

# PF\_RING User Guide

Linux High Speed Packet Capture

Version 4.7.1

July 2011

© 2004-11 ntop.org

# 1. Table of Contents

Introduction .....	4
What's New with PF_RING User's Guide? .....	4
Welcome to PF_RING .....	4
PF_RING Installation .....	8
Linux Kernel Module Installation .....	8
Running PF_RING .....	9
Checking PF_RING Device Configuration .....	9
Libpfring and Libpcap Installation .....	10
Application Examples .....	10
PF_RING Additional Modules .....	11
PF_RING for Application Developers .....	12
The PF_RING API .....	12
Return Codes .....	12
PF_RING Device Name Convention .....	12
PF_RING: SOCKET Initialization .....	13
PF_RING: Device Termination .....	14
PF_RING: Read Incoming Packets .....	15
PF_RING: Ring Clusters .....	17
PF_RING: Packet Reflection .....	18
PF_RING: Packet Sampling .....	19
PF_RING: Packet Filtering .....	20
PF_RING: Wildcard Filtering .....	20
PF_RING: Hash Filtering .....	22
PF_RING: In-NIC Packet Filtering .....	23
PF_RING: Filtering Policy .....	24

PF_RING: Send Packets .....	24
PF_RING: Miscellaneous Functions.....	26
The C++ PF_RING interface.....	29
Writing PF_RING Plugins.....	30
Implementing a PF_RING Plugin.....	30
PF_RING Plugin: Handle Incoming Packets .....	31
PF_RING Plugin: Filter Incoming Packets .....	32
PF_RING Plugin: Read Packet Statistics .....	33
Using a PF_RING Plugin .....	34
PF_RING Data Structures.....	35
PF_RING DNA On Virtual Machines .....	37
BIOS Configuration .....	37
VMware ESX Configuration.....	38
KVM Configuration.....	41

## 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:

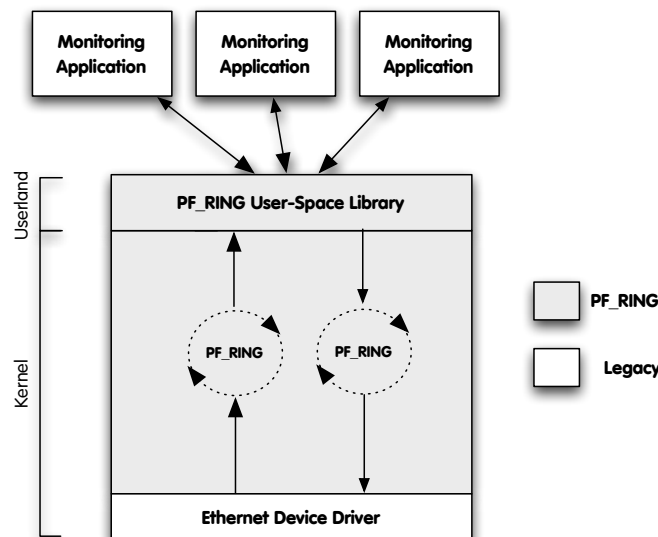
- PF\_RING installation and configuration.
- PF\_RING SDK.

### 2.1. What's New with PF\_RING User's Guide?

- Release 4.7 (July 2011)
  - Updated guide to PF\_RING version 4.7.1
- 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 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.

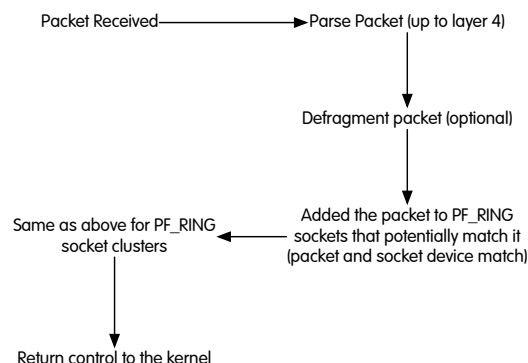
PF\_RING implements a new socket type (named PF\_RING) on which user-space applications can speak with the PF\_RING kernel module. Applications can obtain a PF\_RING handle, and issue API calls that are described later in this manual. A handle can be bound to a:

- 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.

