PF_RING User Guide

Linux High Speed Packet Capture

Version 4.6
February 2011

1.Table of Contents

2. Introduction

PF_RING is a high speed packet capture library that turns a commodity PC into an efficient and cheap network measurement box suitable for both packet and active traffic analysis and manipulation. Moreover, PF_RING opens totally new markets as it enables the creation of efficient application such as traffic balancers or packet filters in a matter of lines of codes.

This manual is divided in two parts:

- PF_RING installation and configuration.
- PF_RING SDK.

2.1. What's New with PF_RING User's Guide?

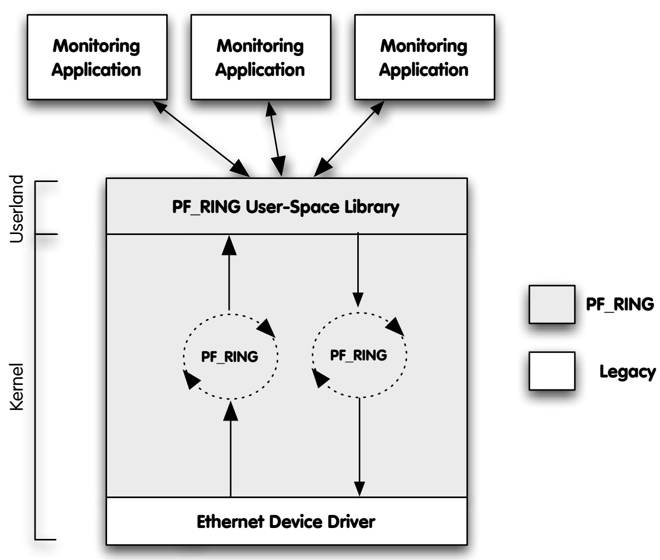
- Release 4.6 (February 2011)
    - Updated guide to PF_RING version 4.6.x.

- Release 1.1 (January 2008)
    - Described PF_RING plugins architecture.

- Release 1.0 (January 2008)
    - Initial PF_RING users guide.

3.Welcome to PF_RING

PF_RING's architecture is depicted in the figure below.



The main building blocks are:

- The accelerated kernel driver that provides low-level packet copying into the kernel PF_RINGs.
- The user space PF_RING SDK that provides transparent PF_RING-support to user-space applications.

- Specialized PF_RING-aware drivers (optional) that allow to further enhance packet capture by efficiently copying packets from the driver to PF_RING without passing through the kernel data structures. Please note that PF_RING can operate with any NIC driver, but for maximum performance it is necessary to use these specialized drivers that can be found into the kernel/ directory part of the PF_RING distribution. Note that the way drivers pass packets to PF_RING is selected when the PF_RING kernel module is loaded by means of the transparent_mode parameter.

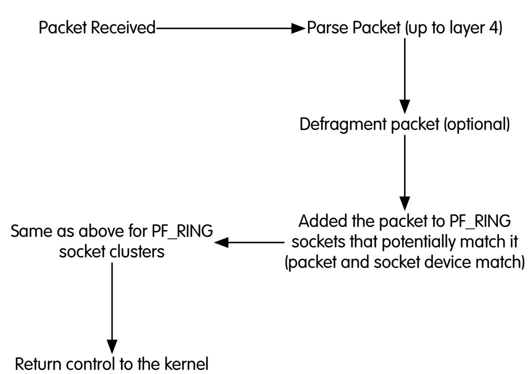
PF_RING implements a new socket type (named PF_RING) on which user-space applications can speak with the PF_RING kernel module. Applications can obtain a PF_RING handle, and issue API calls that are described later in this manual. A handle can be bound to a:

- Physical network interface.
- A RX queue, only on multi-queue network adapters.
- To the 'any' virtual interface that means packets received/sent on all system interfaces are accepted.

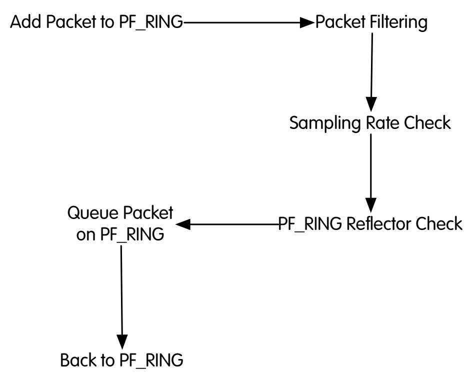
As specified above, packets are read from a memory ring allocated at creation time. Incoming packets are copied by the kernel module to the ring, and read by the user-space applications. No per-packet memory allocation/deallocation is performed. Once a packet has been read from the ring, the space used in the ring for storing the packet just read will be used for accommodating future packets. This means that applications willing to keep a packet archive, must store themselves the packets just read as the PF_RING will not preserve them.

