PF_RING User Guide

Linux High Speed Packet Capture

Version 5.4.6
Aug 2012

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 5.4.6 (Aug 2012)
  - Updated guide to PF_RING version 5.4.6

- Release 5.4.0 (May 2012)
  - Updated guide to PF_RING version 5.4.0
  - New libzero for zero-copy flexible packet processing on top of DNA.

- Release 5.3.1 (March 2012)
  - Updated guide to PF_RING version 5.3.1

- Release 5.2.1 (January 2012)
  - Updated guide to PF_RING version 5.2.1
  - New API functions for managing hardware clocks and timestamps.
  - New kernel plugin callbacks.

- Release 5.1 (September 2011)
  - Updated guide to PF_RING version 5.1.0

- Release 4.7.1 (July 2011)
  - Updated guide to PF_RING version 4.7.1
  - Described PF_RING modular library and some modules (DAG, DNA)

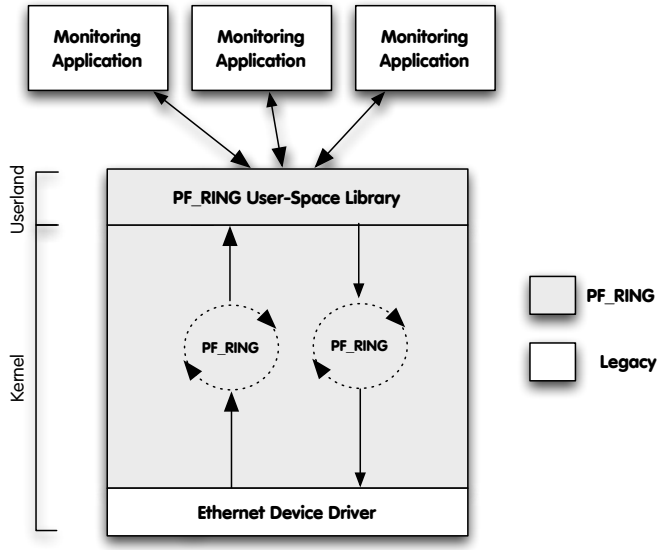
- Release 4.6.1 (March 2011)
  - Updated guide to PF_RING version 4.6.1

- Release 4.6 (February 2011)
  - Updated guide to PF_RING version 4.6.0.

- 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 module that provides low-level packet copying into the PF_RING 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.

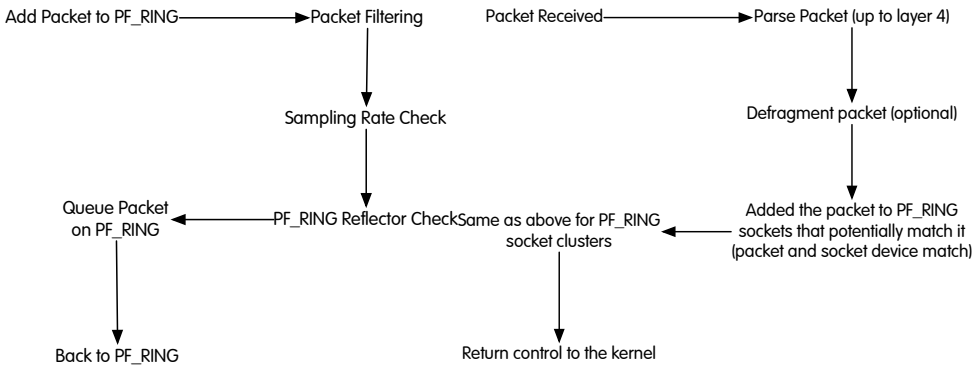
3.1.Packet Filtering

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 82599-

based or Silicom Redirector NICs, support hardware-based filters that are also supported by PF_RING via specified API calls (e.g. pfring_add_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.

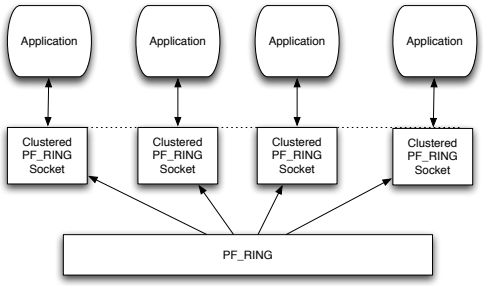
3.2.Packet Journey

The packet journey in PF_RING is quite long before being queued into a PF_RING ring.



3.3.Packet Clustering

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.

4.PF_RING Driver Families

As previously stated, PF_RING can work both on top of standard NIC drivers, or on top of specialized drivers. The PF_RING kernel module is the same, but based on the drivers being used some functionality and performance are different.

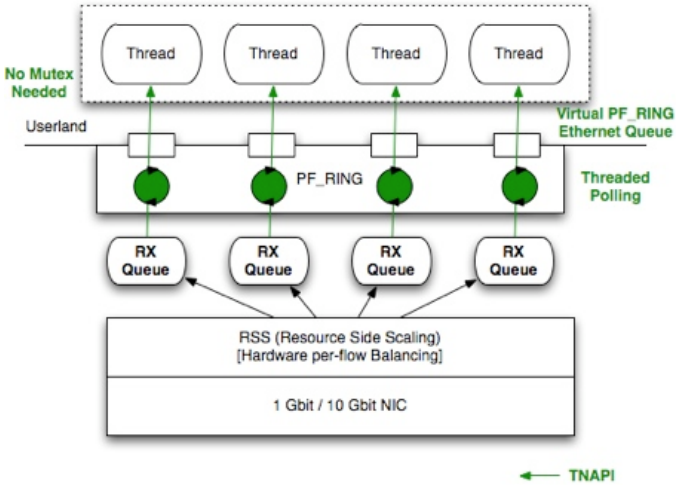
4.1.PF_RING-aware Drivers

These drivers (available on PF_RING/driver/PF_RING-aware) are designed to improve packet capture by pushing packets directly to PF_RING without going through the standard Linux packet dispatching mechanisms. With these drivers you can use the transparent_mode with values 1, or 2 (see below on this document for details.

In addition to PF_RING aware drivers, for some selected adapters, it is possible to use other driver types that further increase packet capture.

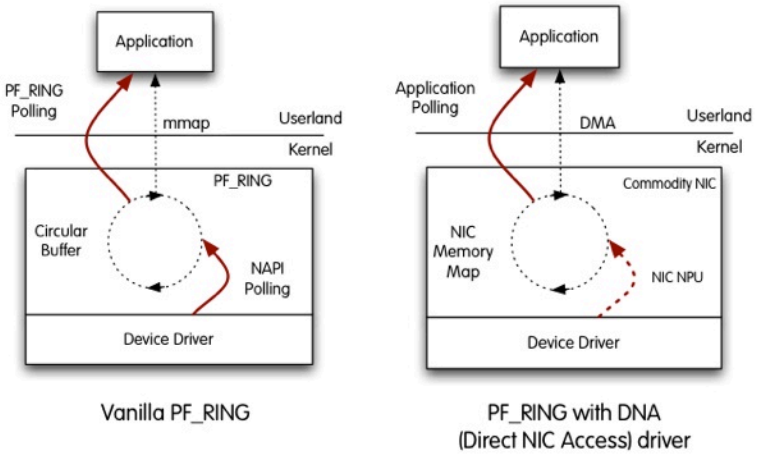
4.2.TNAPI

The first family of drivers is named TNAPI (Threaded NAPI), that allow packets to be pushed more efficiently into PF_RING by means of kernel threads activated directly by the TNAPI driver. The TNAPI drivers are designed for improving packet capture, and thus they cannot be used to transmit packets as the TX path is disabled.



4.3.DNA

For those users that who need maximum packet capture speed with 0% CPU utilization for copying packets to the host (i.e. the NAPI polling mechanism is not used), 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 in a zero-copy fashion.