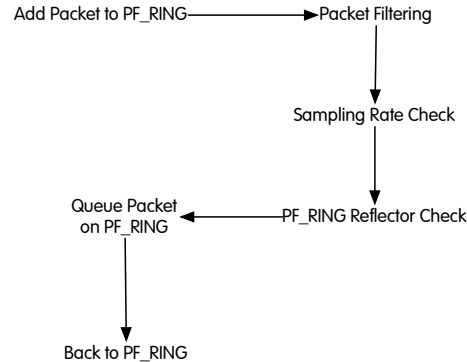
As specified above, packets are read from a memory ring allocated at creation time. Incoming packets are copied by the kernel module to the ring, and read by the user-space applications. No per-packet memory allocation/deallocation is performed. Once a packet has been read from the ring, the space used 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.

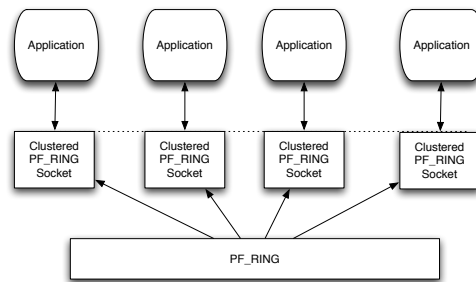
The packet journey in PF\_RING is quite long



before being queued into a PF\_RING ring.

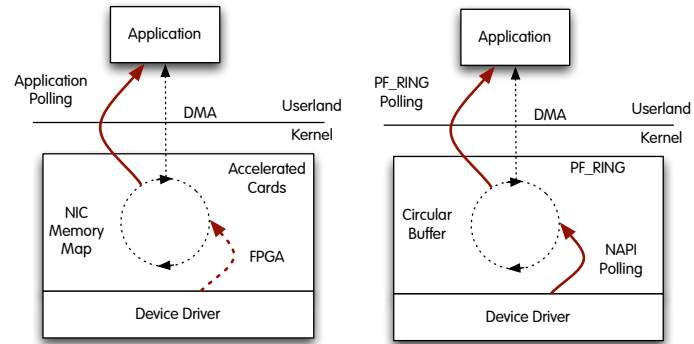


PF\_RING can also increase the performance of packet capture applications by implementing two mechanisms named balancing and clustering. These mechanisms allow applications, willing to partition the set of packets to handle, to handle a portion of the whole packet stream while sending all the remaining 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:

- 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.
- (Legacy) An automatic patch mechanism allows you to automatically patch a vanilla kernel with PF\_RING.

PF\_RING is downloaded by means of SVN as explained in [http://www.ntop.org/PF\\_RING.html](http://www.ntop.org/PF_RING.html)

The PF\_RING source code layout is the following:

- doc/
- drivers/
- kernel/
- legacy/
- Makefile
- README
- README.DNA
- userland/

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

### 4.1. 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 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 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).

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.

### 5.1. Checking PF\_RING Device Configuration

When PF\_RING is activated, a new entry `/proc/net/pf_ring` is created.

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

# cat /proc/net/pf_ring/info
Version          : 4.7.1
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.

## 5.2. Libpfiring and Libpcap Installation

Both `libpfiring` (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.

## 5.3. 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          preflect.c
dummy_plugin_pfcount.c   pfcount_bundle.c         ptwin.c
forwarder.c              pfcount_dummy_plugin.c   pwrite.c
interval.c               pfcount_multichannel.c   ringc.c
pcap2nspcap.c            pffilter_test.c          ringm.c
pcaps.c                  pfmap.c                  rings.c
pcount.c                 pfsend.c                 vdevice_simulator.c
pfbounce.c               pfsystest.c
pfcount.c                pftwin.c
$ make
```

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

```
# ./pfcount -i eth0
Using PF_RING v.4.7.1
...
=====
Absolute Stats: [19919331 pkts rcvd][0 pkts dropped]
Total Pkts=19919331/Dropped=0.0 %
19'919'331 pkts - 1'195'159'860 bytes [9'959'331.86 pkt/sec - 4'780.47
Mbit/sec]
=====
Actual Stats: 6654505 pkts [1'000.02 ms][6'654'305.37 pkt/sec]
=====
```

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

