw. The command, creates the netlink socket (or ioctl socket), composes the message, sends the message and receives the response, parses the results, closes the socket and returns the results to the caller. At no time does the caller need to be aware of any underlying Netlink processes or structures.
- **Persistent Sockets** Communication and parsing only. The onus of socket creation and deletion is on the caller which allows them to create one (or more) socket(s). The pyw functions will only handle message construction, message sending and receiving and message parsing.

The caller needs to be cognizant of whether the function requires a netlink or ioctl socket. Passing the wrong type will result in an error.

NOTE: One must remember that there is an upper limit to the number of open netlink sockets. It is advised to use one-time functions as much as possible and save the use of persistent sockets for use in code that repeatedly makes use of netlink.

The latest version of pyw.py (v 0.1.*) implements this functionality through the use of what I call templates[3], Listing 2 and stubs Listing 3.

[2] we have found that it is better to delete all interfaces on the same phy ensuring that external processes don't interfere with the new device

[3] I use templates and stubs for the lack of any better naming convention

```
def fcttemplate(arg0,arg1,..,argn,*argv):
    # put parameter validation (if any) here
    try:
        nlsock = argv[0]
    except IndexError:
        return _nlstub_(fcttemplate,arg0,arg1,...,argn)

    # command execution
    ...
    return results
```

Listing 2: A Basic Netlink Function Template

The template function in Listing 2 checks if argv has a netlink socket[4] at index 0. If so, it proceeds to execution. If there is no socket, the stub is executed which creates one. If something other than a netlink socket is at argv[0], an error will be raised during execution.

```
def _nlstub_(fct,*argv):
    nlsock = None
    try:
        nlsock = nlsock = nl.nl_socket_alloc()
        argv = list(argv) + [nlsock]
        return fct(*argv)
    except pyric.error:
        raise # catch & release
    finally:
        if nlsock: nl.nl_socket_free(nlsock)
```

Listing 3: Function _nlstub_

The stub function, Listing 3 allocates a netlink socket, executes the original (now with a netlink socket) and then destroys the netlink socket.

```
1: import pyric                          # pyric errors
2: from pyric import pyw                 # for iw functionality
3: from pyric.lib import libnl as nl # for netlink sockets
4:
5: nlsock = nl.nl_socket_alloc(timeout=1)
6: card = pyw.getcard('wlan0',nlsock)
7: print pyw.devmodes(card,nlsock)
8: nl.nl_socket_free(nlsock)
```

Listing 4: Using Persistent Sockets

Listing 4, shows the creation of a persistent netlink socket that is used in the creation of a card and in retrieved the card's supported modes.

Use Python's built in help features on pyw functions or see Appendex A to determine what type of socket is needed.

[4] ioctl calls operate in the same manner

3.2 Additional Tools

In the utils directory, PyRIC includes channels.py, hardware.py, rfkill.py and ouifetch.py. These provide a port of rfkill, channel/frequency enumeration and device chipset, driver retrieval as well as some mac address functions. More information can be found in the Appendices and in README.md.

3.3 Interacting with the Kernel: libnl.py and libio.py

The kernel interfaces, libnl.py and libio.py are located in the lib directory. They handle socket creation/deletion, message creation/parsing and kernel communication. Aside from creating and deleting persistent sockets, there is little need to access their functions unless you plan on extending pyw functionality. As such, a further discussion of libnl.py and libio.py can be found in the next section.

4 Extending PyRIC

You may find that pyw does not offer some of the functionality you need. Using libnl.py and/or libnl.io, additional functionality can be added to your program.

It is helpful if the reader has a basic knowledge of netlinks. For a review, see "Communicating between the kernel and user-space in Linux using Netlink Sockets" [3].

4.1 Porting C

All Python ports of C header files can be found in the net directory. C Enums and #defines are ported using constants. C structs are ported using three Python structures and the Python struct package:

1. a format string for packing and unpacking the struct
2. a constant specifying the size of the struct in bytes
3. a function taking the attributes of the struct as arguments and returning a packed string

Listing 5 shows the C definition of the nlmsghdr found in netlink.h.

```
struct nlmsghdr {
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};
```

Listing 5: C Struct nlmsghdr

And Listing 6 shows the ported version in Python.

```
nl_nlmsghdr = "IHHII"
NLMSGHDRLEN = struct.calcsize(nl_nlmsghdr)
def nlmsghdr(mlen,nltype,flags,seq,pid):
    return struct.pack(nl_nlmsghdr,NLMSGHDRLEN+mlen,nltype,flags,seq,pid)
```

Listing 6: Corresponding Python Definition

When using pyw, dealing with these structures is handled transparently by libnl.py and libio.py. When extending or customizing pyw, a basic understanding of the definitions in netlink_h.py, genetlink_h.py and if_h.py.

