

# PF\_RING User Guide

Linux High Speed Packet Capture

Version 6.7.0

April 2017

© 2004-17 ntop.org

# 1. Table of Contents

1. Table of Contents .....	2
2. Introduction .....	4
2.1. What's New? .....	4
3. Welcome to PF_RING .....	6
3.1. Packet Filtering .....	6
3.2. Packet Journey .....	7
3.3. Packet Clustering .....	7
4. PF_RING Drivers .....	9
4.1. PF_RING ZC .....	9
5. PF_RING Installation .....	11
5.1. Linux Kernel Module Installation .....	11
6. Running PF_RING .....	12
6.1. ZC Drivers .....	12
6.2. Configuring a PF_RING Deb/RPM package .....	13
6.3. Checking PF_RING Device Configuration .....	13
6.4. Libpfiring and Libpcap Installation .....	14
6.5. Application Examples .....	14
6.6. PF_RING Additional Modules .....	15
6.7. PF_RING ntopdump: A Wireshark Extcap Compatible Tool .....	16
7. PF_RING for Application Developers .....	19
7.1. The PF_RING API .....	19
7.2. Return Codes .....	19
7.3. PF_RING Device Name Convention .....	19
7.4. PF_RING Packet Reflection .....	20
7.5. PF_RING Packet Filtering .....	20

- 7.6. PF\_RING In-NIC Packet Filtering.....20
- 8. PF\_RING ZC Device Drivers On Virtual Machines .....21
  - 8.1. BIOS Configuration .....21
  - 8.2.VMware ESX Configuration .....22
  - 8.3. KVM Configuration .....25

## 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?

PF\_RING 6.6 Changelog:

- ▶ PF\_RING Library
  - New `pfring_findalldevs/pfring_freealldevs` API for listing all interfaces supported by `pf_ring`
  - New timeline module based on `libnpcap` for seamlessly extracting traffic from a `n2disk` dumpset using the `pf_ring` API
  - Dynamic capture modules loading with `dlopen` support
  - Improved `pfring_set_bpf_filter` to set hw rules when supported by the network card thanks to the nBPF engine
- ▶ ZC Library
  - New `pfring_zc_set_bpf_filter/pfring_zc_remove_bpf_filter` API for setting BPF filters to device queues
  - Fixed `pfring_zc_queue_is_full` for device queues
  - Flushing SPSC queues when a consumer attaches (RX only)
- ▶ PF\_RING-aware Libpcap/Tcpdump
  - Support for extracting traffic from a `n2disk` dumpset using `libpcap`
  - `tcpdump` upgrade to v.4.9.0
- ▶ PF\_RING kernel module
  - Support for latest ubuntu and centos stable kernels
  - Support for SCTP and ICMP packet parsing
  - Packet hash improvements
  - Added tunneled IP version to packet metadata
  - Added IP version to sw filters
  - New kernel cluster hash types for tunneled traffic
  - QinQ VLAN parsing
  - Removed deprecated kernel plugins support
  - Promisc fix in case of multiple devices in a single socket
- ▶ Drivers
  - Support for latest ubuntu and centos stable kernels
  - FPGA modules/libraries are now loaded at runtime using `dlopen`
  - RSS support on Intel i211

- Jumbo frames support on i40e
- i40e tx optimisations
- i40e interrupts fixes in case of RSS
- Fiberblaze capture module with chunk mode support
- Exablaze capture module
- Accolade improvements
- Endace DAG update and support for streams
- Myricom ports aggregation fixes, new syntax myri:<port>,<port>

► nBPF

- New nBPF filtering engine supporting an extended subset of the BPF syntax (tunneled traffic and I7 protocols are supported)
- nBPF support for hw filtering on Fiberblaze cards
- nBPF support for hw filtering on Intel FM10K cards (Silicom PE3100G2DQIR)
- nBPF support for hw filtering on Exablaze cards
- nBPF support for hw filtering on Napatech cards and NTPL generation
- Support for "start <time> and end <time> and <bpf>" when extracting from a n2disk timeline
- Support for vlan [id], mpls [label], gtp

► Examples

- pfcount:
  - ability to list interfaces with -L (-v 1 for more info)
  - ability to dump traffic on PCAP file with -o
- psend:
  - option to force flush per packet (-F)
  - options to specify src/dst IP for packet forging (-S/-D)
  - option to forge packets on the fly instead of at preprocessing time (-O)
  - option to randomize generated ips sequence (-z)
  - ability to generate IPv6 traffic (-V 6)
  - ability to generate mixed v4 and v6 traffic (-V 0)
  - TCP/UDP checksum when reforging
- zbalance\_ipc
  - option to use hw aggregation when supported by the card (-w)
  - IP-based filtering with ZMQ support for rules injection