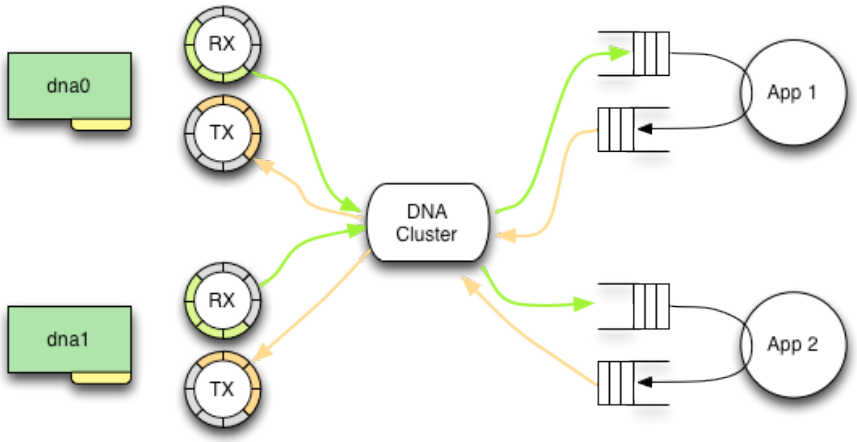
In DNA both RX and TX operations are supported. As the kernel is bypassed, some PF_RING functionality are missing, and they include:

- In kernel packet filtering (BPF and PF_RING filters)
- PF_RING kernel plugins have no effect.

5.Libzero for DNA

As most applications need complex packet processing features, starting with PF_RING 5.4.0 a library named libzero has been introduced, sitting on top of the low-level DNA interface and implementing zero-copy packet processing. The libzero provides two main components: the DNA Cluster and the DNA Bouncer.

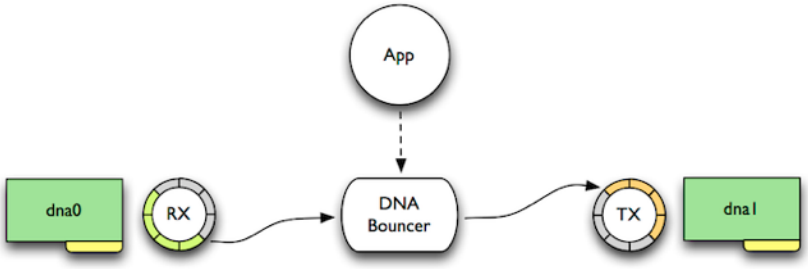
5.1.DNA Cluster



The DNA Cluster implements packet clustering, so that all applications belonging to the same cluster can share incoming packets using a flexible balancing function and transmit packets all in zero-copy. In essence is a custom implementation of RSS, that allows to distribute packets across queues inside network adapters. The cluster allow used to define their dispatching function for filtering, distributing and duplicating packets towards multiple threads and applications.

5.2.DNA Bouncer

The DNA Bouncer switches packets across two interfaces in zero-copy, leaving the user the ability to specify a function that can decide, packet-by-packet, whether a given packet has to be forwarded or not.



Forwarding in mono-directional, thus in case you want to implement bridging, two bouncers (one per direction) need to be instantiated.

6. PF_RING Installation

When you download PF_RING you fetch the following components:

- 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.
- The PF_RING kernel module.
- 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/get-started/download/.

The PF_RING source code layout is the following:

- doc/
- drivers/
- kernel/
- Makefile
- README
- README.DNA
- README.FIRST
- userland/
- vPF_RING/

You can compile the entire tree typing make (as normal, non-root, user) from the main directory.

6.1. Linux Kernel Module Installation

In order to compile the PF_RING kernel module you need to have the linux kernel headers (or kernel source) installed.

```
$ cd <PF_RING PATH>/kernel
$ make
```

Note that:

- the kernel module installation (via make install) 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.

7. Running PF_RING

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

insmod <PF_RING PATH>/kernel/pf_ring.ko [transparent_mode=0|1|2] [min_num_slots=x][enable_tx_capture=1|0] [enable_ip_defrag=1|0] [quick_mode=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).

> IMPORTANT: Do NOT use transparent_mode 1 and 2 with vanilla drivers as it will result in no packet capture.

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.
- quick_mode
  Set to 1 to run at full speed but with up to one socket per interface.

7.1. Checking PF_RING Device Configuration

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

```
# ls /proc/net/pf_ring/
dev  info  plugins_info

# cat /proc/net/pf_ring/info
Version             : 5.4.0
Ring slots          : 4096
Slot version        : 13
Capture TX          : Yes [RX+TX]
IP Defragment       : No
Socket Mode         : Standard
Transparent mode    : Yes (mode 0)
Total rings         : 0
Total plugins       : 2
```

```
# cat /proc/net/pf_ring/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.
```

7.2. Libpfring and Libpcap Installation

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

```
$ cd <PF_RING PATH>/userland/lib
$ ./configure
$ make
$ sudo make install
$ cd ../libpcap
$ ./configure
$ make
```

Note that the lib is reentrant hence it's necessary to link your 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.

7.3. Application Examples

If you are new to PF_RING, you can start with some examples. The userland/examples folder is rich of ready-to-use PF_RING applications:

```
$ cd <PF_RING PATH>/userland/examples
$ ls *.c
alldevs.c               pfcount_82599.c            pffilter_test.c
dummy_plugin_pfcount.c  pfcount_aggregator.c       pfhyperengine.c
interval.c              pfcount_bundle.c           pfmap.c
pcap2nspcap.c           pfcount_dummy_plugin.c     pfsend.c
pcount.c                pfcount_multichannel.c     pfsystest.c
pfbounce.c              pfdnabounce.c              preflect.c
pfbridge.c              pfdnacluster_master.c      pwrite.c
pfcount.c               pfdnacluster_multithread.c
$ make
```

For instance, pfcount allows you to receive packets printing some statistics:

```
# ./pfcount -i dna0
Using PF_RING v.5.4.0
...
=========================
Absolute Stats: [64415543 pkts rcvd][0 pkts dropped]
Total Pkts=64415543/Dropped=0.0 %
64'415'543 pkts - 5'410'905'612 bytes [4'293'748.94 pkt/sec - 2'885.39
Mbit/sec]
=========================
Actual Stats: 14214472 pkts [1'000.03 ms][14'214'017.15 pps/9.55 Gbps]
=========================
```

Another example is pfsend, which allows you to send packets (synthetic packets, or optionally a .pcap file can be used) at a specific rate:

```
# ./pfsend -f 64byte_packets.pcap -n 0 -i dna0 -r 5
...
TX rate: [current 7'508'239.00 pps/5.05 Gbps][average 7'508'239.00 pps/
5.05 Gbps][total 7'508'239.00 pkts]
```

7.4. PF_RING Additional Modules

As of version 4.7, the PF_RING library has a new modular architecture, making it possible to use additional components other than the standard PF_RING kernel module. These components are compiled inside the library according to the supports detected by the configure script.
Currently, the set of additional modules includes:

- DAG module.
  This module adds native support for Endace DAG cards in PF_RING. In order to use this module it's necessary to have the dag library (4.x or later) installed and to link your PF_RING-enabled application using the -ldag flag.

- DNA module.
  This module can be used to open a device in DNA mode, if you own a supported card and a DNA driver. Please note that the PF_RING kernel module must be loaded before the the DNA driver. With DNA you can dramatically increase the packet capture and transmission speed as the kernel layer is bypassed and applications can communicate directly with drivers.
  Currently these DNA-aware drivers are available:
  - e1000e
  - igb
  - ixgbe

  The drivers are part of the PF_RING distribution and can be found in drivers/DNA/.
  With all the drivers you can achieve wire rate at any packet size, both for RX and TX. You can test RX using the pfcount application, and TX using the pfsend application.

  Note that in case of TX, the transmission speed is limited by the RX performance. This is because when the receiver cannot keep-up with the capture speed, the ethernet NIC sends ethernet PAUSE

frames back to the sender to slow it down. If you want to ignore these frames and thus send at full speed, you need to disable autonegotiation and ignore them (ethtool -A dnaX autoneg off rx off tx off).