4.2 Input/Output Control (ioctl)

PyRIC provides more than just iw-related functions, it also implements functions from ifconfig and iwconfig. These command line tools still use ioctl (or the proc directory). For example, interfaces() reads from '/proc/net/dev' to retrieve all system interfaces and winterfaces() use ioctl to check if a device is wireless. Input/Output control calls have only been used when there was no viable alternative and, it should not be necessary to have to add any further ioctl commands. If you find that you need an ioctl related command, search through if_h.py for the appropriate structure and add it's definitions to ifreq.

4.3 Netlink and nl80211

Documentation on Netlink, and nl80211 in particular, is so minimal as to be neglible. The clusterfuck of code and lack of comments in the iw source tree make it impossible to use as any sort of roadmap. Fortunately Thomas Graf's site[2] has excellent coverage of libnl, the Netlink library. Using this as a reference, a simple Netlink parser was put together which later became libnl.py. Using the command line tool strace and libnl.py, Netlink messages could be dissected and analyzed.

Let us consider adding a virtual interface with the command:

```
sudo iw phy0 interface add test0 type monitor
```

First, we need to see what is going on under the covers. Using strace:

```
strace -f -x -s 4096 iw phy0 interface add test0 type monitor
```

from a terminal will give a you a lot of output, most irrelevant (to us). Scroll through this until the netlink socket creation as highlighted in Figure 1. You can see that a socket of type PF_NETLINK is created and the send/receive buffers are set to 32768.

What we want to analyze are the messages sent and received over the netlink socket. In Figure 1, iw is requesting the family id for nl80211. This id will be used in subsequent requests related to nl80211 as we will see shortly. The return message gives the nl80211 family id as 26 and returns other nl80211 attributes. This is handled by the private function _familyid_ in pyw.py.
Figure 2 shows the add interface message being sent to the kernel.

We are interested in the byte sequence following msg_iov(1). Copy this and paste into in a python variable as in Listing 7 and pass it to the function nlmsg_fromstream which parses the byte stream and returns the GENLMsg.