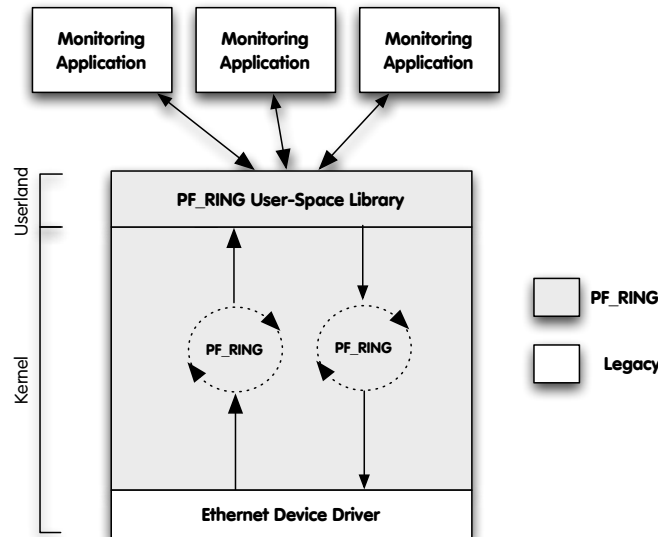
► Wireshark

- New extcap module 'ntopdump' for Wireshark 2.x

► Misc

- Improved systemd support (Ubuntu 16)

### 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.
- Specialised 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. Please note that PF\_RING can operate with any NIC driver, but for maximum performance it is necessary to use these specialised drivers that can be found into the drivers/ directory part of the PF\_RING distribution.

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

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

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

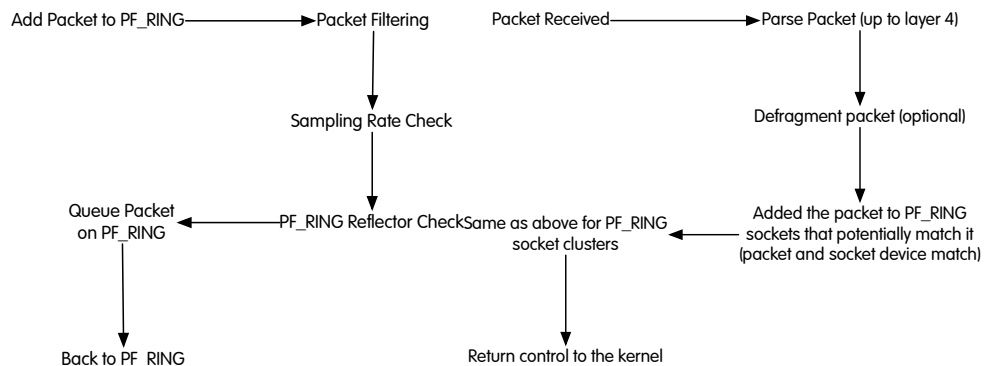
#### 3.1. Packet Filtering

PF\_RING supports both legacy BPF filters (i.e. those supported by pcap-based applications such as tcpdump), and also two additional types of filters (named wildcard and precise filters, depending on the fact that some or all filter elements are specified) that provide developers a wide choice of options. Filters are evaluated inside the PF\_RING module thus in kernel. Some modern adapters such as Intel 82599-based or Silicom Redirector NICs, support hardware-based filters that are also supported by PF\_RING via specified API calls (e.g. `pfring_add_hw_rule`). PF\_RING filters (except hw filters) can have an action

specified, for telling to the PF\_RING kernel module what action needs to be performed when a given packet matches the filter. Actions include pass/don't pass the filter to the user space application, stop evaluating the filter chain, or reflect packet. In PF\_RING, packet reflection is the ability to transmit (unmodified) the packet matching the filter onto a network interface (this except the interface on which the packet has been received). The whole reflection functionality is implemented inside the PF\_RING kernel module, and the only activity requested to the user-space application is the filter specification without any further packet processing.

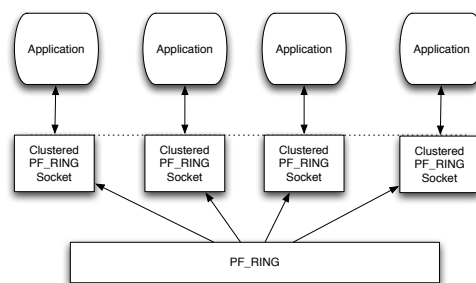
### 3.2. Packet Journey

The packet journey in PF\_RING is quite long before being queued into a PF\_RING ring.