```
# ./pfsend -f 64byte_packets.pcap -n 0 -i eth1 -r 5
...
TX rate: [current 7'585'044.84 pps/5.10 Gbps][average 7'585'896.26 pps/
5.10 Gbps][total 15'172'187.00 pkts]
```

## 5.4. 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:

- The 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.
- The 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 two DNA-aware drivers are available:
  - ▶ e1000e (RX only). The driver can be downloaded from the <http://shop.ntop.org/> site.
  - ▶ ixgbe (RX and TX). The driver can be found in `drivers/intel/ixgbe`.
 With both drivers you can achieve wire rate at any packet size. 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 ethX autoneg off rx off tx off`).

## 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 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.
- (Legacy) 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 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:ethX@Y` for the `dna` module, `dag:dagX:Y` for the `dag` module).

## 6.4.PF\_RING: SOCKET 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 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:ethX@Y, dag:dagX:Y).
- 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).

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 in 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 in 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.

## 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 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_loop(pfring *ring, pfringProcessPacket loop, const u_char *user_bytes)
```

This call processes packets until pfring\_breakloop() is called or an error occurs.

Input parameters:

ring  
The PF\_RING handle.

loop  
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.



## 6.7. PF\_RING: Ring Clusters

```
int pfiring_set_cluster(pfiring *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 nutshell 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 or 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 pfiring_remove_from_cluster(pfiring *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_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.

## 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 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.

## 6.15. PF\_RING: Send Packets

```
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. 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.

flush\_packet  
Flush possible transmission queues.

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 returned 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_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 its 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_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      : 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.

`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 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.

