

# PF\_RING User Guide

Linux High Speed Packet Capture

Version 5.6.2  
Jan 2014

© 2004-13 ntop.org

# 1. Table of Contents

Introduction .....	5
What's New with PF_RING User's Guide? .....	5
Welcome to PF_RING .....	6
Packet Filtering .....	6
Packet Journey .....	7
Packet Clustering .....	7
PF_RING Driver Families .....	9
PF_RING-aware Drivers .....	9
TNAPI .....	9
DNA .....	10
Libzero for DNA .....	11
DNA Cluster .....	11
DNA Bouncer .....	11
PF_RING Installation .....	12
Linux Kernel Module Installation .....	12
Running PF_RING .....	13
Checking PF_RING Device Configuration .....	13
Libpfiring and Libpcap Installation .....	14
Application Examples .....	14
PF_RING Additional Modules .....	15
PF_RING for Application Developers .....	17
The PF_RING API .....	17
Return Codes .....	17
PF_RING Device Name Convention .....	17
PF_RING Packet Reflection .....	18

PF_RING Packet Filtering .....	19
PF_RING In-NIC Packet Filtering .....	20
libzero for DNA .....	21
The DNA Cluster .....	21
The Master API .....	21
The Slave API .....	21
DNA Bouncer .....	22
The DNA Bouncer API .....	22
Code Snippets for Common Use Cases .....	23
DNA Cluster: receive a packet and put it aside .....	23
DNA Cluster: receive a packet and send it in zero-copy .....	24
DNA Cluster: replace the default balancing function with a custom function .....	25
DNA Cluster: replace the default balancing function with a fan-out function .....	26
DNA Cluster: send an incoming packet directly without passing through a slave .....	27
Writing PF_RING Plugins .....	28
Implementing a PF_RING Plugin .....	28
PF_RING Plugin: Handle Incoming Packets .....	30
PF_RING Plugin: Filter Incoming Packets .....	31
PF_RING Plugin: Read Packet Statistics .....	32
Using a PF_RING Plugin .....	33
PF_RING Data Structures .....	34
PF_RING DNA On Virtual Machines .....	36
BIOS Configuration .....	36
VMware ESX Configuration .....	37
KVM Configuration .....	40



## 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.

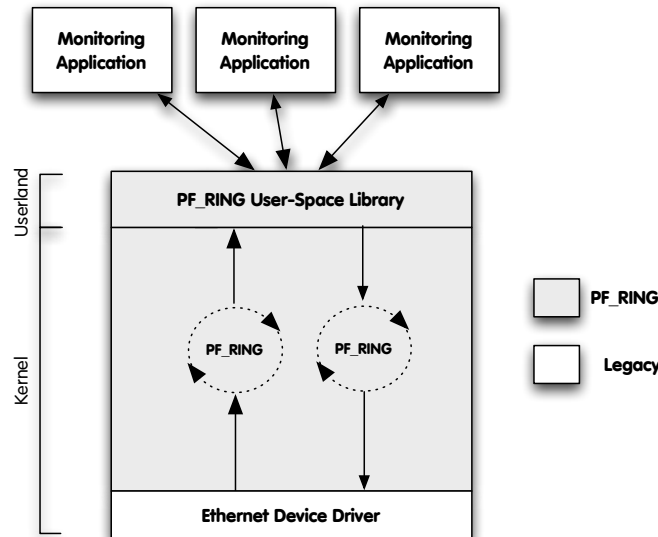
This manual is divided in two parts:

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

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

- Release 5.6.2 (Nov 2013)
  - Updated guide to PF\_RING version 5.6.2
- Release 5.4.0 (May 2012)
  - New libzero for zero-copy flexible packet processing on top of DNA.
- Release 5.2.1 (January 2012)
  - New API functions for managing hardware clocks and timestamps.
  - New kernel plugin callbacks.
- Release 4.7.1 (July 2011)
  - Described PF\_RING modular library and some modules (DAG, DNA)
- 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.