PF_RING supports both legacy BPF filters (i.e. those supported by pcap-based applications such as tcpdump), and also two additional types of filters (named wildcard and precise filters, depending on the fact that some or all filter elements are specified) that provide developers a wide choice of options. Filters are evaluated inside the PF_RING module thus in kernel. Some modern adapters such as Intel NICs based on 82599, support hardware-based filters that are also supported by PF_RING via specified API calls (e.g. pfring_set_hw_rule). PF_RING filters (except hw filters) can have an action specified, for telling to the PF_RING kernel module what action needs to be performed when a given packet matches the filter. Actions include pass/don't pass the filter to the user space application, stop evaluating the filter chain, or reflect packet. In PF_RING, packet reflection is the ability to transmit (unmodified) the packet matching the filter onto a network interface (this except the interface on which the packet has been received). The whole reflection functionality is implemented inside the PF_RING kernel module, and the only activity requested to the user-space application is the filter specification without any further packet processing.

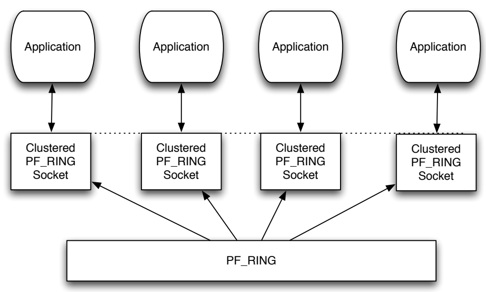
The packet journey in PF_RING is quite long



before being queued into a PF_RING ring.

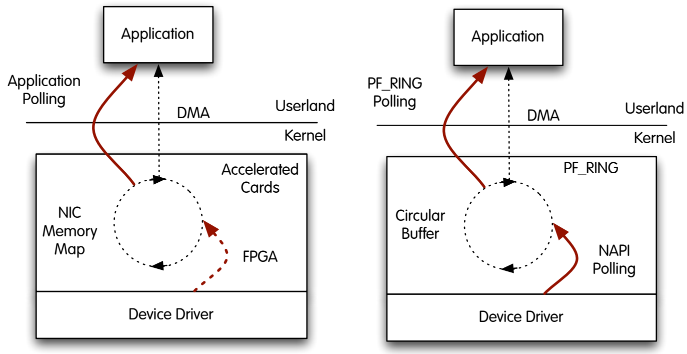


PF_RING can also increase the performance of packet capture applications by implementing two mechanisms named balancing and clustering. These mechanisms allow applications, willing to partition the set of packets to handle, to handle a portion of the whole packet stream while sending all the remaining packets to the other members of the cluster. This means that different applications opening PF_RING sockets can bind them to a specific cluster Id (via pfring_set_cluster) for joining the forces and each analyze a portion of the packets.



The way packets are partitioned across cluster sockets is specified in the cluster policy that can be either per-flow (i.e. all the packets belonging to the same tuple <proto, ip src/dst, port src/dst>) that is the default or round-robin. This means that if you select per-flow balancing, all the packets belonging to the same flow (i.e. the 5-tuple specified above) will go to the same application, whereas with round-robin all the apps will receive the same amount of packets but there is no guarantee that packets belonging to the same queue will be received by a single application. So in one hand per-flow balancing allows you to preserve the application logic as in this case the application will receive a subset of all packets but this traffic will be consistent. On the other hand if you have a specific flow that takes most of the traffic, then the application that will handle such flow will be over-flooded by packets and thus the traffic will not be heavily balanced.

As previously stated, PF_RING can work both on top of standard NIC drivers, or on top of PF_RING aware drivers for improving packet capture. In addition to these drivers, for some selected adapters, it is possible to use other driver types that further increase packet capture. The first family of drivers is named TNAPI, that allow packets to be pushed more efficiently into PF_RING by means of kernel threads activated directly by the TNAPI driver.



For those users that instead need maximum packet capture speed, it is also possible to use a different type of driver named DNA, that allows packets to be read directly from the network interface by simultaneously bypassing both the Linux kernel and the PF_RING module.

4. PF_RING Installation

When you download PF_RING you fetch the following components:

- An automatic patch mechanism allows you to automatically patch a vanilla kernel with PF_RING.
- The PF_RING user-space SDK.
- An enhanced version of the libpcap library that transparently takes advantage of PF_RING if installed, or fallback to the standard behavior if not installed.
- PF_RING aware drivers for different chips of various vendors.

PF_RING is downloaded by means of SVN as explained in http://www.ntop.org/PF_RING.html

4.1. Linux Kernel Module Installation

The PF_RING source code layout is the following:

- doc/
- README
- kernel/
- userland/
- drivers/
- Makefile

You can compile the code typing (as normal, non-root, user) `make` from the main directory. In order to compile PF_RING you need to have the linux kernel headers (or kernel source) installed.