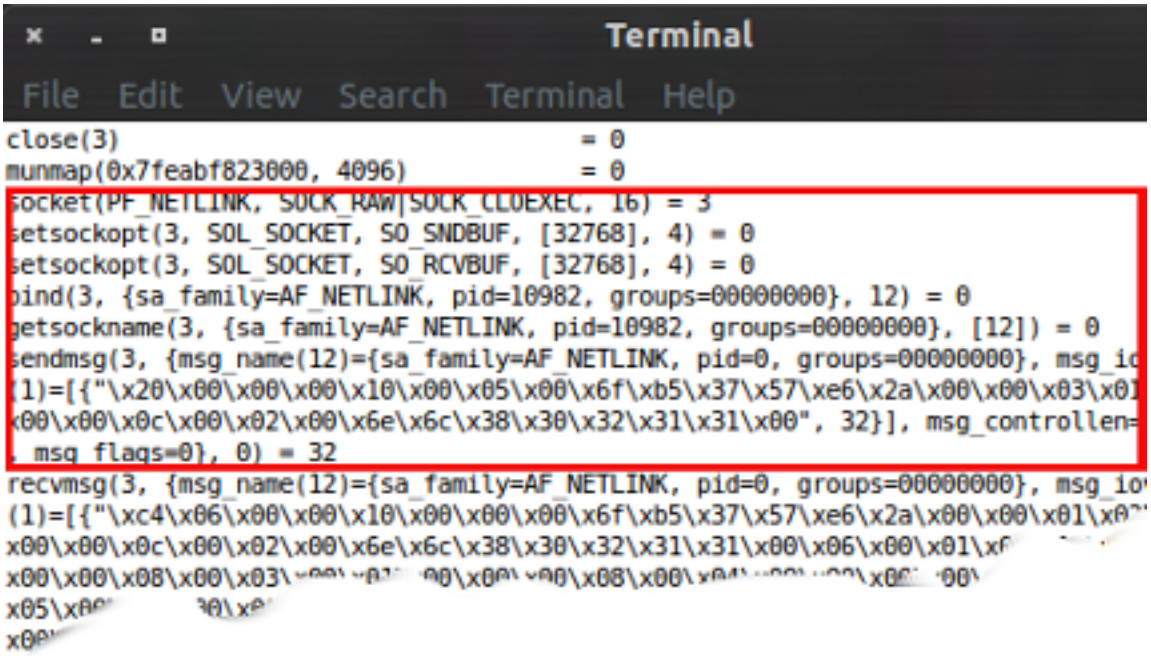

Figure 1: Netlink socket creation

```
>>> from pyric.lib import libnl as nl
>>> sent = "\x30\x00\x00\x00\x1a...\x00\x00"
>>> msg = nl.nlmsg_fromstream(sent)
>>> msg
nlmsghdr(len=48,type=26,flags=5,seq=1463268720,pid=10982)
genlmsghdr(cmd=7)
attributes:
  0: type=1,datatype=3
  value=0
  1: type=4,datatype=5
  value=test0
  2: type=5,datatype=3
  value=6
```

Listing 7: Parsing netlink messages

The first thing to notice is nlmsghdr type = 26, which of course is nl80211 family id. The rest of the nlmsghdr components len, flags, seq, and pid are handled by libnl.py although you can supply your own flags if desired. At this time, you can manually look up what values the cmd, type and datatype correspond to in nl80211_h.py and netlink_h.py or you can use the tools provided in nlhelp.py.

```
>>> from pyric.net.netlink_h import NLA_DATATYPES
>>> from pyric.docs import nlhelp
>>> nlhelp.cmdbynum(7)
u'@NL80211_CMD_NEW_INTERFACE'
>>>
>>> for attr in msg.attrs:
...     print nlhelp.attrbynum(attr[0]), NLA_DATATYPES[attr[2]], attr[1]
```

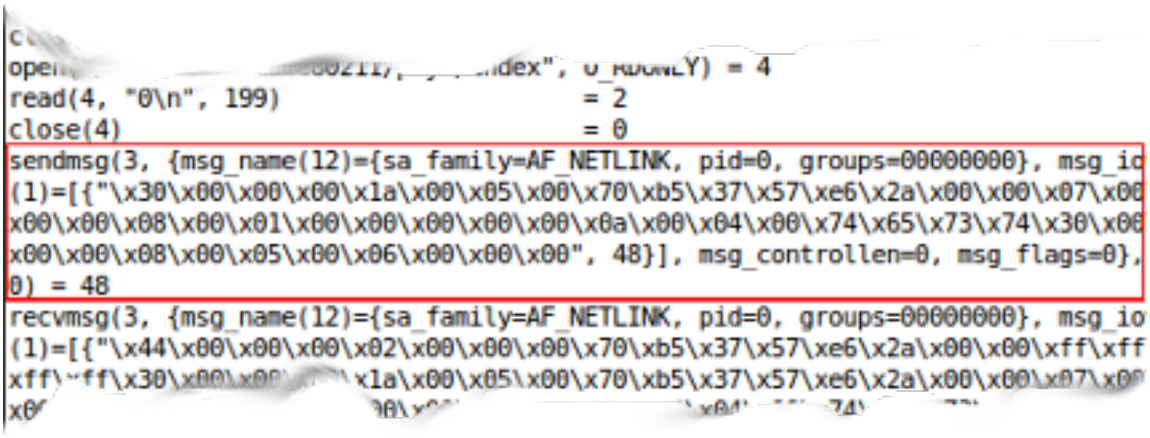

Figure 2: Netlink sendmsg

```
...
@NL80211_ATTR_WIPHY u32 0
@NL80211_ATTR_IFNAME string test0
@NL80211_ATTR_IFTYPE u32 6
>>>
>>> from pyric.net.wireless.nl80211_h import NL80211_IFTYPES
>>> NL80211_IFTYPES[6]
'monitor'
```

Listing 8: Parsing netlink messages continued

In Listing 8 command number 7 corresponds to NL80211_CMD_NEW_INTERFACE and the attributes that need to be passed to the kernel are NL80211_ATTR_WIPHY, NL80211_ATTR_IFNAME and NL80211_ATTR_IFTYPE. The IFTYPE is also known as the mode i.e. 'monitor' which can be found in nl80211_h.py NL80211_IFTYPES. We don't parse the return message from the kernel but, it follows the same SOP. In this case, it returns the attributes of the new virtual interface.