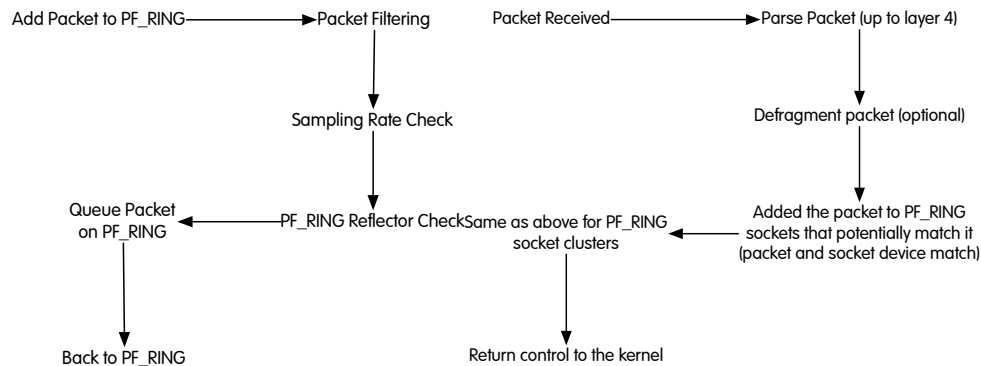
#### 3.1. Packet Filtering

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 on 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.

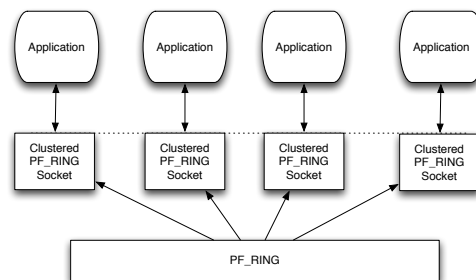
### 3.2. Packet Journey

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.

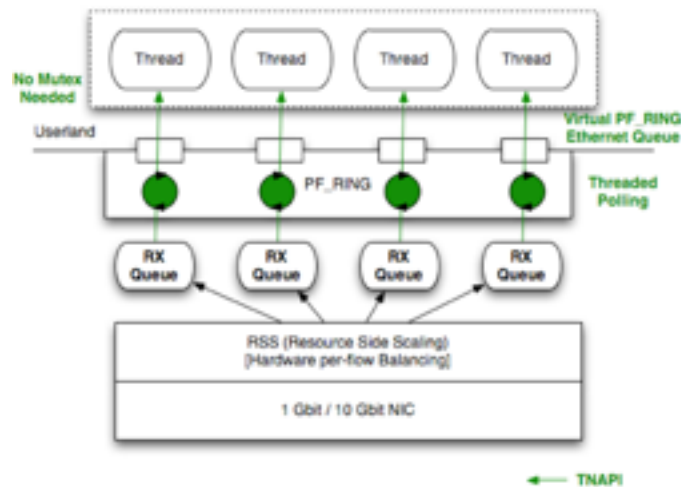
### 4.1. PF\_RING-aware Drivers

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.

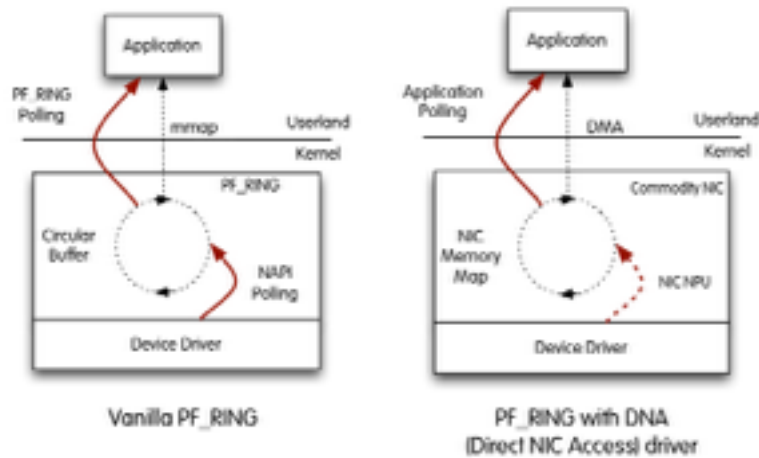
### 4.2. TNAPI

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.