- Link Aggregation ("multi") module.
  This module can be used to aggregate multiple interfaces in order to capture packets from all of them opening a single PF_RING socket. For example it is possible to open a ring with device name "multi:ethX;ethY;ethZ".

- Userspace RING ("userspace") module.
  This module allows an application to send packets to another process leveraging on the standard PF_RING API by creating virtual devices (e.g. usrX, where X is a unique identifier for the userspace ring). In order to do this, the sending application has to open a ring by using as device name "userspace:usrX" (where "userspace:" identifies the Userspace RING module), while the receiving application has to open a ring in the standard way by using as device name "usrX".

- Libzero consumer ("dnacluster") module.
  This module can be used to attach to a DNA Cluster allowing the application to send and receive packets leveraging on the standard PF_RING API. The sending application has to open a ring by using as device name "dnacluster:X@Y" where X is the cluster identifier and Y is the consumer identifier, or "dnacluster:X" for auto-assigning the consumer identifier.

8. 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 the 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.

8.1. 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.

8.2. 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.

8.3. 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 than 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 queues, 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.

As stated in the previous chapter, PF_RING 4.7 has a modular architecture. In order to indicate to the library which module we are willing to use, it is possible to prepend the module name to the device name, separated by a colon (e.g. dna:dnaX@Y for the dna module, dag:dagX:Y for the dag module, "multi:ethA@X;ethB@Y;ethC@Z" for the Link Aggregation module, "dnacluster:A@X" for the Cluster consumer module).

8.4.PF_RING: SOCKET Initialization

pfring* pfring_open(char *device_name, u_int32_t caplen, u_int flags)

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

- You can use physical (e.g. ethX) and virtual (e.g. tapX) devices, RX-queues (e.g. ethX@Y), and additional modules (e.g. dna:dnaX@Y, dag:dagX:Y, "multi:ethA@X;ethB@Y;ethC@Z", "dnacluster:A@X").
- You need super-user capabilities in order to open a device.

Input parameters:

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

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

flags
It allows several options to be specified on a compact format using bitmaps:

- PF_RING_REENTRANT
  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.
- PF_RING_LONG_HEADER
  If uset, PF_RING does not fill the field extended_hdr of struct pfring_pkthdr. If set, the extended_hdr field is also properly filled. In case you do not need extended information, set this value to 0 in order to speedup the operation.
- PF_RING_PROMISC
  The device is open in promiscuous mode.
- PF_RING_DNA_SYMMETRIC_RSS
  Set the hw RSS function to symmetric mode (both directions of the same flow go to the same hw queue). Supported by DNA drivers only.
- PF_RING_TIMESTAMP
  Force PF_RING to set the timestamp on received packets (usually it is not set when using zero-copy, for optimizing performance).
- PF_RING_HW_TIMESTAMP
  Enable hw timestamping, when available.

Return value:

On success a handle is returned, NULL otherwise.

u_int8_t pfring_open_multichannel(char *device_name,
u_int32_t caplen, u_int flags,
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

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

flags
See pfring_open() for details.

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.

8.5. 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.

8.6. PF_RING: Read Incoming Packets

int pfring_recv(pfring *ring, u_char** buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_int8_t 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. Note that this parameter is a pointer to a pointer, in order to enable zero-copy implementations (buffer_len must be set to 0).

buffer_len
The length of the memory area above. Note that the incoming packet is cut if it is too long for the allocated area. A length of 0 indicates to use the zero-copy optimization, when available.

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 (non-blocking), 1 in case of success, -1 in case of error.

int pfring_recv_parsed(pfring *ring, u_char** buffer, u_int buffer_len, struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet, u_int8_t level, u_int8_t add_timestamp, u_int8_t add_hash)

Same of pfring_recv(), with additional parameters to force packet parsing.

Input parameters not present in pfring_recv():

level
The header level where to stop parsing.

add_timestamp
Add the timestamp.

add_hash
Compute an IP-based bidirectional hash.

int pfring_loop(pfring *ring, pfringProcessPacket looper, const u_char *user_bytes, u_int8_t wait_for_packet)

This call processes packets until pfring_breakloop() is called or an error occurs.

Input parameters:

ring
The PF_RING handle.

looper
A callback to be called for each received packet. The parameters passed to this routine are: a pointer to a struct pfring_pkthdr, a pointer to the packet memory, and a pointer to user_bytes.

user_bytes
A pointer to user's data which is passed to the callback.

wait_for_packet
If 0 active wait is used to check the packet availability.

Return value:

A non-negative number if pfring_breakloop() is called. A negative number in case of error.

int pfring_next_pkt_time(pfring *ring, struct timespec *ts)

This call returns the arrival time of the next incoming packet, when available.

Input parameters:

ring
The PF_RING handle where we perform the check.

ts
The struct where the time will be stored.

Return value:

0 in case of success, a negative number in case of error.

int pfring_next_pkt_raw_timestamp(pfring *ring, u_int64_t *timestamp_ns)

This call returns the raw timestamp of the next incoming packet, when available. This is available with adapters supporting rx hardware timestamping only.

Input parameters:

ring
The PF_RING handle where we perform the check.

timestamp_ns
Where the timestamp will be stored.

Return value:

0 in case of success, a negative number in case of error.

8.7. PF_RING: Ring Clusters

int pfring_set_cluster(pfring *ring, u_int clusterId, cluster_type the_type)

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.

the_type
The cluster type (per flow or round robin).

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.

8.8. PF_RING: Packet Reflection

Packet reflection is the ability to bridge packets in kernel without sending them to userspace and back. 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.

8.9. 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.

8.10. 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.

8.10.1.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 removed.

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).

int pfring_purge_idle_rules(pfring *ring, u_int16_t inactivity_sec);

Remove filtering rules inactive for the specified number of seconds.

Input parameters:

ring
The PF_RING handle on which the rules will be removed.

inactivity_sec
The inactivity threshold.

Return value:

Zero if success, a negative value otherwise.

8.10.2.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).

int pfring_purge_idle_hash_rules(pfring *ring, u_int16_t inactivity_sec);

Remove hash filtering rules inactive for the specified number of seconds.

Input parameters:

ring
The PF_RING handle on which the rules will be removed.

inactivity_sec
The inactivity threshold.

Return value:

Zero if success, a negative value otherwise.

8.10.3.PF_RING: BPF Filtering

As of version 5.1, it is possible to set BPF filters through the PF_RING API. In order to do this, it's necessary to enable (this is the default) BPF support at compile time and link PF_RING-enabled applications against the -lpcap library (it is possible to disable the BPF support with "cd userland/lib/; ./configure --disable-bpf; make" to avoid linking libpcap).

int pfring_set_bpf_filter(pfring *ring, char* filter_buffer)

Set a BPF filter to an existing ring.

Input parameters:

ring
The PF_RING handle on which the filter will be set.

filter_buffer
The filter to set.

Return value:

Zero if success, a negative value otherwise.

int pfring_remove_bpf_filter(pfring *ring)

Remove the BPF filter.

Input parameters:

ring
The PF_RING handle.

Return value:

Zero if success, a negative value otherwise.

8.11. 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.

NOTE: Kernel packet filtering is not supported by DNA.

int pfring_add_hw_rule(pfring *ring, hw_filtering_rule *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.

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.

Return value:

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

int pfring_remove_hw_rule(pfring *ring, hw_filtering_rule *rule)

Remove the specified filtering rule from the NIC.

Input parameters:

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

rule
The filtering rule to be removed from the NIC.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_filtering_mode(pfring *ring, filtering_mode mode)

Sets the filtering mode (software only, hardware only, both software and hardware) in order to implicitly add/remove hardware rules by means of the same API functionality used for software (wildcard and hash) rules.