Note that:

- the kernel installation requires super user (root) capabilities.
- For some Linux distributions a kernel installation/compilation package is provided.
- As of PF_RING 4.x you NO LONGER NEED to patch the linux kernel as in previous PF_RING versions.

5. Running PF_RING

Before using any PF_RING application pf_ring kernel module should be loaded (as superuser):

```
#insmod  PATH_TO_MODULE/pf_ring.ko  [transparent_mode=0|1|2][  min_num_slots=x]
[ enable_tx_capture=1|0][ enable_ip_defrag=1|0]
```

Note:

- transparent_mode=0 (default)
  Packets are received via the standard Linux interface. Any driver can use this mode.
- transparent_mode=1 (Both vanilla and PF_RING-aware drivers)
  Packets are memcpy() to PF_RING and also to the standard Linux path.
- transparent_mode=2 (PF_RING -aware drivers only)
  Packets are ONLY memcpy() to PF_RING and not to the standard Linux path (i.e. tcpdump won't see anything).

The higher is the transparent_mode value, the faster it gets packet capture.

Other parameters:

- min_num_slots
  Min number of ring slots (default – 4096).
- enable_tx_capture
  Set to 1 to capture outgoing packets, set to 0 to disable capture outgoing packets (default – RX+TX).
- enable_ip_defrag
  Set to 1 to enable IP defragmentation, only rx traffic is defragmented.

5.1. Checking PF_RING Device Configuration

When PF_RING is activated, a new entry /proc/net/pf_ring is created.

```
nbox-factory:/home/deri# ls /proc/net/pf_ring/
info  plugins_info
nbox-factory:/home/deri# cd /proc/net/pf_ring/
nbox-factory:/proc/net/pf_ring# cat info
Version              : 3.7.5
Bucket length        : 2000 bytes
Ring slots           : 4096
Slot version         : 9
Capture TX           : Yes [RX+TX]
IP Defragment        : No
Transparent mode     : Yes
Total rings          : 0
Total plugins        : 2
nbox-factory:/proc/net/pf_ring# cat plugins_info
ID    Plugin
2     sip [SIP protocol analyzer]
12    rtp [RTP protocol analyzer]
```

PF_RING allows users to install plugins for handling custom traffic. Those plugins are also registered in the pf_ring /proc tree and can be listed by typing the plugins_info file.

5.2. Libpfring and Libpcap Installation

Both libpfring (userspace PF_RING library) and libpcap are distributed in source format. They can be compiled as follows:

- cd userland/libpfring
- make
- sudo make install
- cd ../libpcap-1.1.1-ring/
- ./configure
- make

Note that the lib is reentrant hence it's necessary to link you PF_RING-enabled applications also against the -lpthread library.

IMPORTANT
Legacy pcap-based applications need to be recompiled against the new libpcap and linked with a PF_RING enabled libpcap.a in order to take advantage of PF_RING. Do not expect to use PF_RING without recompiling your existing application.

6. PF_RING for Application Developers

Conceptually PF_RING is a simple yet powerful technology that enables developers to create high-speed traffic monitor and manipulation applications in a small amount of time. This is because PF_RING shields the developer from inner kernel details that are handled by a library and kernel driver. This way developers can dramatically save development time focusing on their application they are developing without paying attention to the way packets are sent and received.

This chapter covers:

- The PF_RING API.
- Extensions to the libpcap library for supporting legacy applications.
- How to patch the Linux kernel for enabling PF_RING

6.1. The PF_RING API

The PF_RING internal data structures should be hidden to the user who can manipulate packets and devices only by means of the available API defined in the include file pfring.h that comes with PF_RING.

6.2. Return Codes

By convention, the library returns negative values for errors and exceptions. Non-negative codes indicate success. In case return code have another meaning, then they are described inside the corresponding function.

6.3. PF_RING Device Name Convention

In PF_RING device names are the same as libpcap and ifconfig. So eth0 and eth5 are valid names you can use in PF_RING. You can specify also a virtual device named 'any' that instructs PF_RING to capture packets from all available network devices.

As previously explained, with PF_RING you can use both the drivers that come with your Linux distribution (thus that are not PF_RING-specific), or some PF_RING-aware drivers (you can find them into the `drivers/` directory of PF_RING) that push PF_RING packets much more efficiently that vanilla drivers. If you own a modern multi-queue NIC running with a PF_RING-aware driver (e.g. the Intel 10 Gbit adapter), PF_RING allows you to capture packet from the whole device (i.e. capture packets regardless of the RX queue on which the packet has been received, ethX for instance) or from a specific queue (e.g. ethX@Y). Supposing to have an adapter with Z queue, the queue Id Y, must be in range 0..Z-1. In case you specify a queue that does not exist, no packets will be captured.