## 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 highest 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;

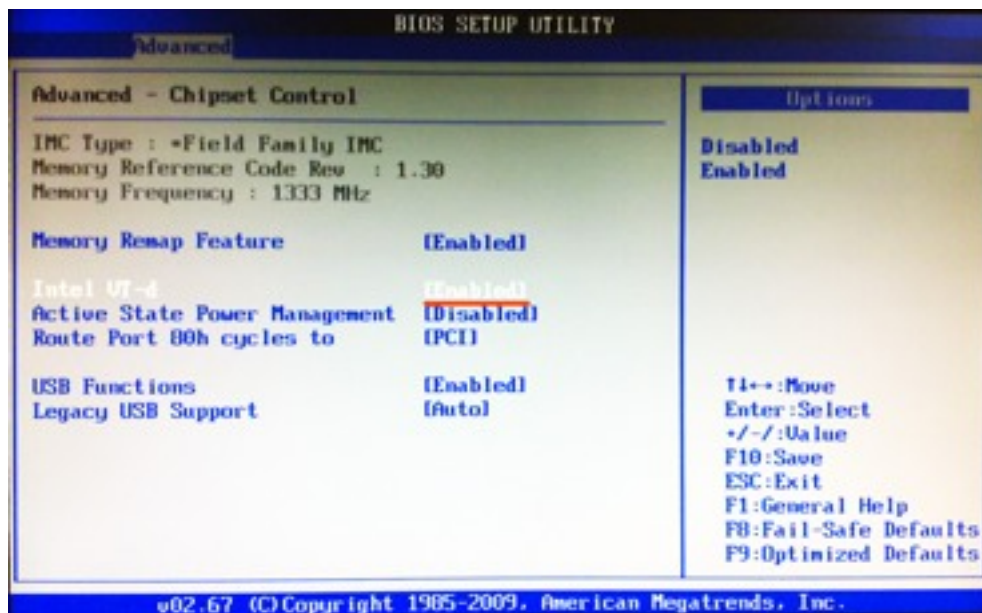
```

## 9. 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.

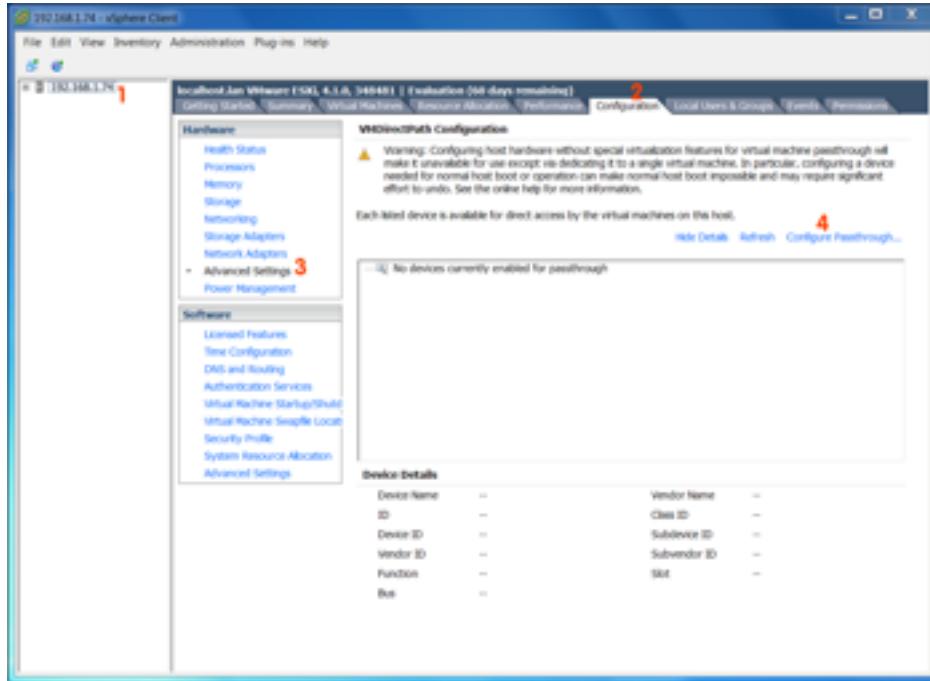
### 9.1. BIOS Configuration

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