Input parameters:

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

mode
The filtering mode.

Return value:

Zero if success, a negative value otherwise.

8.12. 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 or 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.

8.13. PF_RING: Packet Transmission

Depending on the driver being used, packet transmission happens differently:

- Vanilla and PF_RING aware drivers: PF_RING does not accelerate the TX so the standard Linux transmission facilities are used. Do not expect speed advantage when using PF_RING on this mode.
- TNAPI: packet transmission is not supported.
- DNA: line rate transmission is supported.

int pfring_send(pfring *ring, u_char *pkt, u_int pkt_len, u_int8_t flush_packet)

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 MAC address up) and it will be transmitted as-is without any further manipulation.

Input parameters:

ring
The PF_RING handle on which the packet has to be sent.

pkt
The buffer containing the packet to send.

pkt_len
The length of the pkt buffer.

flush_packet
1 = Flush possible transmission queues. If set to 0, you will decrease your CPU usage but at the cost of sending packets in trains and thus at larger latency.

Return value:

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

int pfring_send_ifindex(pfring *ring, u_char *pkt, u_int pkt_len, u_int8_t flush_packet, int if_index)

Same of pfring_send(), with the possibility to specify the outgoing interface index.

Input parameters not present in pfring_send():

if_index
The interface index assigned to the outgoing device.

int pfring_send_get_time(pfring *ring, u_char *pkt, u_int pkt_len, struct timespec *ts)

This function allows to send a raw packet returning the exact time (ns) it has been sent on the wire. Note that this is available when the adapter supports tx hardware timestamping only and might affect performance.

Input parameters not present in pfring_send():

ts
The struct where the tx timestamp will be stored.

int pfring_send_last_rx_packet(pfring *ring, int tx_interface_id)

Send the last received packet to the specified device. This is an optimization working with standard PF_RING only.

Input parameters:

ring
The PF_RING handle on which the packet has been received.

tx_interface_id
The outgoing interface index.

Return value:

Zero if success, a negative value otherwise.

8.14. 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_disable_ring(pfring *ring)

Disable a ring.

Input parameters:

ring
The PF_RING handle to disable.

Return value:

Zero if success, a negative value otherwise.

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)

Tell 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_socket_mode(pfring *ring, socket_mode mode)

Tell PF_RING if the application needs to send and/or receive packets to/from the socket.

Input parameters:

ring
The PF_RING handle to enable.

mode
The socket mode (send, receive or both send and receive).

Return value:

Zero if success, a negative value otherwise.

int pfring_poll(pfring *ring, u_int wait_duration)

Performs passive wait on a PF_RING socket, similar to the standard poll(), taking care of data structures synchronization.

Input parameters:

ring
The PF_RING socket to poll.

wait_duration
The poll timeout in msec.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_poll_watermark(pfring *ring, u_int16_t watermark)

Whenever a user-space application has to wait until incoming packets arrive, it can instruct PF_RING not to return from poll() call unless at least "watermark" packets have been returned. A low watermark value such as 1, reduces the latency of poll() but likely increases the number of poll() calls. A high watermark (it cannot exceed 50% of the ring size, otherwise the PF_RING kernel module will top its value) instead reduces the number of poll() calls but slightly increases the packet latency. The default value for the watermark (i.e. if user-space applications do not manipulate is value via this call) is 128.

Input parameters:

ring
The PF_RING handle to enable.

watermark
The packet poll watermark.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_tx_watermark(pfring *ring, u_int16_t watermark)

Set the number of packets that have to be enqueued in the egress queue before being sent on the wire.

Input parameters:

ring
The PF_RING handle to enable.

watermark
The tx watermark.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_poll_duration(pfring *ring, u_int duration)

Set the poll timeout when passive wait is used.

Input parameters:

ring
The PF_RING handle to enable.

duration
The poll timeout in msec.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_application_name(pfring *ring, char *name)

Tell 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   : 13 [4.7.1]
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.

int pfring_get_bound_device_address(pfring *ring, u_char mac_address[6])

Returns the MAC address of the device bound to the socket.

Input parameters:

ring
The PF_RING handle to query.

mac_address
The memory area where the MAC address will be copied.

Return value:

Zero if success, a negative value otherwise.

int pfring_get_bound_device_id(pfring *ring, int* device_id)

Returns the MAC address of the device bound to the socket.

Input parameters:

ring
The PF_RING handle to query.

mac_address
The memory area where the MAC address will be copied.

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.

int pfring_enable_rss_rehash(pfring *ring)

Tells PF_RING to rehash incoming packets using a bi-directional hash function.

Input parameters:

ring
The PF_RING handle to query.

Return value:

Zero if success, a negative value otherwise.

int pfring_get_device_clock(pfring *ring, struct timespec *ts)

Reads the time from the device hardware clock, when the adapter supports hardware timestamping.

Input parameters:

ring
The PF_RING handle.

ts
The struct where time will be stored.

Return value:

Zero if success, a negative value otherwise.

int pfring_set_device_clock(pfring *ring, struct timespec *ts)

Sets the time in the device hardware clock, when the adapter supports hardware timestamping.

Input parameters:

ring
The PF_RING handle.

ts
The time to be set.

Return value:

Zero if success, a negative value otherwise.

int pfring_adjust_device_clock(pfring *ring, struct timespec *offset, int8_t sign)

Adjust the time in the device hardware clock with an offset, when the adapter supports hardware timestamping.

Input parameters:

ring
The PF_RING handle.

offset
The time offset.

sign
The offset sign.

Return value:

Zero if success, a negative value otherwise.

int pfring_enable_hw_timestamp(pfring *ring, char *device_name, u_int8_t enable_rx, u_int8_t enable_tx)

Enables rx and tx hardware timestamping, when the adapter supports it.

Input parameters:

ring
The PF_RING handle.

device_name
The name of the device where timestamping will be enabled.

enable_rx
Flag to enable rx timestamping.

enable_tx
Flag to enable tx timestamping.

Return value:

Zero if success, a negative value otherwise.

int pfring_parse_pkt(u_char *pkt, struct pfring_pkthdr *hdr, u_int8_t level, u_int8_t add_timestamp, u_int8_t add_hash)

Parse a packet.

It expects that the hdr memory is either zeroed or contains valid values for the current packet, in order to avoid parsing twice the same packet headers. This is implemented by controlling the l3_offset and l4_offset fields, indicating that respectively the L2 and L3 layers have been parsed when other than zero.

Input parameters:

pkt
The packet buffer.

hdr
The header to be filled.

level
The header level where to stop parsing.

add_timestamp
Add the timestamp.

add_hash
Compute an IP-based bidirectional hash.

Return value:

A non-negative number indicating the topmost header level on success, a negative value otherwise.

8.15. The C++ PF_RING interface

The C++ interface (see. PF_RING/userland/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);

9. libzero for DNA

This library implements a zero-copy Inter Process Communication, so that it can be used both in multi-thread and multi-process applications. As reported in the introduction it provides two main components: the DNA Cluster and the DNA Bouncer.

9.1.The DNA Cluster

The DNA Cluster implements packet clustering, so that all applications belonging to the same cluster can share incoming packets in zero-copy using a user-defined balancing function. Applications can also transmit packets in zero-copy. Each application reads/sends packets from/to a "slave" socket.

A master thread/application is responsible of dispatching incoming packets to the slaves by using an user-defined balancing function (the default one is a bidirectional IP-based hashing function). It can also act as a fan-out, delivering the same packet to multiple slave threads/applications, without the slowest consumer to affect the performance of faster ones.

The cluster allows application to process packets "with holes" (i.e. do not process packets in sequence), moving to the next incoming packet even though the previous one has not been processed yet.

9.1.1.The Master API

pfring_dna_cluster* dna_cluster_create(u_int32_t cluster_id, u_int32_t num_apps, u_int32_t flags)

Create a new DNA Cluster handle. The cluster just created has no ring associated.