With this information, we can now code our function. Recall the fcttemplate as defined in Listing 2 and fill in the command execution as shown in Listing 9.

```
# construct the message
msg = nl.nlmsg_new(nltype=_familyid_(nlsock),
                   cmd=nl80211h.NL80211_CMD_NEW_INTERFACE,
                   flags=nlh.NLM_F_REQUEST | nlh.NLM_F_ACK)
nl.nla_put_u32(msg, card.phy, nl80211h.NL80211_ATTR_WIPHY)
nl.nla_put_string(msg, vdev, nl80211h.NL80211_ATTR_IFNAME)
nl.nla_put_u32(msg, IFTYPES.index(mode), nl80211h.NL80211_ATTR_IFTYPE)

# send, receive and parse return results, returning the new Card
nl.nl_sendmsg(nlsock, msg)
rmsg = nl.nl_recvmsg(nlsock) # success returns new device attributes
return Card(card.phy, vdev, nl.nla_find(rmsg, nl80211h.NL80211_ATTR_IFINDEX))
```

Listing 9: Coding the function

We construct a new GENLMsg passing the nl80211 family id, the command we got earlier and flags specifying that this is a request and we want to get an ACK back[5]. Now, add each attribute to the message. Note the order: value, then attribute. With the message constructed, send it to the kernel, get the results, parse and return them.

Rather simple, in fact the hardest part is figuring out what to send to the kernel. Everything else is handled behind the scenes by libnl.py.

Appendix A API: pyw.py

A.1 Constants

- **_FAM80211ID_**: Global netlink family id of nl80211. Do not touch
- **IFTYPES**: redefined (from nl80211_h.py) interface modes
- **MNTRFLAGS**: redefined (from nl80211_h.py) monitor mode flags
- **TXPOWERSETTINGS**: redefined (from nl80211_h.py) power level settings

A.2 Objects/Classes

Card A wrapper around a tuple `t = (physical index,device name,interface index)` which exposes the following properties through '.':

- **phy**: physical index
- **dev**: device name
- **idx**: interface index (ifindex)

Because the underlying Netlink calls will sometimes require the physical index, sometimes the device name, and sometimes the ifindex, pyw functions accept a Card, object. This allows callers to use pyw functions without having to remember which identifier the function requires. However, in some cases the function requires a dev or accepts both. See the next section on functions.

While callers could create their own Cards, it is recommend to use one of the following

- **pyw.getcard** returns a Card object from a given dev
- **pyw.devinfo** returns the dict info where info['card'] is the Card object. This function will take either a card or a dev
- **pyw.devadd** returns a new Card object
- **pyw.devadd** returns a new Card object
- **pyw.ifaces** returns a list of tuples t = (Card,mode) sharing the same phy as a given device to do so. It is also recommended to periodically validate the Card. On some cheaper usb wireless nics, there are periodic disconnects which results in a new phy and ifindex.

[5] libnl.py always forces an ACK and handles the underlying process of receiving it

A.3 Functions

- interfaces(): (ifconfig), type: filesystem, returns list of all network dev
- isinterface(dev): (ifconfig <dev>) type: filesystem, check dev is an interface
- winterfaces([iosock]): (iwconfig), type: ioctl, list wireless interfaces
- iswireless(dev,[iosock]): (iwconfig <dev>), type: ioctl, check dev is a wireless interface
- phylist(): (iw phy | grep wiphy) type: N/A, list phy indexes and phy names present on system
- regget([nlsock]: (iw reg get), type: netlink, get regulatory domain
- regset(rd,[nlsock]): (iw reg set <rd>), type: netlink, set regulatory domain to rd
- getcard(dev,[nlsock]) (N/A), type: hybrid netlink and ioctl: get a Card object for dev
- validcard(card,[nlsock]): (N/A), type: (hyrbrid netlink and ioctl), verify card is still valid
- macget(card,[iosock]): (ifconfig), type: ioctl, determine if card is up or down
- macset(card,mac,[iosock]): (ifconfig card.<dev> hw ether <mac>), type: ioctl, set card's hw address to mac
- isup(card,[iosock]): (ifconfig card.<dev>)
- up(card,[iosock]) (ifconfig card.<dev> up), type: ioctl, bring card up
- down(card,[iosock]): (ifconfig card.<dev> down), type: ioctl, bring card down
- isblocked(card): (rfkill list <rfkill_idx>): type N/A returns tuple (Soft Block State, Hard Block State)
- block(card): (rfkill block <rfkill_idx>) type: N/A, soft blocks card
- unblock(card): (rfkill unblock <rfkill_idx>) type: N/A, removes the soft block on card
- pwrsaveget(card,[nlsock]) (iw dev card.<dev> get power_save) type: netlink get card's power save state True = on, False = off
- pwrsaveset(card,on,[nlsock]) (iw dev card.<dev> set power_save <on>) type: netlink set card's power save state True = on, False = off
- covclassget(card,[nlsock]) (iw phy card.<phy> get coverage <cc>) type: netlink get card's coverage class
- covclassset(card,cc,[nlsock]) (iw phy card.<phy> set coverage <cc>) type: netlink set card's coverage class
- retryshortget(card,[nlsock]) (iw phy card.<phy> info | grep 'retry short') type:netlink get card's retry short limit
- retryshortset(card,lim,[nlsock]) (iw phy card.<phy> set retry short <lim>) type:netlink set card's retry short limit. NOTE: although 255 is specified as the max limit for this and the long retry, kernel v4 will not allow it.