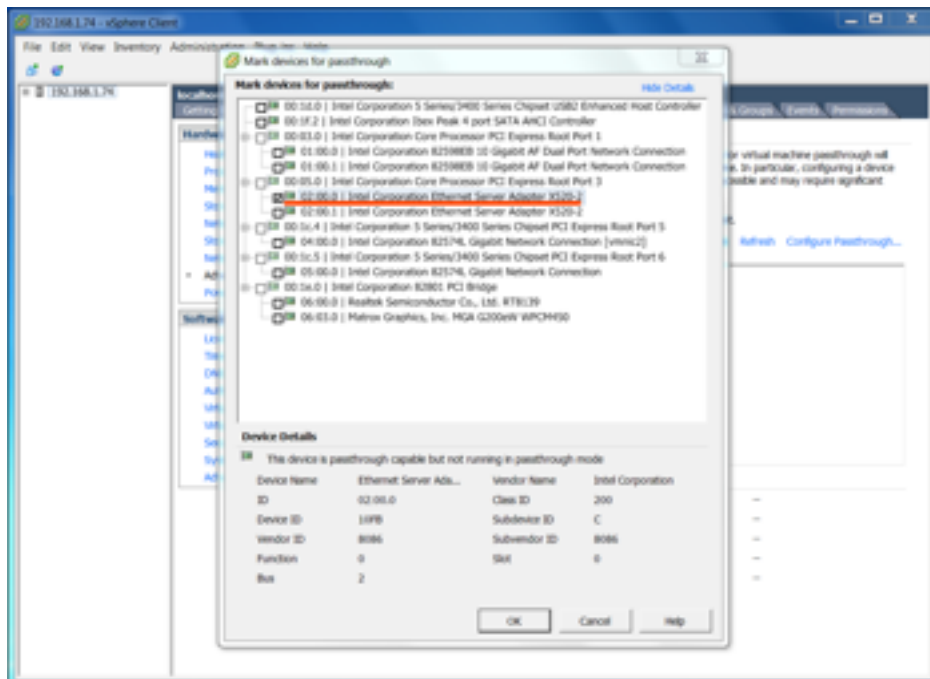


## 9.2.VMware ESX Configuration

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".

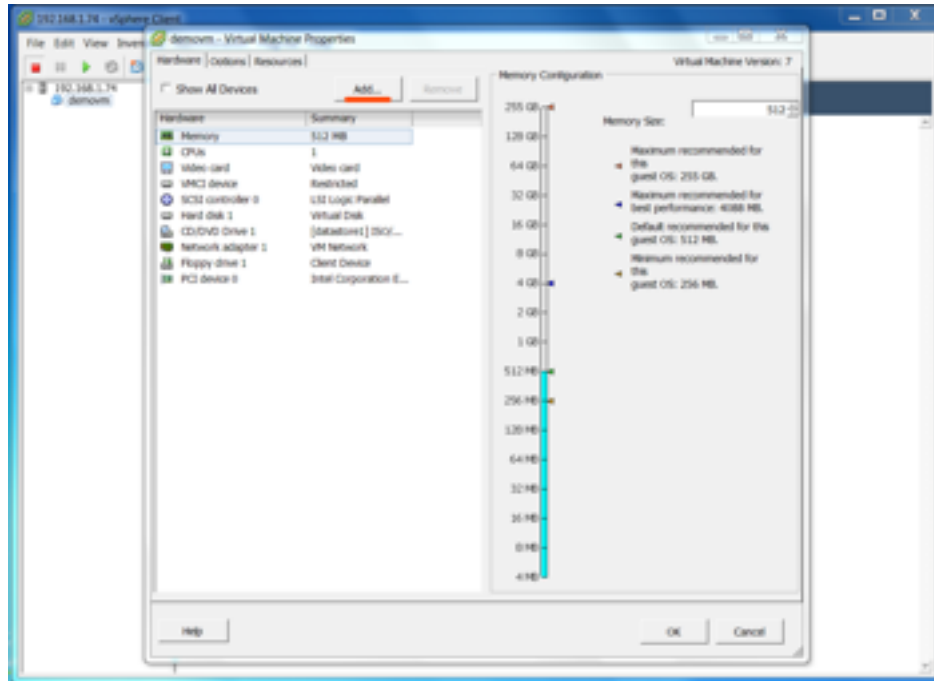


Select the devices you want to assign to the VMs.

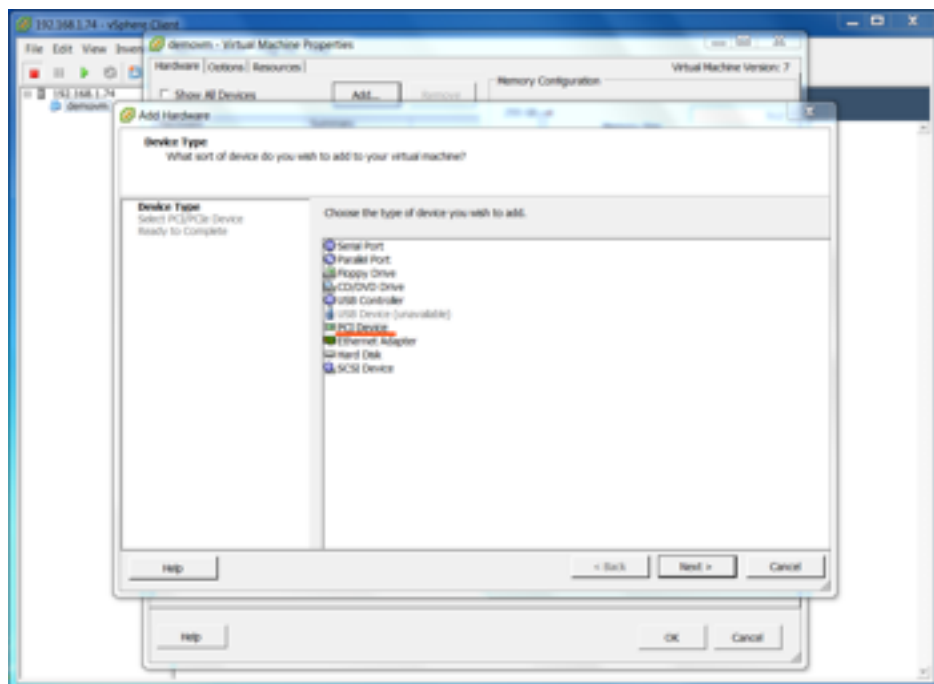


Reboot the server.

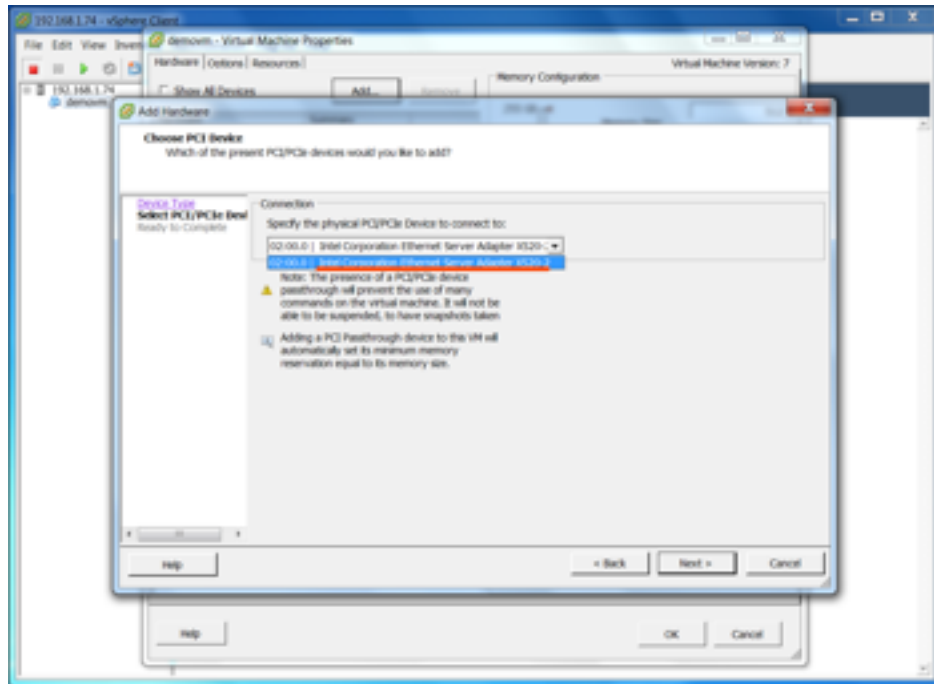
After the reboot, make sure that the VMs where the PCI