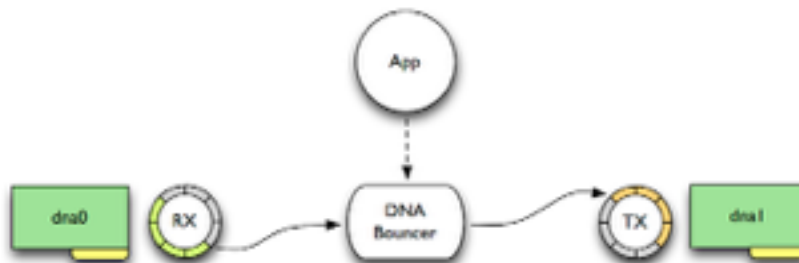
### 5.1.DNA Cluster



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 allows users 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 can be mono-directional (thus in case you want to implement bridging, two bouncer threads, one per direction, need to be instantiated) or bi-directional.

## 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/

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 memcp() to PF\_RING and also to the standard Linux path.
- transparent\_mode=2 (PF\_RING -aware drivers only)  
Packets are ONLY memcp() 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
PF_RING Version      : 5.5.3
Total rings          : 0
```

```
Standard (non DNA) Options
Ring slots           : 4096
Slot version         : 15
Capture TX           : Yes [RX+TX]
IP Defragment        : No
Socket Mode          : Standard
Transparent mode      : Yes [mode 0]
```

```
Total plugins          : 2
Cluster Fragment Queue : 0
Cluster Fragment Discard : 0
```

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.

```
# cat /proc/net/pf_ring/plugins_info
ID      Plugin
2       sip [SIP protocol analyzer]
12      rtp [RTP protocol analyzer]
```

## 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 statically-linked pcap-based applications need to be recompiled against the new 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_aggregator.c    pffilter_test.c
dummy_plugin_pfcount.c   pfcount_bundle.c        pflatency.c
interval.c               pfcount_dummy_plugin.c  pfmap.c
pcap2nspcap.c            pfcount_multichannel.c  pfsend.c
pcount.c                 pfdnabounce.c           pfsystest.c
pfbounce.c               pfdnacluster_master.c   pfutils.c
pfbridge.c               pfdnacluster_mt_rss_frwd.c pfwrite.c
pfcount.c                pfdnacluster_multithread.c preflect.c
```

```
pfcount_82599.c          pfdump.c
$ make
```

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

```
# ./pfcount -i dna0
Using PF_RING v.5.5.3
...
=====
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.
- Linux TCP/IP Stack injection ("stack") module.  
This module can be used to inject/capture packets to/from the Linux TCP/IP Stack, simulating the arrival/sending of those packets on an interface. The application has to open a ring by using as device name "stack:dnaX" where dnaX is the interface bound to the packets injected into the stack. In order to inject a packet to the stack `pfring_send()` has to be used, in order to capture outgoing packets `pfring_rcv()` has to be used.



## 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 overview.
- Extensions to the libpcap library for supporting legacy applications.

Please refer to the doxygen documentation (pfring.h header file) for functions descriptions.

### 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 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.

## 8.5. 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.

## 8.6. 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.

## 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

Please refer to the doxygen documentation for the pfring\_zero.h file (look for dna\_cluster\_\*).

#### 9.1.2. The Slave API

A DNA Cluster slave thread/application uses a superset of the PF\_RING API, granting backward compatibility with all existing applications.

In addition to the standard PF\_RING API the functions below are defined. Please refer to the doxygen documentation for pfring.h.

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

```
int pfring_set_pkt_buff_len(pfring *ring, pfring_pkt_buff *pkt_handle, u_int32_t len);
```

```
int pfring_set_pkt_buff_ifindex(pfring *ring, pfring_pkt_buff *pkt_handle, int if_index);
```