- retrylongget(card,[nlsock]) (iw phy card.<phy> info | grep 'retry long') type:netlink get card's retry long limit
- retrylongset(card,lim,[nlsock]) (iw phy card.<phy> set retry long <lim>) type:netlink set card's retry long limit
- rtsthreshget(card,[nlsock]) (iw phy card.<phy> info | grep rts) type: netlink set card's RTS threshold
- rtsthreshset(card,thresh,[nlsock]) (iw phy card.<phy> set rts <thresh>) type: netlink set card's RTS threshold
- fragthreshget(card,[nlsock]) (iw phy card.<phy> info | grep frag) type: netlink get card's fragmentation threshold
- fragthreshset(card,thresh,[nlsock]) (iw phy card.<phy> set frag <thresh>) type: netlink set card's fragmentation threshold
- inetget(card,[iosock]): (ifconfig card.<dev>), type: ioctl, get ip4 address, netmask and broadcast address of card
- inetset(card,ipaddr,netmask,broadcast,[iosock]): (ifconfig card/<dev> <ipaddr> netmask <netmask> broadcast <broadcast>), type: ioctl, set the interface addresses of the card
- ip4set(card,ipaddr,[iosock]): (ifconfig card.<dev> <ipaddr>), type: ioctl, set the card's ip4 address
- netmaskset(card,netmask,[iosock]): (ifconfig card.<dev> netmask <netmask>), type: ioctl, set the card's netmask
- broadcastset(card,broadcast,[iosock]): (ifconfig card.<dev> broadcast <broadcast>), type: ioctl, set the card's broadcast address
- devfreqs(card,[nlsock]): (iw phy card.phy info), type: netlink, get card's supported frequencies
- devchs(card,[nlsock]): (iw phy card.phy info), type: netlink, get card's supported channels
- devstds(card,[nlsock]): (iwconfig card.<dev> | grep IEEE), type: nlsock, returns a list of card's 802.11 supported standards by letter designator
- devmodes(card,[nlsock]): (iw phy card.phy info | grep interface), type: netlink, get card's supported modes
- devcmds(card,[nlsock]): (iw phy card.phy info | grep commands), type: netlink, get card's supported commands
- ifinfo(card,[iosock]): (ifconfig card.<dev>), type: ioctl, get hardware related info for card
- devinfo(card,[nlsock]): (iw dev card.<dev> info), type: netlink, get info for dev
- phyinfo(card,[nlsock]): (iw phy card.<phy> info), type: netlink, get info for phy
- ifaces(card,[nlsock]): (APX iw card.dev | grep phy#), type: netlink, get all cards (w/ modes) of interfaces sharing the same phy as card