Input parameters:

cluster_id
The cluster identifier.

num_apps
The number of slave applications/threads.

flags
A mask for enabling additional extensions, such as the "direct forwarding" for transmitting incoming packets to an outgoing interface according to the return values of he distribution function.

Return value:

The cluster handle.

int dna_cluster_register_ring(pfring_dna_cluster *handle, pfring *ring)

Add a PF_RING socket to the DNA Cluster.

Input parameters:

handle
The DNA Cluster handle.

ring
The PF_RING handle.

Return value:

Zero if success, a negative value otherwise.

void dna_cluster_set_cpu_affinity(pfring_dna_cluster *handle, u_int32_t rx_core_id, u_int32_t tx_core_id)

Bind the RX and TX master threads to the given core ids. In clusters, threads are used to poll (RX) and send (TX) cluster packets. This function is used to specify thread affinity across cores.

Input parameters:

handle
The DNA Cluster handle.

rx_core_id
The core id for the RX thread.

tx_core_id
The core id for the TX thread.

int dna_cluster_set_mode(pfring_dna_cluster *handle, socket_mode mode)

Set the cluster mode: receive (TX only), send (RX and TX), or both receive and send (RX and TX).

Input parameters:

handle
The DNA Cluster handle.

mode
The cluster mode.

Return value:

Zero if success, a negative value otherwise.

void dna_cluster_set_distribution_function(pfring_dna_cluster *handle, pfring_dna_cluster_distribution_func func)

Set the packet distribution function (the default function is a bidirectional IP-based hash function). This call allows developers to specify their own custom function.

Input parameters:

handle
The DNA Cluster handle.

func
The distribution function.

void dna_cluster_set_wait_mode(pfring_dna_cluster *handle, u_int32_t active_wait)

Set the ingress packet wait mode: passive (poll) or active wait.

Input parameters:

handle
The DNA Cluster handle.

active_wait
A boolean value: 0 for passive mode, 1 for active mode.

Return value:

Zero if success, a negative value otherwise.

int dna_cluster_enable(pfring_dna_cluster *handle)

Enable the cluster.

Input parameters:

handle
The DNA Cluster handle.

Return value:

Zero if success, a negative value otherwise.

int dna_cluster_disable(pfring_dna_cluster *handle)

Disable the cluster.

Input parameters:

handle
The DNA Cluster handle.

Return value:

Zero if success, a negative value otherwise.

int dna_cluster_stats(pfring_dna_cluster *handle, u_int64_t *tot_rx_packets, u_int64_t *tot_tx_packets, u_int64_t *tot_rx_processed)

Return cluster statistics.

Input parameters:

handle
The DNA Cluster handle.

tot_rx_packets
Total amount of received packets.

tot_tx_packets
Total amount of sent packets.

tot_rx_processed
Total amount of packets processed by the slaves.

Return value:

Zero if success, a negative value otherwise.

void dna_cluster_destroy(pfring_dna_cluster *handle)

Destroy the cluster, and close the bound PF_RING sockets.

Input parameters:

handle
The DNA Cluster handle.

Return value:

Zero if success, a negative value otherwise.

9.1.2.The Slave API

A DNA Cluster slave thread/application uses a superset of the PF_RING API, granting backward scompatibility with all existing applications. Later on this document you can find the differences between the two sets.

pfring_pkt_buff* pfring_alloc_pkt_buff(pfring *ring)

Return a packet buffer handle. The memory is allocated by PF_RING into the kernel and it is managed by PF_RING (i.e. no free() on this memory) using the pfring_XXX_XXX calls.

Input parameters:

ring
The PF_RING handle.

Return value:

The buffer handle just returned.

void pfring_release_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle)

Free a packet buffer handle previously allocated by pfring_alloc_pkt_buff.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

int pfring_recv_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle, struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet)

Receive a packet filling the buffer pointed by pkt_handle instead of returning a new buffer. In a nutshell, the returned packet is put on the passed function argument.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle where the incoming packets will be placed.

hdr
The PF_RING header.

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 (non-blocking), 1 in case of success, -1 in case of error.

int pfring_send_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle, u_int8_t flush_packet)

Send the packet pointed by the pkt_handle buffer handle. Note: this function resets the content of the buffer handle so if you need to keep its content, make sure you copy the data before you call it.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

flush_packet
Flush possible transmission queues.

Return value:

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

u_char* pfring_get_pkt_buff_data(pfring *ring, pfring_pkt_buff *pkt_handle)

Return a pointer to the buffer pointed by the packet buffer handle.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

Return value:

The pointer to the packet buffer.

void pfring_set_pkt_buff_len(pfring *ring, pfring_pkt_buff *pkt_handle, u_int32_t len)

Sets the length of the packet. This function call is not necessary unless you want to custom set the packet length, instead of using the size from the received packet.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

len
The packet len.

void pfring_set_pkt_buff_ifindex(pfring *ring, pfring_pkt_buff *pkt_handle, int if_index)

Binds the buffer handle (handling a packet) to an interface id. This function call is useful to specify the egress interface index.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

if_index
The interface id.

void pfring_add_pkt_buff_ifindex(pfring *ring, pfring_pkt_buff *pkt_handle, int if_index)

Adds an interface id to the bound interface ids of the buffer handle. This is used to specify the egress interfaces (fan-out) of a packet buffer.

Input parameters:

ring
The PF_RING handle.

pkt_handle
The buffer handle.

if_index
The interface id.

9.2.DNA Bouncer

The DNA Bouncer switches packets across two interfaces in zero-copy, leaving the user the ability to specify a function that can decide, packet-by-packet, whether a given packet has to be forwarded or not. The bouncer is directional, meaning that packets are copied only on one direction (ingress to egress rings). If you need a bi-directional copy you need to create two bouncers.

9.2.1.The DNA Bouncer API

pfring_dna_bouncer *pfring_dna_bouncer_create(pfring *ingress_ring, pfring *egress_ring)

Create a new DNA Bouncer handle.

Input parameters:

ingress_ring
The socket where packets will be read.

egress_ring
The socket where packets will be sent.

Return value:

The DNA Bouncer handle.

int pfring_dna_bouncer_set_mode(pfring_dna_bouncer *handle, dna_bouncer_mode mode)

Set the DNA Bouncer mode to one-way (default) or two-way.

Input parameters:

handle
The DNA Bouncer handle.

mode
The mode: one_way_mode or two_way_mode.

Return value:

Zero if success, a negative value otherwise.

int pfring_dna_bouncer_loop(pfring_dna_bouncer *handle, pfring_dna_bouncer_decision_func func, const u_char *user_bytes, u_int8_t wait_for_packet)

Enable the DNA Bouncer.

Input parameters:

handle
The DNA Bouncer handle.

func
The packet processing function.

user_bytes
A pointer to user data.

wait_for_packet
If 0 active wait is used to check the packet availability.

Return value:

Zero if success, a negative value otherwise.

void pfring_dna_bouncer_breakloop(pfring_dna_bouncer *handle)

Stop the bouncer.

Input parameters:

handle
The DNA Bouncer handle.

void pfring_dna_bouncer_destroy(pfring_dna_bouncer *handle)

Destroy the bouncer and close the bound PF_RING sockets.

Input parameters:

handle
The DNA Bouncer handle.