6.4.PF_RING: SCOKET Initialization

pfring* pfring_open(char *device_name, u_int8_t promisc, u_int32_t caplen, u_int8_t reentrant)

This call is used to initialize an PF_RING socket hence obtain a handle of type struct pfring that can be used in subsequent calls. Note that:

- You can use both physical (e.g. eth0), virtual (e.g. tap devices), and also RX-queues (e.g. ethX@0).
- You need super-user capabilities in order to open a device.
- If you want to open a device with driver in DNA mode, you must call pfring_open_dna() instead.

Input parameters:

device_name
Symbolic name of the PF_RING-aware device we're attempting to open (e.g. eth0).

promisc
If set to a value different than zero, the device is open in promiscuous mode.

caplen
Maximum packet capture len (also known as snaplen).

reentrant
If set to a value different than zero, the device is open in reentrant mode. This is implemented by means of semaphores and it results is slightly worse performance. Use reentrant mode only for multithreaded applications.

Return value:

On success a handle is returned, NULL otherwise.

u_int8_t pfring_open_multichannel(char *device_name, u_int8_t promisc, u_int32_t caplen, u_int8_t _reentrant, pfring* ring[MAX_NUM_RX_CHANNELS])

This call is similar to pfring_open() with the exception that in case of a multi RX-queue NIC, instead of opening a single ring for the whole device, several individual rings are open (one per RX-queue)

Input parameters:

device_name
Symbolic name of the PF_RING-aware device we're attempting to open (e.g. eth0). No queue name hash to be specified, but just the main device name

promisc
If set to a value different than zero, the device is open in promiscuous mode.

caplen

Maximum packet capture len (also known as snaplen).

reentrant
If set to a value different than zero, the device is open in reentrant mode. This is implemented by means of semaphores and it results is slightly worse performance. Use reentrant mode only for multithreaded applications.

ring
A pointer to an array of rings that will contain the opened ring pointers.

Return value:
The last index of the ring array that contain a valid ring pointer.

pfring* pfring_open_dna(char *device_name, u_int8_t promisc, u_int8_t _reentrant)

This call is similar to pfring_open() but only for devices with a DNA driver. If you want to open a ring on top of non-DNA device please open pfring_open() instead. Please note that in case of DNA devices, the PF_RING kernel module must be loaded before the the DNA driver.

Input parameters:
device_name
Symbolic name of the PF_RING-aware device we're attempting to open (e.g. eth0).

promisc
If set to a value different than zero, the device is open in promiscuous mode.

reentrant
If set to a value different than zero, the device is open in reentrant mode. This is implemented by means of semaphores and it results is slightly worse performance. Use reentrant mode only for multithreaded applications.

Return value:
On success a handle is returned, NULL otherwise.

6.5. PF_RING: Device Termination

void pfring_close(pfring *ring)

This call is used to terminate an PF_RING device previously open. Note that you must always close a device before leaving an application. If unsure, you can close a device from a signal handler.

Input parameters:
ring
The PF_RING handle that we are attempting to close.

6.6. PF_RING: Read an Incoming Packet

int pfring_recv(pfring *ring, char* buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_char wait_for_incoming_packet)

This call returns an incoming packet when available.

Input parameters:

ring
The PF_RING handle where we perform the check.

buffer
A memory area allocated by the caller where the incoming packet will be stored.

buffer_len
The length of the memory area above. Note that the incoming packet is cut if the incoming packet is too long for the allocated area.

hdr
A memory area where the packet header will be copied.

wait_for_incoming_packet
If 0 we simply check the packet availability, otherwise the call is blocked until a packet is available.

Return value:

0 in case of no packet being received, 1 in case of success, -1 in case of error.

int pfring_read(pfring *ring, char* buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet)

This function is an alias for pfring_recv() and it has been preserved for compatibility reason with old applications.

6.7. PF_RING: Ring Clusters

int pfring_set_cluster(pfring *ring, u_int clusterId)

This call allows a ring to be added to a cluster that can spawn across address spaces. On a nuthsell when two or more sockets are clustered they share incoming packets that are balanced on a per-flow manner. This technique is useful for exploiting multicore systems of for sharing packets in the same address space across multiple threads.

Input parameters:

ring
The PF_RING handle to be cluster.

clusterId
A numeric identifier of the cluster to which the ring will be bound.

Return value:

Zero if success, a negative value otherwise.

int pfring_remove_from_cluster(pfring *ring);

This call allows a ring to be removed from a previous joined cluster.

Input parameters:

ring
The PF_RING handle to be cluster.

clusterId
A numeric identifier of the cluster to which the ring will be bound.

Return value:

Zero if success, a negative value otherwise.

6.8. PF_RING: Packet Reflection

You can specify packet reflection inside the filtering rules.

```
typedef struct {
  ...
  char reflector_device_name[REFLECTOR_NAME_LEN];
  ...
} filtering_rule;
```

In the `reflector_device_name` you need to specify a device name (e.g. eth0) on which packets matching the filter will be reflected. Make sure NOT to specify as reflection device the same device name on which you capture packets, as otherwise you will create a packet loop.