### 3.3. Packet Clustering

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



The way packets are partitioned across cluster sockets is specified in the cluster policy that can be either per-flow (i.e. all the packets belonging to the same tuple <proto, ip src/dst, port src/dst>) that is the default or round-robin. This means that if you select per-flow balancing, all the packets belonging to the same flow (i.e. the 5-tuple specified above) will go to the same application, whereas with round-robin all the apps will receive the same amount of packets but there is no guarantee that packets belonging to the same queue will be received by a single application. So in one hand per-flow balancing allows you to preserve the application logic as in this case the application will receive a subset of all packets but this traffic will be consistent. On the other hand if you have a specific flow that takes most of the traffic, then

the application that will handle such flow will be over-flooded by packets and thus the traffic will not be heavily balanced.

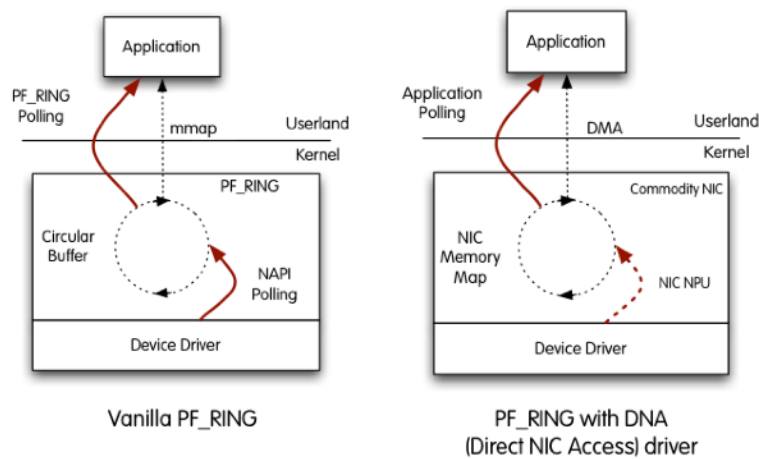


## 4. PF\_RING Drivers

As previously stated, PF\_RING can work both on top of standard NIC drivers, or on top of specialised 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 ZC

For those users who need maximum packet capture speed with 0% CPU utilisation for copying packets to the host (i.e. the NAPI polling mechanism is not used) it is also possible to use ZC (aka new generation DNA) drivers, 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 ZC 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.

These drivers, available in `PF_RING/drivers/`, are standard drivers with support for the PF\_RING ZC library. They can be used as standard kernel drivers or in zero-copy kernel-bypass mode (using the PF\_RING ZC library) adding the prefix "zc:" to the interface name.

Once installed, the drivers operate as standard Linux drivers where you can do normal networking (e.g. ping or SSH). If you open a device in zero copy (e.g. `pfcount -i zc:eth1`) the device becomes unavailable to standard networking as it is accessed in zero-copy through kernel bypass, as happened with the predecessor DNA. Once the application accessing the device is closed, standard networking activities can take place again. An interface in ZC mode provides the same performance as DNA.

PF\_RING ZC (Zero Copy) is a flexible packet processing framework that allows you to achieve 1/10 Gbit line-rate packet processing (both RX and TX) at any packet size. It implements zero-copy operations including patterns for inter-process and inter-VM (KVM) communications. It can be considered as the successor of DNA/LibZero that offers a single and consistent API implementing simple building blocks (queue, worker and pool) that can be used from threads, applications and virtual machines.

The following example shows how to create an aggregator+balancer application in 6 lines of code.

```

1  zc = pfiring_zc_create_cluster(ID, MTU, MAX_BUFFERS, NULL);
2  for (i = 0; i < num_devices; i++)
3    inzq[i] = pfiring_zc_open_device(zc, devices[i], rx_only);
4  for (i = 0; i < num_slaves; i++)
5    outzq[i] = pfiring_zc_create_queue(zc, QUEUE_LEN);
6  zw = pfiring_zc_run_balancer(inzq, outzq, num_devices,
    num_slaves, NULL, NULL, !wait_for_packet, core_id);

```

PF\_RING ZC driver, once installed, operate as standard Linux drivers where you can do normal networking (e.g. ping or SSH). If you open a device in zero copy (e.g. `pfcount -i zc:eth1`) the device becomes unavailable to standard networking as it is accessed in zero-copy through kernel bypass, as happened with the predecessor DNA. Once the application accessing the device is closed, standard networking activities can take place again.

PF\_RING ZC allows you to forward (both RX and TX) packets in zero-copy for a KVM Virtual Machine without using techniques such as PCIe passthrough. Thanks to the dynamic creation of ZC devices on VMs, you can capture/send traffic in zero-copy from your VM without having to patch the KVM code, or start KVM after your ZC devices have been created. In essence now you can do 10 Gbit line rate to your KVM using the same command you would use on a physical host, without changing a single line of code.