9.3.Code Snippets for Common Use Cases

9.3.1.DNA Cluster: receive a packet and put it aside

```
pfring_pkt_buff *pkt_handle = pfring_alloc_pkt_buff(ring);

if (pkt_handle != NULL) {

  rc = pfring_recv_pkt_buff(ring, pkt_handle, &hdr, wait_for_packet);

  if (rc > 0) {
    /* put the packet aside and do something later on */
    enqueue_packet(pkt_handle);
  }
}



pkt_handle = dequeue_packet();

/* do something with the packet and release it */
buffer = pfring_get_pkt_buff_data(ring, pkt_handle);
pfring_release_pkt_buff(ring, pkt_handle);
```

9.3.2.DNA Cluster: receive a packet and send it in zero-copy

```
pfring_pkt_buff *pkt_handle = pfring_alloc_pkt_buff(ring);

if (pkt_handle != NULL) {

 rc = pfring_recv_pkt_buff(ring, pkt_handle, &hdr, wait_for_packet);

 if (rc > 0) {
  if (forward_packet_to_another_interface) {
   pfring_set_pkt_buff_ifindex(ring[thread_id], pkt_handle, if_index);
  } else {
   /* bounce packet on the rx interface (already set in pkt_handle) */
  }

  pfring_send_pkt_buff(ring[thread_id], pkt_handle, 0);
 }
}
```

9.3.3.DNA Cluster: replace the default balancing function with a custom function

```
int hash_distribution_function(const u_char *buffer,
                               const u_int16_t buffer_len,
                               const u_int32_t num_slaves,
                               u_int32_t *id_mask,
                               u_int32_t *hash) {
  u_int32_t slave_idx;

  /* computing a bidirectional software hash */
  *hash = custom_hash_function(buffer, buffer_len);

  /* balancing on hash */
  slave_idx = (*hash) % num_slaves;
  *id_mask = (1 << slave_idx);
  return DNA_CLUSTER_PASS;
}

dna_cluster_set_distribution_function(dna_cluster_handle,
                                      hash_distribution_function);
```

9.3.4.DNA Cluster: replace the default balancing function with a fan-out function

```
int fanout_distribution_function(const u_char *buffer,
                                 const u_int16_t buffer_len,
                                 const u_int32_t num_slaves,
                                 u_int32_t *id_mask,
                                 u_int32_t *hash) {
  u_int32_t n_zero_bits = 32 - num_slaves;

  /* returning slave id bitmap */
  *id_mask = ((0xFFFFFFFF << n_zero_bits) >> n_zero_bits);
  return DNA_CLUSTER_PASS;
}

dna_cluster_set_distribution_function(dna_cluster_handle,
                                      fanout_distribution_function);
```

9.3.5.DNA Cluster: send an incoming packet directly without passing through a slave

```
int hash_distribution_function(const u_char *buffer,
                               const u_int16_t buffer_len,
                               const u_int32_t num_slaves,
                               u_int32_t *id_mask,
                               u_int32_t *hash) {
  u_int32_t socket_idx = get_out_socket_index();

  *id_mask = (1 << socket_idx);
  return DNA_CLUSTER_FRWD;
}

pfring_dna_cluster *dna_cluster_handle;
dna_cluster_handle = dna_cluster_create(cluster_id,
                                        num_threads,
                                        DNA_CLUSTER_DIRECT_FORWARDING);
int dna_cluster_set_mode(dna_cluster_handle, send_and_recv_mode);
dna_cluster_set_distribution_function(dna_cluster_handle,
                                      hash_distribution_function);
```

10. 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.

10.1. 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, implement by the function dummy_plugin_init(), is responsible for registering the plugin by calling the register_plugin() function. The parameter passed to the registration function is a data structure of type 'struct pfring_plugin_registration' that contains:

- plugin_id
  A unique integer plugin Id.
- 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_handle_skb
  A pointer to a function called whenever an incoming packet is received.
- 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_purge_idle
  A pointer to a function called whenever a user wants to purge a filtering rule that has set this plugin as action.
- pfring_plugin_free_rule_mem
  A pointer to a function called when a filtering rule that has set this plugin as action is removed.
- pfring_plugin_free_ring_mem
  A pointer to a function called when the plugin is unregistered (rmmod) or a ring using the plugin is removed. Free here any global 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.

- 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_del_rule_and_stop_rule_evaluation.

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).

10.2. PF_RING Plugin: Handle Incoming Packets

```
static int plugin_handle_skb( struct pf_ring_socket *pfr,
                              sw_filtering_rule_element *rule,
                              sw_filtering_hash_bucket *hash_rule,
                              struct pfring_pkthdr *hdr,
                              struct sk_buff *skb, int displ,
                              u_int16_t filter_plugin_id,
                              struct parse_buffer **parse_memory,
                              rule_action_behaviour *behaviour)
```

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.

parse_memory
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.

10.3. PF_RING Plugin: Filter Incoming Packets

```
int plugin_filter_skb(  struct pf_ring_socket *pfr,
                        sw_filtering_rule_element *rule,
                        struct pfring_pkthdr *hdr,
                        struct sk_buff *skb, int displ,
                        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.

10.4. PF_RING Plugin: Read Packet Statistics

```
int plugin_plugin_get_stats(  struct pf_ring_socket *pfr,
                              filtering_rule_element *rule,
                              filtering_hash_bucket  *hash_rule,
                              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.

10.5. 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.