6.9. PF_RING: Packet Sampling

int pfring_set_sampling_rate(pfring *ring, u_int32_t rate)

Implement packet sampling directly into the kernel. Note that this solution is much more efficient than implementing it in user-space. Sampled packets are only those that pass all filters (if any)

Input parameters:

ring
The PF_RING handle on which sampling is applied.

rate
The sampling rate. Rate of X means that 1 packet out of X is forwarded. This means that a sampling rate of 1 disables sampling

Return value:

Zero if success, a negative value otherwise.

6.10. PF_RING: Packet Filtering

PF_RING allows filtering packets in two ways: precise (a.k.a. hash filtering) or wildcard filtering. Precise filtering is used when it is necessary to track a precise 6-tuple connection <vlan Id, protocol, source IP, source port, destination IP, destination port>. Wildcard filtering is used instead whenever a filter can have wildcards on some of its fields (e.g. match all UDP packets regardless of their destination). If some field is set to zero it will not participate in filter calculation

6.11. PF_RING: Wildcard Filtering

int pfring_add_filtering_rule(pfring *ring, filtering_rule* rule_to_add)

Add a filtering rule to an existing ring. Each rule will have a unique rule Id across the ring (i.e. two rings can have rules with the same id).

Input parameters:

ring
The PF_RING handle on which the rule will be added.

rule_to_add
The rule to add as defined in the last chapter of this document.

Return value:

Zero if success, a negative value otherwise.

int pfring_remove_filtering_rule(pfring *ring, u_int16_t rule_id)

Remove a previously added filtering rule.

Input parameters:

ring
The PF_RING handle on which the rule will be added.

rule_id
The id of a previously added rule that will be removed.

Return value:

Zero if success, a negative value otherwise (e.g. the rule does not exist).

int pfring_get_filtering_rule_stats(pfring *ring, u_int16_t rule_id, char* stats, u_int *stats_len)

Read statistics of a hash filtering rule.

Input parameters:

ring
The PF_RING handle from which stats will be read.

rule_id
The rule id that identifies the rule for which stats are read.

stats
A buffer allocated by the user that will contain the rule statistics. Please make sure that the buffer is large enough to contain the statistics. Such buffer will contain number of received and dropped packets.

stats_len
The size (in bytes) of the stats buffer.

Return value:

Zero if success, a negative value otherwise (e.g. the rule does not exist).

6.12. PF_RING: Hash Filtering

int pfring_handle_hash_filtering_rule(pfring *ring, hash_filtering_rule* rule_to_add, u_char add_rule)

Add or remove a hash filtering rule.

Input parameters:

ring
The PF_RING handle from which stats will be read.

rule_to_add
The rule that will be added/removed as defined in the last chapter of this document. All rule parameters should be defined in the filtering rule (no wildcards).

add_rule
If set to a positive value the rule is added, if zero the rule is removed.

Return value:

Zero if success, a negative value otherwise (e.g. the rule to be removed does not exist).

All rule parameters should be defined in the filtering rule (no wildcards).

int pfring_get_hash_filtering_rule_stats(pfring *ring, hash_filtering_rule* rule, char* stats, u_int *stats_len)

Read statistics of a hash filtering rule.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

rule
The rule for which stats are read. This needs to be the same rule that has been previously added.

stats
A buffer allocated by the user that will contain the rule statistics. Please make sure that the buffer is large enough to contain the statistics. Such buffer will contain number of received and dropped packets.

stats_len
The size (in bytes) of the stats buffer.

Return value:

Zero if success, a negative value otherwise (e.g. the rule to be removed does not exist).

6.13. PF_RING: In-NIC Packet Filtering

Some multi-queue modern network adapters feature "packet steering" capabilities. Using them it is possible to instruct the hardware NIC to assign selected packets to a specific RX queue. If the specified queue has an Id that exceeds the maximum queueId, such packet is discarded thus acting as a hardware firewall filter.

int pfring_set_hw_rule(pfring *ring, hw_filtering_rule *rule, u_int8_t add_rule)

Sets a specified filtering rule into the NIC. Note that no PF_RING filter is added, but only a NIC filter.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

rule
The filtering rule to be set in the NIC as defined in the last chapter of this document. All rule parameters should be defined, and if set to zero they do not participate to filtering.

add_rule
Set it to 0 to remove the specified filtering rule, 1 to add the rule

Return value:

Zero if success, a negative value otherwise (e.g. the rule to be added/removed has wrong format or if the NIC to which this ring is bound does not support hardware filters).

6.14. PF_RING: Filtering Policy

int pfring_toggle_filtering_policy(pfring *ring, u_int8_t rules_default_accept_policy)

Set the default filtering policy. This means that if no rule is matching the incoming packet the default policy will decide if the packet is forwarded to user space of dropped. Note that filtering rules are limited to a ring, so each ring can have a different set of rules and default policy.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

rules_default_accept_policy
If set to a positive value the default policy is accept (i.e. forward packets to user space), drop otherwise.