```
int pfring_add_pkt_buff_ifindex(pfring *ring, pfring_pkt_buff *pkt_handle, int if_index);
```

```
pfring_pkt_buff* pfring_alloc_pkt_buff(pfring *ring);
```

```
void pfring_release_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle);
```

```
int pfring_rcv_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle, struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet);
```

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

## 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 able to work in mode-directional mode, 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 bouncer threads/applications), or bi-directional mode.

### 9.2.1.The DNA Bouncer API

Please refer to the doxygen documentation for the `pfring_zero.h` file (look for `pfring_dna_bouncer_*`).

## 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_rcv_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 0 on 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 0 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 the length of the rule stats, or 0 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 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 */
    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;

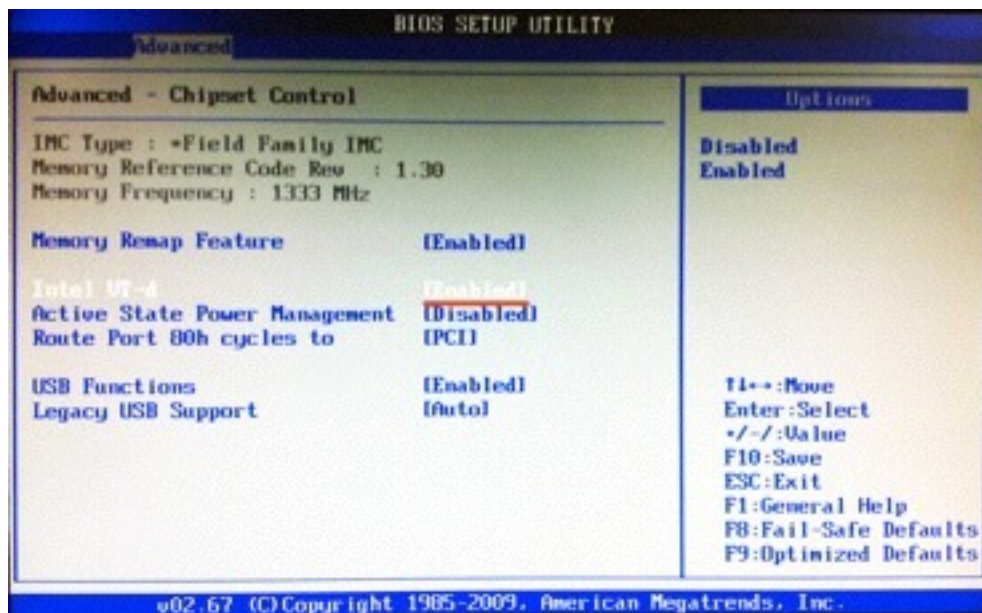
```