11. 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 */
  u_int8_t locked;                  /* Do not purge */
  u_int8_t bidirectional;           /* Swap peers (Default: mono) */
  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 smac[ETH_ALEN], dmac[ETH_ALEN];  /* Use '0' (zero-ed MAC address) for
                                               any MAC address. This is applied
                                               to both source and destination */
  u_int16_t vlan_id;                /* Use '0' for any vlan */
  u_int8_t  proto;                  /* Use 0 for 'any' protocol */
  ip_addr   shost, dhost;           /* User '0' for any host. This is applied
                                       to both source and destination. */
  ip_addr   shost_mask, dhost_mask; /* IPv4/6 network mask */
  u_int16_t sport_low, sport_high;  /* All ports between port_low...port_high
                                       means 'any' port */
  u_int16_t dport_low, dport_high;  /* All ports between port_low...port_high
                                       means 'any' port */
} 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;
```

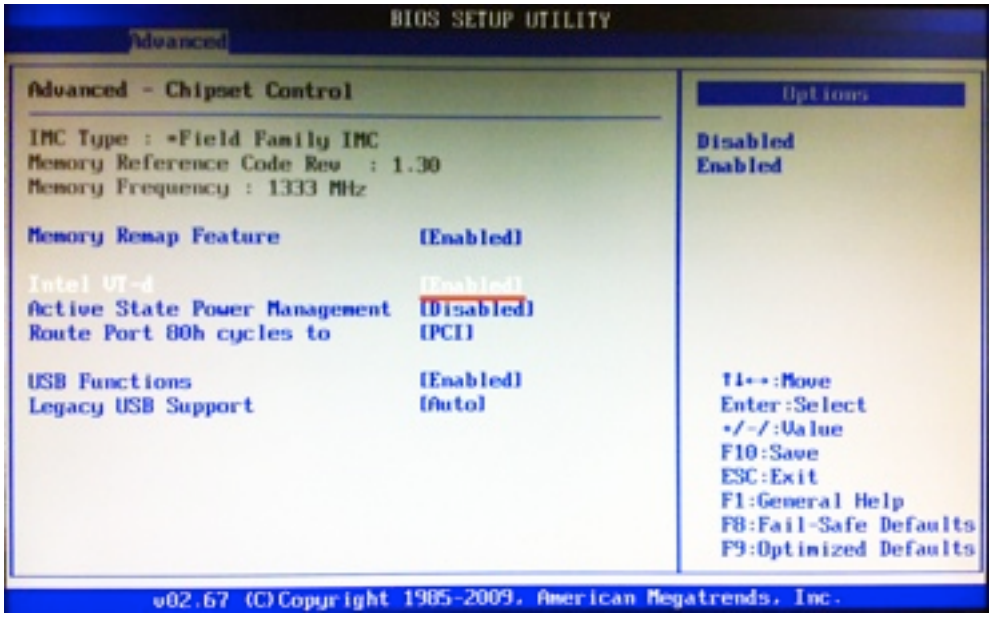
```
typedef enum {
  forward_packet_and_stop_rule_evaluation = 0,
  dont_forward_packet_and_stop_rule_evaluation,
  execute_action_and_continue_rule_evaluation,
  execute_action_and_stop_rule_evaluation,
  forward_packet_add_rule_and_stop_rule_evaluation,/* auto-filled hash rule or
                                                      via plugin_add_rule() */
  forward_packet_del_rule_and_stop_rule_evaluation,/* plugin_del_rule() only */
  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;
  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 struct {
  u_int64_t recv, drop;
} pfring_stat;
```

12. PF_RING DNA On Virtual Machines

Section 5.4 contains a brief introduction to the PF_RING DNA module, which allows you to manipulate packets at 10 Gbit wire speed for any packet size. Thanks to Virtualization Technologies based on IOMMUs (Intel VT-d or AMD IOMMU), it is now possible to assign a device to a given guest operating system, benefiting from the PF_RING DNA module within a VM (Virtual Machine). The following sections show how to configure VMware and KVM (the Linux-native virtualization system). XEN users can use similar system configurations.