- txset(card,pwr,lvl,[nlsock]) (iw phy phy0 set txpower <lvl> <pwr>), type: netlink,sets the tx power to pwr (in dBm) with level setting lvl
- txget(card,[iosock]): (iwconfig card.<dev> | grep Tx-Power card), type: ioctl, get card's transmission power
- chget(card,[nlsock]): (iw dev <card.dev> info | grep channelS), type: netlink, get card's current channel (only works for cards in mode managed)
- chset(card,ch,chw,[nlsock]): iw phy <card.phy> set channel <ch> <chw>), type: netlink, set card's current channel to ch with width chw
- freqset(card,rf,chw,[nlsock]): iw phy <card.phy> set freq <rf> <chw>), type: netlink, set card's current frequency to rf with width chw
- devmodes(card,[iosock]): (iw phy card.<phy>), type: netlink, get modes supported by card
- modeset(card,mode,[flags],[nlsock]): (iw dev card.<dev> set type <mode> [flags]), type: netlink, set card's mode to mode with flags (if mode is monitor)
- modeget(card[nlsock]): (iw dev card.<dev> info | grep mode), type: netlink, get card's mode
- devset(card,ndev,[nlsock]): (N/A) sets the dev (name) of card to ndev
- phyadd(card (or phy),vnic,mode,[flags],[nlsock]): (iw phy card.<phy> interface add <vnic> type <mode> flags <flags>)[6], type: netlink, creates a new virtual interface with dev vdev, in mode and using flags. Note: flags are only supported when creating a monitor mode
- devadd(card (or ifindex),vnic,mode,[flags],[nlsock]): (iw phy card.<dev> interface add <vnic> type <mode> flags <flags>), type: netlink, creates a new virtual interface with dev vdev, in mode and using flags. Note: flags are only supported when creating a monitor mode
- devdel(card,[nlsock]): (iw card.<dev> del), type: netlink, deletes card
  - isconnected(card, [nlsock]): (iw dev card.<dev> info | grep channel), type: netlink, determines if card is connected
  - disconnect(card, [nlsock]): (iw dev card.<dev> disconnect), type: netlink, disconnects card from AP
  - link(card, [nlsock]): (iw dev card.<dev> link), type: netlink, displays link specific details, i.e. AP details that card is connected to
  - stainfo(card, mac, [nlsock]): (iw dev card.<dev> link) type: netlink, displays tx, rx metrics of the AP that card is connected to
  - _hex2mac_(v): returns a ':' separated mac address from byte stream v
  - _mac2hex_(v): returns a hex string corresponding to mac address v
  - _hex2ip4_(v): returns a '.' separated ip4 address from byte stream v
  - _validip4_(addr): determines if addr is a valid ip4 address
  - _validmac_(addr): determines if addr is a valid mac address

[6]There is a bug in some kernel v4.4.0-x where the given dev name is ignored and a system chosen one is used instead. See https://wraithwireless.wordpress.com. Whenever possible, use devadd to create interfaces instead.

- _issetf_(flags,flag): determines if flag is set in flags
- _setf_(flags,flag): set flag in flags to on
- _unsetf_(flags,flag): set flag in flags to off
- _familyid_(nlsock): returns and sets the Netlink family id for nl80211, only called once per module import
- _ifindex_(dev,[iosock]): returns dev's ifindex
- _flagsget_(dev,[iosock]): get's the dev's interface flags
- _flagsset_(dev,flags,[iosock]): set's the dev's interface flags
- _iftypes_(i): returns the mode corresponding to i
- _bands_(band): futher parses band attribute returns dict of bands containting rf information and rate information
- _band_rates_(rs): extracts list of rates from the unpacked rates rs
- _band_rfs_(rfs): extracts list of RFs (and other data) from the unpacked frequencies rfs
- _unparsed_rfs_(band): (legacy) returns a list of frequencies from the unparsed byte string band
- _commands_(command): converts the list of numeric commands to a list of commands as strings
- _ciphers_(cipher): returns a list of ciphers from the packed byte string cipher
- _rateinfo_(ri): returns parsed rate info from the packed byte string ri
- _iostub_(fct,*argv): ioctl stub function, calls fct with parameter list argv and an allocated ioctl socket
- _nlstub_(fct,*argv): netlink stub function, calls fct with parameter list argv and an allocated netlink socket

Appendix B API: channels.py

Channel, Frequency enumeration and conversions can be found in channels.py.

B.1 Constants

1. **CHTYPES**: imported channel types from nl80211_h
2. **CHWIDTHS**: imported channel widths from nl80211_h
3. **ISM_24_C2F**: Dict containing ISM channel (key) to frequency (value) pairs
4. **ISM_24_F2C**: Dict containing ISM frequency (key) to channel (value) pairs
5. **UNII_5_C2F**: Dict containing UNII 5Ghz channel (key) to frequency (value) pairs
6. **UNII_5_F2C**: Dict containing UNII 5Ghz frequency (key) to channel (value) pairs
7. **UNII_4_C2F**: Dict containing UNII upper 4Ghz channel (key) to frequency (value) pairs
8. **UNII_4_F2C**: Dict containing UNII upper 4Ghz frequency (key) to channel (value) pairs

B.2 Functions

1. channels(): returns a list of all channels
2. freqs(): returns a list of all frequencies
3. ch2rf(c): convert channel c to frequency
4. rf2ch(f): convert frequency f to channel

Appendix C API: hardware.py

Hardware related: driver, chipset, manufacturer and mac address utility functions can be found in device.py.