Return value:

Zero if success, a negative value otherwise.

6.15. PF_RING: Send Packets

int pfring_send(pfring *ring, char *pkt, u_int pkt_len)

Although PF_RING has been optimized for RX, it is also possible to send packets (TX). This function allows to send a raw packet (i.e. it is sent on wire as specified). This packet must be fully specified (the the MAC address up) and it will be transmitted as-is without any further manipulation. Note that it is much more efficient to send packets from inside the kernel rather than from the user space.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

pkt
The buffer containing the packet to send.

pkt_len
The length of the pkt buffer.

Return value:

The number of bytes sent if success, a negative value otherwise.

6.16. PF_RING: Miscellaneous Functions

int pfring_enable_ring(pfring *ring)

When a ring is created, it is not enabled (i.e. incoming packets are dropped) until the above function is called.

Input parameters:

ring
The PF_RING handle to enable.

Return value:

Zero if success, a negative value otherwise (e.g. the ring cannot be enabled).

int pfring_stats(pfring *ring, pfring_stat *stats)

Read ring statistics (packets received and dropped).

Input parameters:

ring
The PF_RING handle to enable.

stats
A user-allocated buffer on which stats (number of received and dropped packets) will be stored.

Return value:

Zero if success, a negative value otherwise.

int pfring_version(pfring *ring, u_int32_t *version)

Read the ring version. Note that is the ring version is 3.7 the retuned ring version is 0x030700.

Input parameters:

ring
The PF_RING handle to enable.

version
A user-allocated buffer on which ring version will be copied.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_direction(pfring *ring, packet_direction direction)

Tells PF_RING to consider only those packets matching the specified direction. If the application does not call this function, all the packets (regardless of the direction, either RX or TX) are returned.

Input parameters:

ring
The PF_RING handle to enable.

direction
The packet direction (RX, TX or both RX and TX).

Return value:

Zero if success, a negative value otherwise.

int pfring_set_application_name(pfring *ring, char *name)

Tells PF_RING the name of the application (usually argv[0]) that uses this ring. This information is used to identify the application when accessing the files present in the PF_RING /proc filesystem. Example

```
> cat /proc/net/pf_ring/16614-eth0.0
Bound Device   : eth0
Slot Version   : 12 [4.5.1]
Active         : 1
Sampling Rate  : 1
Appl. Name     : pfcount
IP Defragment  : No
....
```

Input parameters:

ring
The PF_RING handle to enable.

name
The name of the application using this ring.

Return value:

Zero if success, a negative value otherwise.

u_int8_t pfring_get_num_rx_channels(pfring *ring)

Returns the number of RX channels (also known as RX queues) of the ethernet interface to which this ring is bound.

Input parameters:

ring
The PF_RING handle to query.

Return value:

The number of RX channels, or 1 (default) in case this in information is unknown.

int pfring_get_selectable_fd(pfring *ring)

Returns the file descriptor associated to the specified ring. This number can be used in function calls such as poll() and select() for passively waiting for incoming packets.

Input parameters:

ring
The PF_RING handle to query.

Return value:

A number that can be used as reference to this ring, in function calls that require a selectable file descriptor.

6.17. The C++ PF_RING interface

The C++ interface (see. PF_RING/userland/libpfring/c++/) is equivalent to the C interface. No major changes have been made and all the methods have the same name as C. For instance:

- C: int pfring_stats(pfring *ring, pfring_stat *stats);
- C++: inline int get_stats(pfring_stat *stats);

7. Writing PF_RING Plugins

Since version 3.7, developers can write plugins in order to delegate to PF_RING activities like:

- Packet payload parsing
- Packet content filtering
- In-kernel traffic statistics computation.

In order to clarify the concept, imagine that you need to develop an application for VoIP traffic monitoring. In this case it's necessary to:

- parse signaling packets (e.g. SIP or IAX) so that those that only packets belonging to interesting peers are forwarded.
- compute voice statistics into PF_RING and report to user space only the statistics, not the packets.

In this case a developer can code two plugins so that PF_RING can be used as an advanced traffic filter and a way to speed-up packet processing by avoiding packets to cross the kernel boundaries when not needed.

The rest of the chapter explains how to implement a plugin and how to call it from user space.

7.1. Implementing a PF_RING Plugin

Inside the directory kernel/net/ring/plugins/ there is a simple plugin called dummy_plugin that shows how to implement a simple plugin. Let's explore the code.