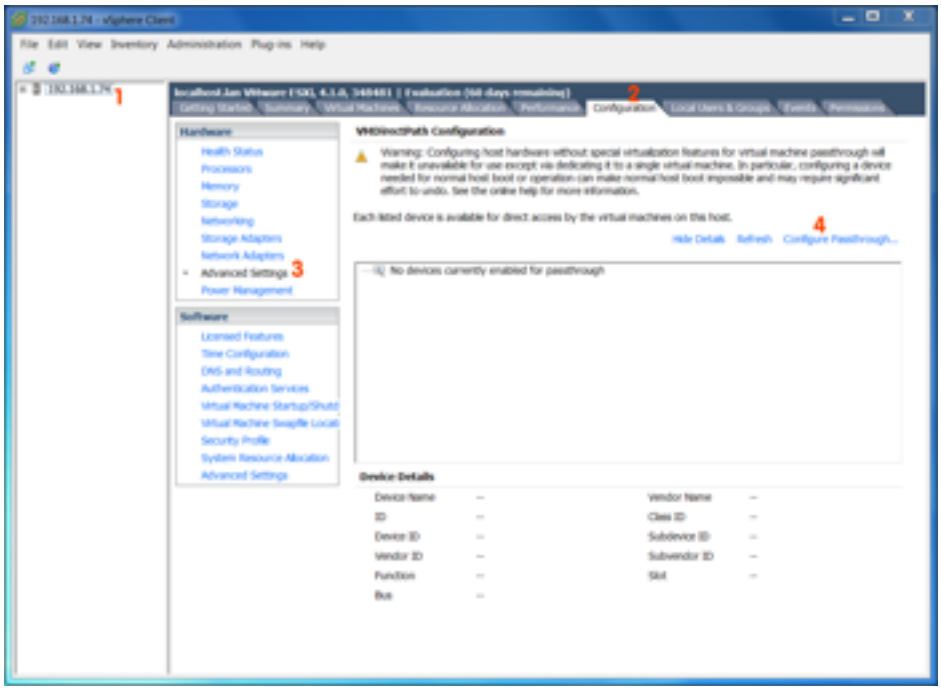
12.1. BIOS Configuration

First of all, make sure that your motherboard supports the PCI passthrough and check that it is enabled in your BIOS.

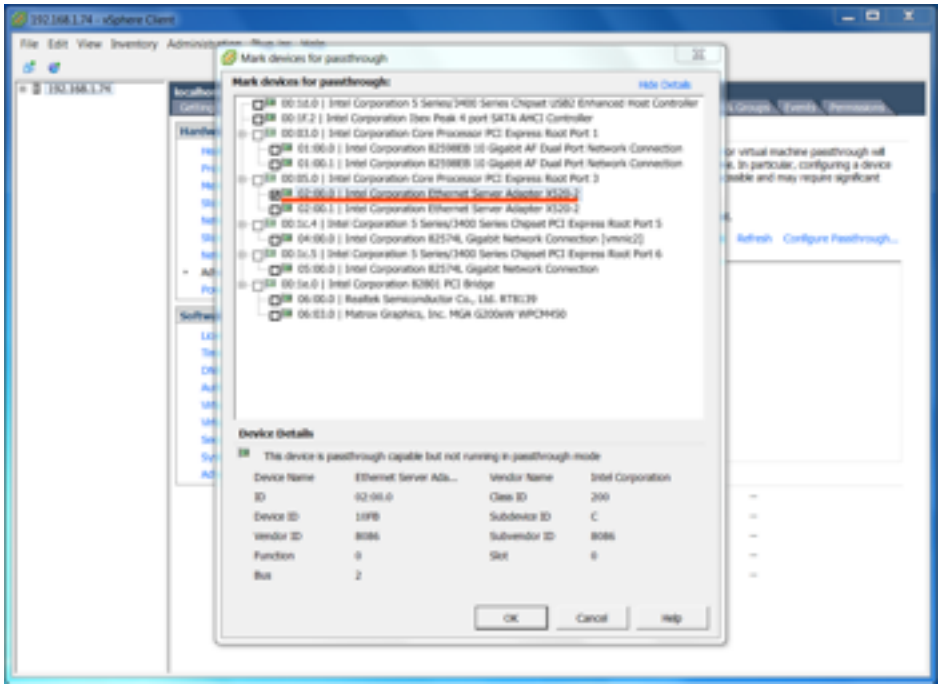


12.2.VMware ESX Configuration

In order to configure the PCI passthrough in VMware, open the vSphere Client and connect to the server. Select the server, go to "Configuration", "Advanced Settings", "Configure Passthrough".

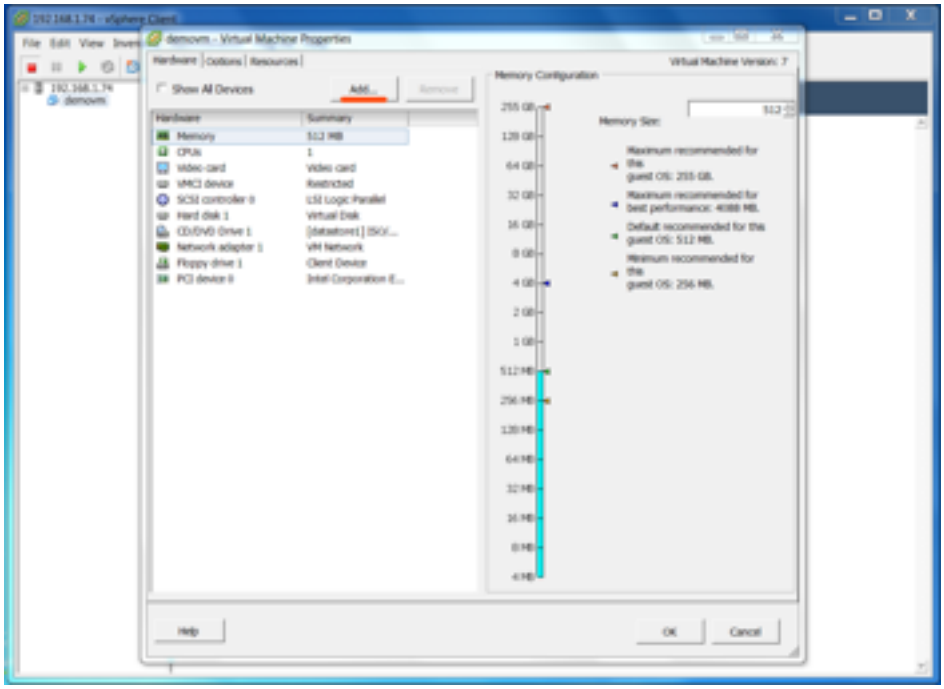


Select the devices you want to assign to the VMs.

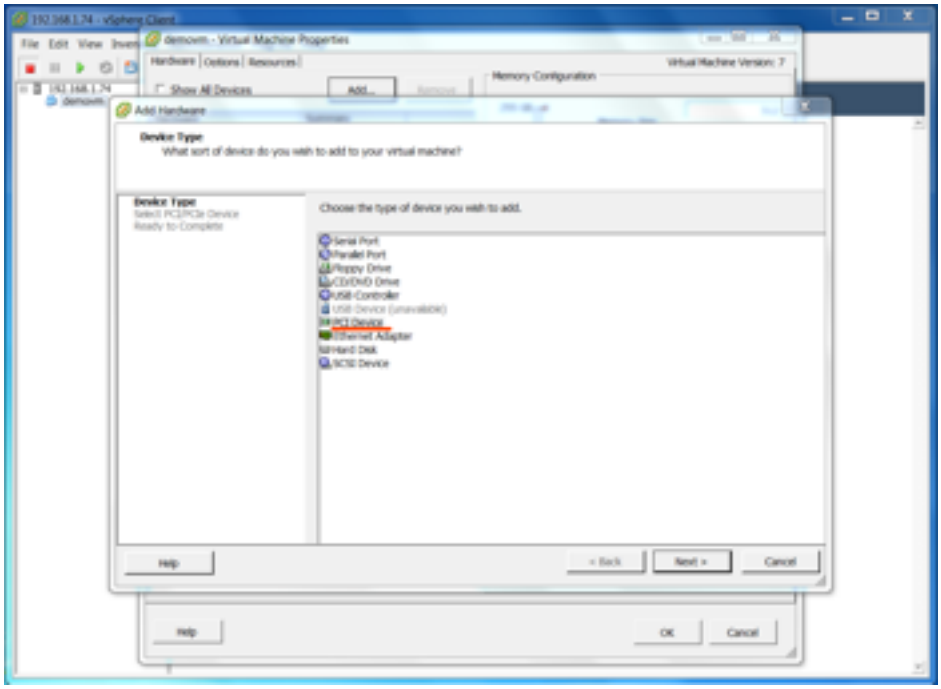


Reboot the server.

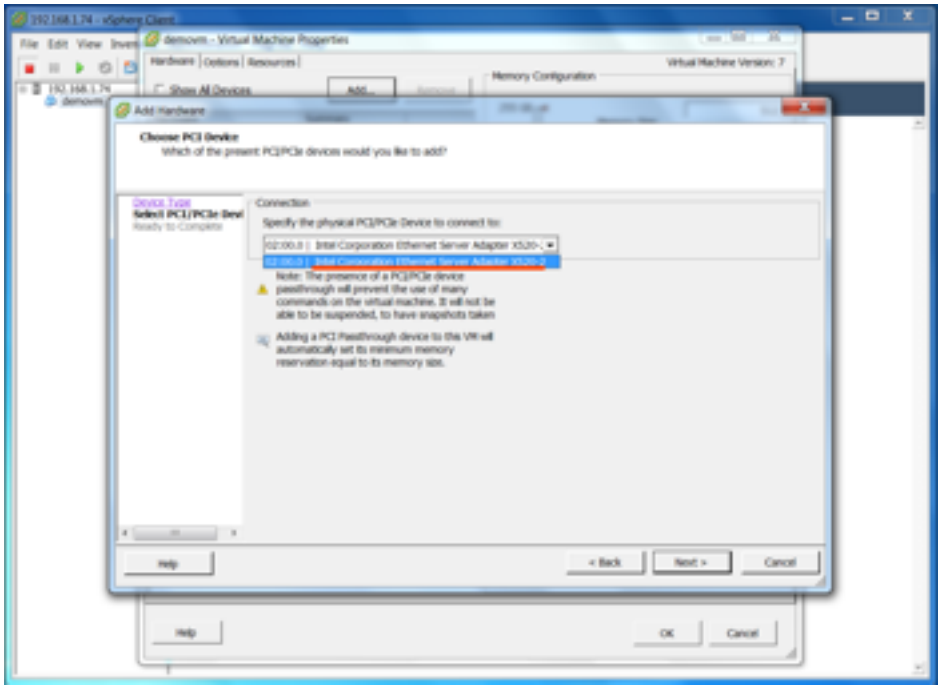
After the reboot, make sure that the VMs where the PCI device will be assigned is in the off state. Open the VM settings, and click on "Add..." in the "Hardware" tab.



Select "PCI Device".



Select the device to assign to the VM.



Boot the VM and install PF_RING with the DNA driver as in the native case.

12.3. KVM Configuration

In order to configure the PCI passthrough with KVM, make sure you have enabled these options in your kernel:

Bus options (PCI etc.)

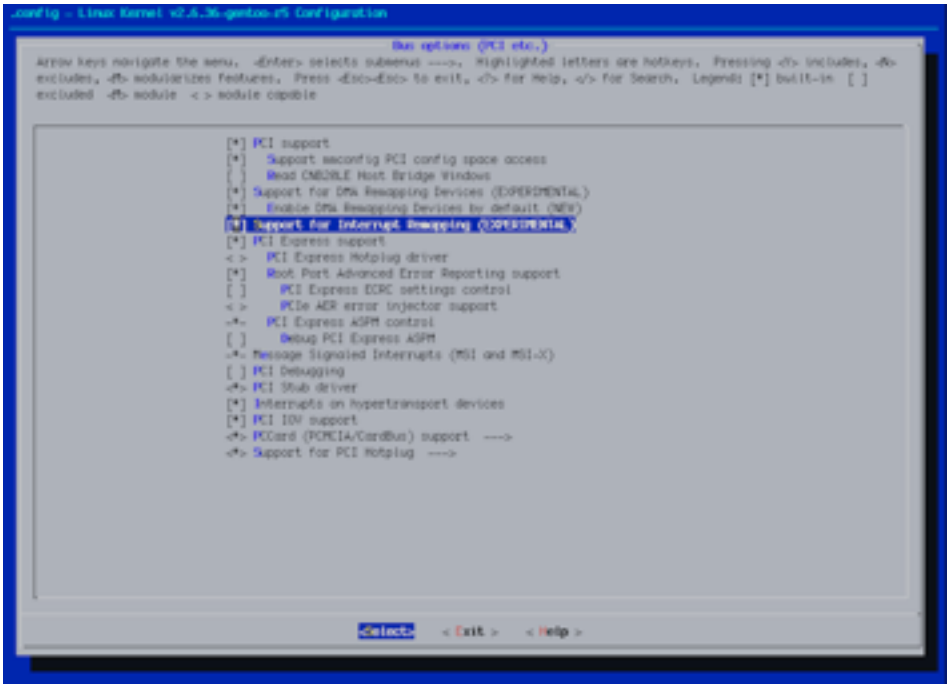
[*] Support for DMA Remapping Devices

[*] Enable DMA Remapping Devices

[*] Support for Interrupt Remapping

<*> PCI Stub driver

```
$ cd /usr/src/linux
$ make menuconfig
```




```
$ make
$ make modules_install
$ make install
```

(or use your distribution-specific way)

Pass "intel_iommu=on" as kernel parameter. For instance, if you are using grub, edit your /boot/grub/menu.lst this way:

```
title Linux 2.6.36
root (hd0,0)
kernel /boot/kernel-2.6.36 root=/dev/sda3 intel_iommu=on
```

Unbind the device you want to assign to the VM from the host kernel driver.

```
$ lspci -n
..
02:00.0 0200: 8086:10fb (rev 01)
..
$ echo "8086 10fb"  > /sys/bus/pci/drivers/pci-stub/new_id
$ echo 0000:02:00.0 > /sys/bus/pci/devices/0000:02:00.0/driver/unbind
$ echo 0000:02:00.0 > /sys/bus/pci/drivers/pci-stub/bind
```

Load KVM and start the VM.

```
$ modprobe kvm
$ modprobe kvm-intel
$ /usr/local/kvm/bin/qemu-system-x86_64 -m 512 -boot c \
      -drive file=virtual_machine.img,if=virtio,boot=on \
      -device pci-assign,host=02:00.0
```

Install and run PF_RING with the DNA driver as in the native case.