C.1 Constants

1. **dpath**: path to system device details
2. **drvpath**: path to device drivers

C.2 Functions

1. oui(mac): returns the oui portion of address <mac>
2. ulm(mac): returns the ulm portion of address <mac>
3. manufacturer(ouis,mac): returns the manufacturer name of <mac> given the dict of <ouis>
4. randhw([ouis]): returns a random mac address. If the dict ouis is specified will select a random oui from the dict otherwise will generate one
5. ifcard(dev): returns the device driver and chipset
6. ifdriver(dev): returns the device driver
7. ifchipset(driver): returns the chipset associated with driver

Appendix D API: ouifetch.py

The file ouifetch.py retrieves and saves a tab seperated file of oui to manufacturer name for use by hardware.py functions. From a command line, type:

D.1 Constants

1. **OUIURL**: url of IEEE oui file
2. **OUIPATH**: path to default location PyRIC oui.txt file

D.2 Functions

1. load([opath]): returns a dict of oui:manufacturer key->value pairs stored in the text file at opath. If opath is not specified, uses the default
2. fetch([opath]): retrieves oui.txt from the IEEE website, parses the files and stores the results in a PyRIC friendly format in opath. If opath is not specified, uses the default. User must have root permissions in order to write to default opath

Appendix E API: rfkill.py

A port of the command line tool rfkill, rfkill.py writes and reads rfkill_event structures to /dev/rfkill using fcntl providing functionality to block and unblock devices.

E.1 Constants

1. **RFKILL_STATE**: list of boolean values corresponding to blocked, unblocked

E.2 Functions

1. rfkill_list(): corresponds to rkill list, returns a dict of dicts name -> {idx, type, soft, hard}. If type is 'wireless', then name will be of the form phy<n> such that n is the physical index of the wireless card
2. rfkill_block(idx): soft blocks the device at rfkill index idx
3. rfkill_blockby(rtype): soft blocks all devices of type rtype
4. rfkill_unblock(idx): turns off the soft block at rfkill index idx
5. rfkill_unblockby(rtype): turns off the soft blocks of all devices of type rtype
6. soft_blockedidx: determines soft block state of device at rfkill index idx
7. hard_blockedidx: determines hard block state of device at rfkill index idx
8. getidx(phy): returns the rfkill index of the device with physical index phy
9. getname(idx): returns the name of the device at rfkill index idx
10. gettype(idx): returns the type of the device at rfkill index idx

Appendix F API: libnl.py

Providing libnl similar functionality, libnl.py provides the interface between pyw and the underlying nl80211 core. It relates similarily to libnl by providing functions handling netlink messages and sockets and where possible uses similarly named functions as those libnl to ease any transitions from C to PyRIC. However, several liberties have been taken as libnl.py handles only nl80211 generic netlink messages.

F.1 Constants

- **BUFSZ** default rx and tx buffer size

F.2 Classes/Objects

The two classes in libnl.py, NLSocket and GENLMsg, discussed in the following sections subclass Python's builtin dict. This has been done IOT to take advantage of dict's already existing functions and primarily their mutability and Python's 'pass by name' i.e. modifications in a function will be reflected in the caller. This makes the classes very similar to the use C pointers to structs in libnl.

F.2.1 NLSocket

NLSocket is a wrapper around a netlink socket which exposes the following properties through '.':

- **sock**: the actual socket
- **fd**: the socket's file descriptor (deprecated)
- **tx**: size of the send buffer
- **rx**: size of the receive buffer
- **pid**: port id
- **grpm**: group mask
- **seq**: sequence number
- **timeout**: socket timeout

and has the following methods:

- incr(): increment sequence number
- send(pkt): sends pkt returning bytes sent
- recv(): returns received message (will block unless timeout is set)
- close(): close the socket

NLSockets are created with nl_socket_alloc and must be freed with nl_socket_free. See Section F.3.