Each plugin is implemented as a Linux kernel module. Each module must have two entry points, module_init and module_exit, that are called when the module is insert and removed. The module_init function, in the dummy_plugin example it's implement by the function dummy_plugin_init(), is responsible for registering the plugin by calling the do_register_pfring_plugin() function. The parameter passed to the registration function is a data structure of type 'struct pfring_plugin_registration' that contains:

- a unique integer pluginId.
- pfring_plugin_handle_skb
  A pointer to a function called whenever an incoming packet is received.
- pfring_plugin_filter_skb
  A pointer to a function called whenever a packet needs to be filtered. This function is called after pfring_plugin_handle_skb().
- pfring_plugin_get_stats
  A pointer to a function called whenever a user wants to read statistics from a filtering rule that has set this plugin as action.
- pfring_plugin_free_ring_mem
  A pointer to a function called when the plugin is unregistered (rmmod). Free here any memory allocated by the plugin during its operations.
- pfring_plugin_add_rule
  A pointer to a function called when a user has set for this plugin a filtering rule with behavior forward_packet_add_rule_and_stop_rule_evaluation. In case of a packet match, this function is called.

A developer can choose not to implement all the above functions, but in this case the plugin will be limited in functionality (e.g. if pfring_plugin_filter_skb is set to NULL filtering is not supported).

7.2. PF_RING Plugin: Handle Incoming Packets

```
static int plugin_handle_skb(filtering_rule_element *rule,
                             filtering_hash_bucket *hash_rule,
                             struct pcap_pkthdr *hdr,
                             struct sk_buff *skb,
                             u_int16_t filter_plugin_id,
                            struct parse_buffer *filter_rule_memory_storage)
```

This function is called whenever an incoming packet (RX or TX) is received. This function typically updates rule statistics. Note that if the developer has set this plugin as filter plugin, then the packet has:

- already been parsed
- passed a rule payload filter (if set).

Input parameters:

rule
A pointer to a wildcard rule (if this plugin has been set on a wildcard rule) or NULL (if this plugin has been set to a hash rule).

hash_rule
A pointer to a hash rule (if this plugin has been set on a hash rule) or NULL (if this plugin has been set to a wildcard rule). Note if rule is NULL, hash_rule is not, and vice-versa.

hdr
A pointer to a pcap packet header for the received packet. Please note that:

- the packet is already parsed
- the header is an extended pcap header containing parsed packet header metadata.

skb
A sk_buff datastructure used in Linux to carry packets inside the kernel.

filter_plugin_id
The id of the plugin that has parsed packet payload (not header that is already stored into hdr). if the filter_plugin_id is the same as the id of the dummy_plugin then this packet has already been parsed by this plugin and the parameter filter_rule_memory_storage points to the payload parsed memory.

filter_rule_memory_storage
Pointer to a data structure containing parsed packet payload information that has been parsed by the plugin identified by the parameter filter_plugin_id. Note that:

- only one plugin can parse a packet.
- the parsed memory is allocated dynamically (i.e. via kmalloc) by plugin_filter_skb and freed by the PF_RING core.

Return value:

Zero if success, a negative value otherwise.

7.3. PF_RING Plugin: Filter Incoming Packets

```
int plugin_filter_skb(filtering_rule_element *rule,
                      struct pcap_pkthdr *hdr,
                      struct sk_buff *skb,
                      struct parse_buffer **parse_memory)
```

This function is called whenever a previously parsed packet (via plugin_handle_skb) incoming packet (RX or TX) needs to be filtered. In this case the packet is parsed, parsed information is returned and the return value indicates whether the packet has passed the filter.

Input parameters:

rule
A pointer to a wildcard rule that contains a payload filter to apply to the packet.

hdr
A pointer to a pcap packet header for the received packet. Please note that:

- the packet is already parsed
- the header is an extended pcap header containing parsed packet header metadata.

skb
A sk_buff data structure used in Linux to carry packets inside the kernel.

Output parameters:

parse_memory
A pointer to a memory area allocated by the function, that will contain information about the parsed packet payload.

Return value:

Zero if the packet has not matched the rule filter, a positive value otherwise.

7.4. PF_RING Plugin: Read Packet Statistics

```
int plugin_plugin_get_stats(filtering_rule_element *rule,
                            filtering_hash_bucket  *hash_bucket,
                            u_char* stats_buffer,
                            u_int stats_buffer_len)
```

This function is called whenever a user space application wants to read statics about a filtering rule.

Input parameters:

rule
A pointer to a wildcard rule (if this plugin has been set on a wildcard rule) or NULL (if this plugin has been set to a hash rule).

hash_rule
A pointer to a hash rule (if this plugin has been set on a hash rule) or NULL (if this plugin has been set to a wildcard rule). Note if rule is NULL, hash_rule is not, and vice-versa.

stats_buffer
A pointer to a buffer where statistics will be copied..

stats_buffer_len
Length in bytes of the stats_buffer.

Return value:

The length of the rule stats, or zero in case of error.