device will be assigned is in the off state. Open the VM settings, and click on "Add..." in the "Hardware" tab.



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.



### 9.3. 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

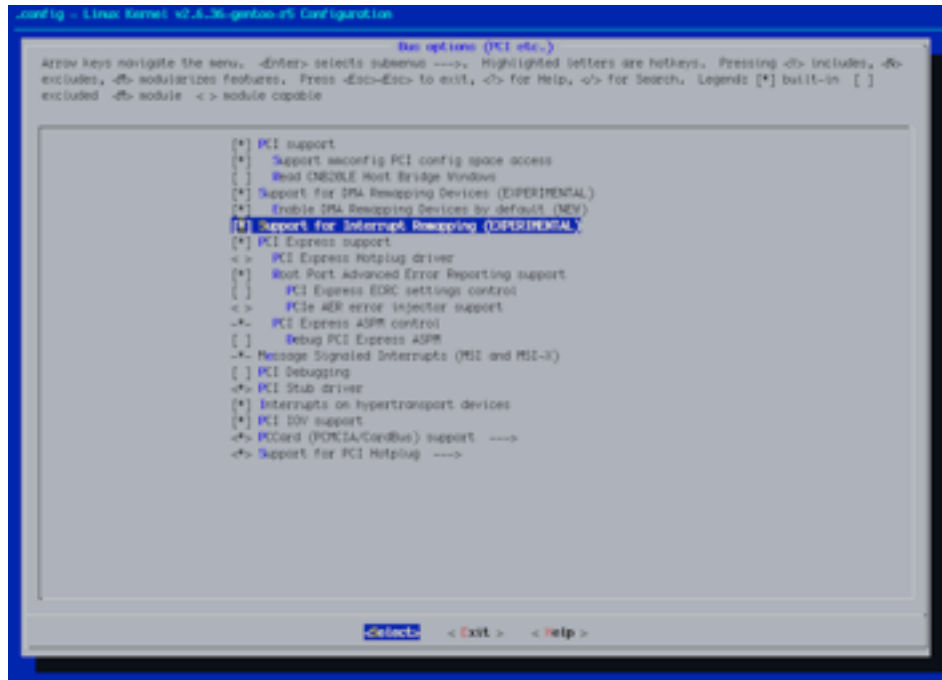
[\*] Enable DMA Remapping Devices

[\*] 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.