F.2.2 GENLMsg

GENLMsg is a wrapper around a dict with the following key->value pairs:

- **len**: total message length including the header
- **nltype**: netlink type
- **flags**: message flags
- **seq**: seq. #
- **pid**: port id
- **cmd**: generic netlink command

- **attrs**: list of message attributes. Each attribute is a tuple t = (attribute,value,datatype) where:
  - **attribute**: netlink attribute type i.e. CTRL_ATTR_FAMILY_ID
  - **value**: the unpacked attribute value
  - **datatype**: datatype of the attribute as defined in nelink_h i.e. NLA_U8

NOTE: as discussed below, on sending, the seq. # and port id are overridden with values of the netlink socket.

GENLMsg exposes the following properties:

- **len**: length of the message (get only)
- **vers**: returns 1 (default version) (get only)
- **nltype**: message content i.e. generic or nl80211 (get or set)
- **flags**: message flags (get or set)
- **seq**: current sequence # (get or set)
- **pid**: port id (get or set)
- **cmd**: netlink command (get or set)
- **attrs**: attribute list (get only)
- **numattrs**: number of attributes (get only)

GENLMsg has the following methods:

- __repr__(): returns a string representation useful for debugging
- tostream(): returns a packed netlink message

  There are two methods of creating a GENLMsg. Create a new message (to send) with nlmsg_new and create a message from a received packet with nlmsg_fromstream. These are discussed below.

F.3 Functions

- **Netlink Socket Related**
  - nl_socket_alloc(pid,grps,seq,rx,tx,timeout): creates a netlink socket with port id = pid, group mask = grps, initial seq. # = seq, send and receive buffer size = tx and rx respectively and blocking timeout = timeout
  - nl_socket_free(sock): closes the socket
  - nl_socket_pid(sock): (deprecated for NLSocket.pid) returns the port id
  - nl_socket_grpmask(sock): (deprecated for NLSocket.grpmask) returns the group mask

    * nl_sendmsg(sock,msg,override=False): sends the netlink msg over socket. NOTE: NLSockets will automatically set the port id and seq. # regardless of their value in the message. If override is True, the message's pid and seq. # will be used instead.
    * nl_recvmsg(sock): returns a GENLMsg or blocks unless the socket's timeout is set. Should only be called once per every nl_sendmsg.
  - **Netlink Message Related**
    * nlmsg_new(nltype=None,cmd=None,pid=None,flags=None,attrs=None): creates a new GENLMsg with zero or more attributes defined.
    * nlmsg_fromstream(stream): parses the message in stream returning the corresponding GENLMsg
    * nla_parse(msg,l,mtype,stream,idx): parses the attributes in stream appending them to the attribute list of message where msg = the GENLMsg, l = the total length of the message, mtype = the message content (i.e. netlink type) stream is the original byte stream and idx is the index of the start of the attribute list
    * nla_parse_nested(nested): returns the list of packed nested attributes extracted from the stream nested. Callers must unpack and parse the returned attributes themselves
    * nla_put(msg,v,a,t): appends the attribute a, with value v and datatype t to the msg's attribute list
    * nla_put_<DATATYPE>(msg,v,a): eight specialized functions that append attribute a with the value v and type <DATATYPE> to msg's attribute list
    * nla_putat(msg,i,v,a,d): puts attribute a, with value v and datatype d at index i in msg's attribute list.
    * nla_pop(msg,i): removes the attribute tuple at index i, returning the popped tuple
    * nla_find(msg,a,value=True): returns the first attribute a in msg's attribute list. If value returns only the value otherwise returns the attribute tuple
    * nla_get(msg,i,value=True): returns the attribute at index i. If value returns only the value otherwise returns the attribute tuple
    * _nla_strip(v): (private) strips padding bytes from the end of v
    * _attrpack(a,v,d): (private) packs the attribute tuple
  - _maxbufsz_(): (private) returns the maximum allowable socket buffer size

Appendix G API: libio.py

A very basic interface to ioctl, libio provides socket creation, deletion and transfer.

G.1 Functions

1. io_socket_alloc(): returns an ioctl socket
2. io_socket_free(iosock): closes the ioctl socket iosock
3. io_transfer_(iosock,flag,ifreq): sends the ifreq structure with sockios control call flag to the kernel and returns the received ifreq structure

Appendix H Copyright and License

PYRIC: Python Radio Interface Controller v0.1.5

References

[1] Gnu general public license, June 2007.

[2] Graf, T. Netlink library (libnl), May 2011.

[3] Pablo Neira Ayuso, Rafael M. Gasca, L. L. Communicating between the kernel and user-space in linux using netlink sockets. *Software - Practice And Experience 40* (August 2010), 797–810.

[4] Patterson, D. V. Wireless reconnaissance and intelligent target harvesting, April 2016.