## 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.

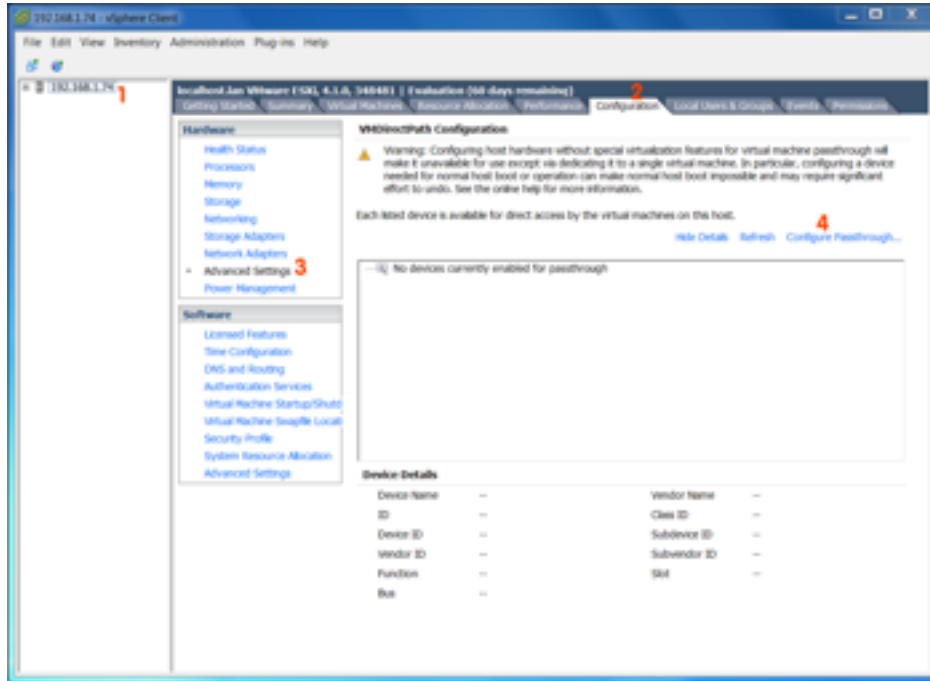
### 12.1. BIOS Configuration

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

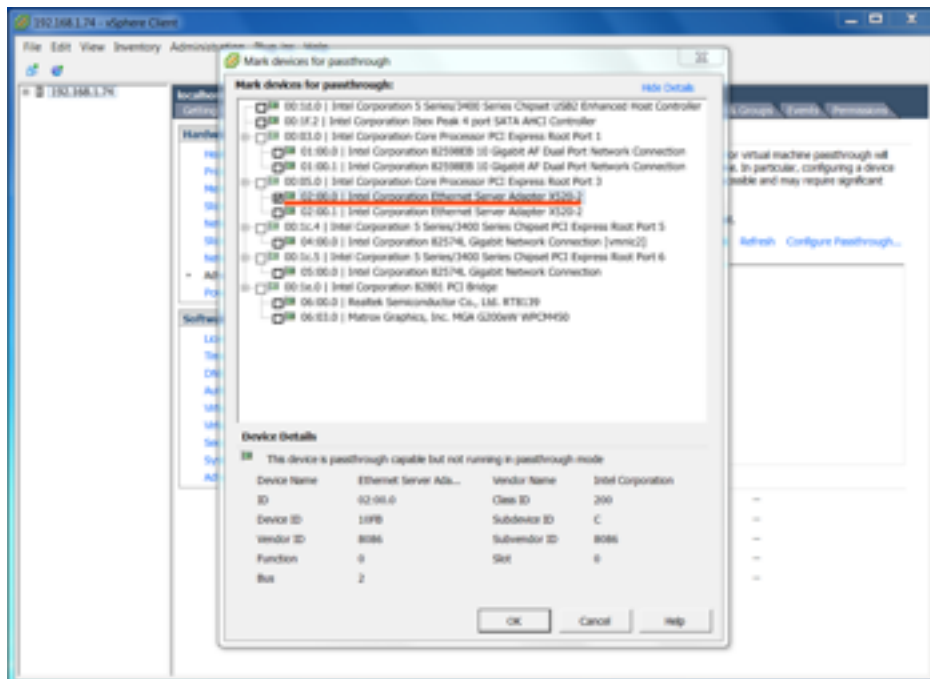


## 12.2.VMware ESX Configuration

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".