7.5. Using a PF_RING Plugin

A PF_RING based application, can take advantage of plugins when filtering rules are set. The filtering_rule data structure is used to both set a rule and specify a plugin associated to it.

```
filtering_rule rule;

rule.rule_id = X;
....
rule.plugin_action.plugin_id = MY_PLUGIN_ID;
```

When the plugin_action.plugin_id is set, whenever a packet matches the header portion of the rule, then the MY_PLUGIN_ID plugin (if registered) is called and the plugin_filter_skb () and plugin_handle_skb() are called.

If the developer is willing to filter a packet before plugin_handle_skb() is called, then extra filtering_rule fields need to be set. For instance suppose to implement a SIP filter plugin and to instrument it so that only the packets with INVITE are returned. The following lines of code show how to do this.

```
struct sip_filter *filter = (struct sip_filter*)rule.extended_fields.filter_plugin_data;

rule.extended_fields.filter_plugin_id = SIP_PLUGIN_ID;
filter->method = method_invite;
filter->caller[0]  = '\0'; /* Any caller */
filter->called[0]  = '\0'; /* Any called */
filter->call_id[0] = '\0'; /* Any call-id */
```

As explained before, the pfring_add_filtering_rule() function is used to register filtering rules.

8. PF_RING Data Structures

Below are described some relevant PF_RING data structures.

```
 typedef struct {
  u_int16_t rule_id;                 /* Rules are processed in order from
                                        lowest to higest id */
  rule_action_behaviour rule_action; /* What to do in case of match */
  u_int8_t balance_id, balance_pool; /* If balance_pool > 0, then pass the
                                        packet above only if the
                                        (hash(proto, sip, sport, dip, dport) %
                                        balance_pool) = balance_id */
  filtering_rule_core_fields     core_fields;
  filtering_rule_extended_fields extended_fields;
  filtering_rule_plugin_action   plugin_action;
  char reflector_device_name[REFLECTOR_NAME_LEN];

  filtering_internals internals;   /* PF_RING internal fields */
} filtering_rule;


 typedef struct {
  u_int8_t dmac[ETH_ALEN], smac[ETH_ALEN]; /* Use '0' (zero-ed MAC address) for
                                              any MAC address. This is applied
                                              to both source and dst. */
  u_int16_t vlan_id;                 /* Use '0' for any vlan */
  u_int8_t  proto;                   /* Use 0 for 'any' protocol */
  ip_addr   host_low, host_high;     /* User '0' for any host. This is applied
                                        to both source and destination. */
  u_int16_t port_low, port_high;     /* All ports between port_low...port_high
                                        0 means 'any' port. This is applied to
                                        both source and destination. This means
                                        that (proto, sip, sport, dip, dport)
                                        matches the rule if one in "sip &
                                        sport", "sip & dport" "dip & sport"
                                        match. */
} filtering_rule_core_fields;


typedef struct {
  char payload_pattern[32];         /* If strlen(payload_pattern) > 0, the
                                       packet payload must match the specified
                                        pattern */
  u_int16_t filter_plugin_id;       /* If > 0 identifies a plugin to which the
                                       datastructure below will be passed for
                                        matching */
  char      filter_plugin_data[FILTER_PLUGIN_DATA_LEN];
           /* Opaque datastructure that is interpreted by the
              specified plugin and that specifies a filtering
             criteria to be checked for match. Usually this data
             is re-casted to a more meaningful datastructure
           */
} filtering_rule_extended_fields;
```

```
typedef enum {
  forward_packet_and_stop_rule_evaluation = 0,
  dont_forward_packet_and_stop_rule_evaluation,
  execute_action_and_continue_rule_evaluation,
  forward_packet_add_rule_and_stop_rule_evaluation,
  reflect_packet_and_stop_rule_evaluation,
  reflect_packet_and_continue_rule_evaluation,
  bounce_packet_and_stop_rule_evaluation,
  bounce_packet_and_continue_rule_evaluation
} rule_action_behaviour;


typedef struct {
  u_int16_t rule_id; /* Future use */
  u_int16_t vlan_id;
  u_int8_t  proto;
  ip_addr host_peer_a, host_peer_b;
  u_int16_t port_peer_a, port_peer_b;
  rule_action_behaviour rule_action; /* What to do in case of match */
  filtering_rule_plugin_action plugin_action;
  char reflector_device_name[REFLECTOR_NAME_LEN];
  filtering_internals internals;   /* PF_RING internal fields */
} hash_filtering_rule;


typedef enum {
  forward_packet_and_stop_rule_evaluation = 0,
  dont_forward_packet_and_stop_rule_evaluation,
  execute_action_and_continue_rule_evaluation,
  forward_packet_add_rule_and_stop_rule_evaluation,
  reflect_packet_and_stop_rule_evaluation,
  reflect_packet_and_continue_rule_evaluation,
  bounce_packet_and_stop_rule_evaluation,
  bounce_packet_and_continue_rule_evaluation
} rule_action_behaviour;

typedef struct {
  u_int64_t recv, drop;
} pfring_stat;
```