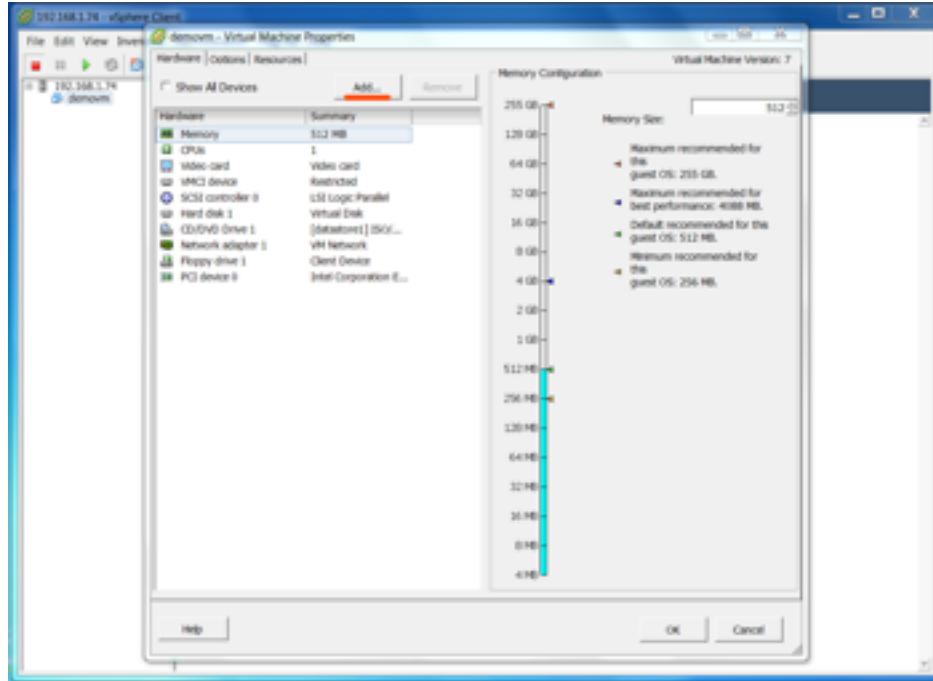


Select the devices you want to assign to the VMs.

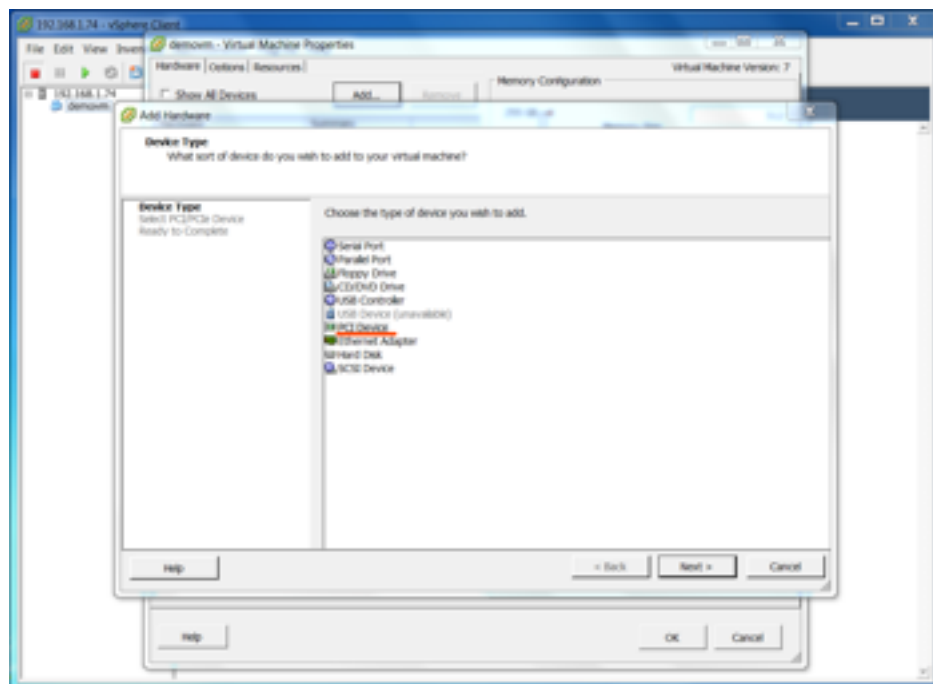


Reboot the server.

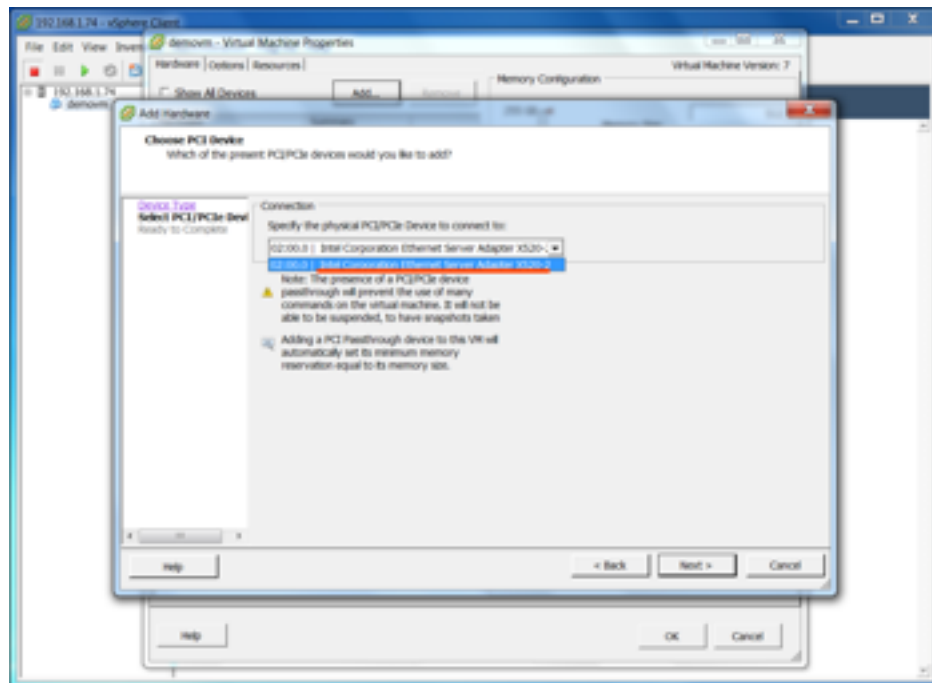
After the reboot, make sure that the VMs where the PCI device will be assigned is in the off state. Open the VM settings, and click on "Add..." in the "Hardware" tab.



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

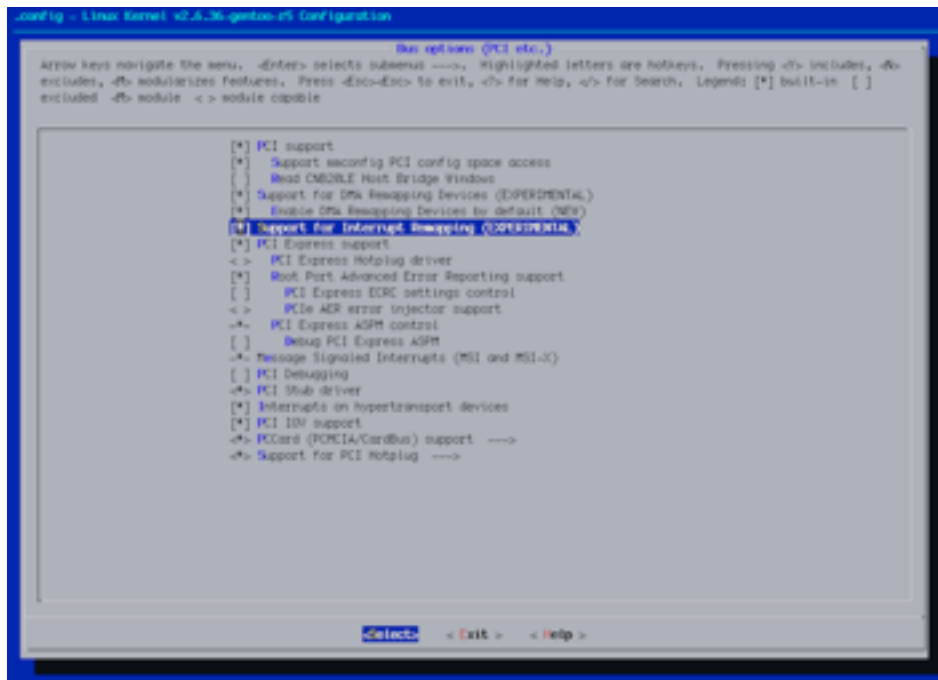
[\*] Enable DMA Remapping Devices

[\*] 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.