In PF\_RING ZC you can use the zero-copy framework even with non-PF\_RING aware drivers. This means that you can dispatch, process, originate, and inject packets into the zero-copy framework even though they have not been originated from ZC devices. Once the packet has been copied (one-copy) to the ZC world, from then onwards the packet will always be processed in zero-copy during all his lifetime. For instance the `zbalance_ipc` demo application can read packet in 1-copy mode from a non-PF\_RING aware device (e.g. a WiFi-device or a Broadcom NIC) and send them inside ZC for performing zero-copy operations with them.

## 5. PF\_RING Installation

PF\_RING can be downloaded in source format from GIT at [https://github.com/ntop/PF\\_RING/](https://github.com/ntop/PF_RING/) or installed from packages using Ubuntu/CentOS repositories at <http://packages.ntop.org> as explained in the "Configuring a PF\_RING Deb/RPM package" section below.

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.

The PF\_RING source code layout is the following:

- Changelog
- LICENSE
- Makefile
- README.FIRST
- doc/
- drivers/
- kernel/
- package/
- userland/

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

### 5.1. Linux Kernel Module Installation

If you choose to install from package please read the section "Configuring a PF\_RING Deb/RPM package" below and skip this section.

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
$ make install
```

Note that:

- the kernel module installation (via make install) requires root capabilities.
- As of PF\_RING 4.x you NO LONGER NEED to patch the linux kernel as in previous PF\_RING versions.

## 6. Running PF\_RING

If you installed from package please read the section “Configuring a PF\_RING Deb/RPM package” below and skip this section.

Before using any PF\_RING application the `pf_ring` kernel module should be loaded (as superuser):

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

Where:

- `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.

Example:

```
$ cd <PF_RING PATH>/kernel
$ insmod pf_ring.ko min_num_slots=8192 enable_tx_capture=0 quick_mode=1
```

### 6.1. ZC Drivers

If you want to achieve line-rate packet capture even at 10 Gigabit, you should use these drivers. ZC drivers are part of the PF\_RING distribution and can be found in “<PF\_RING PATH>/drivers/”.

Currently available ZC drivers are:

- `e1000e`
- `igb`
- `ixgbe`
- `i40e`
- `fm10k`

Please note that:

- the PF\_RING kernel module must be loaded before the ZC driver
- in order to correctly configure the device, it is highly recommended to use the `load_driver.sh` script provided with the drivers (take a look at the script to fine-tune the configuration)
- ZC drivers need hugepages, the `load_driver.sh` script takes care of hugepages configuration

If you installed from package please read the section “Configuring a PF\_RING Deb/RPM package” below and skip this section.

Example loading PF\_RING and the ixgbe-ZC driver:

```
$ cd <PF_RING_PATH>/kernel
$ insmod pf_ring.ko
$ cd PF_RING/drivers/intel/ixgbe/ixgbe-X.X.X-zc/src
$ make
$ ./load_driver.sh
```

## 6.2. Configuring a PF\_RING Deb/RPM package

In addition to source code it is possible to install PF\_RING using the installation packages provided at <http://packages.ntop.org/>.

Once the “pfring” package, and optionally the ZC drivers, is installed following the procedure on the web page, it is possible to use the init script under /etc/init.d/pf\_ring to automate the kernel module and drivers loading. The init script acts as follows:

1. loads the pf\_ring.ko kernel module.
2. scans the folders /etc/pf\_ring/zc/{e1000e,igb,ixgbe,i40e,fm10k}/ searching files:
  - {e1000e,igb,ixgbe,i40e,fm10k}.conf containinig the driver parameters
  - {e1000e,igb,ixgbe,i40e,fm10k}.start that should be just an empty file
3. loads the drivers whose corresponding {e1000e,igb,ixgbe,i40e,fm10k}.start file is present, unloading the vanilla driver.
4. configures hugepages if a ZC driver has been loaded, reading the configuration from /etc/pf\_ring/hugepages. Each line (one per CPU) of the configuration file should contain:

```
node=<NUMA node id> hugepagenumber=<number of pages>
```

Example of a minimal configuration for a dual-port ixgbe card on a uniprocessor:

```
$ mkdir -p /etc/pf_ring/zc/ixgbe
$ echo "RSS=1,1" > /etc/pf_ring/zc/ixgbe/ixgbe.conf
$ touch /etc/pf_ring/zc/ixgbe/ixgbe.start
$ echo "node=0 hugepagenumber=1024" > /etc/pf_ring/hugepages
```

In order to run the init script, after all the files have been configured:

```
$ /etc/init.d/pf_ring start
```

## 6.3. 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  stats/

# cat /proc/net/pf_ring/info
PF_RING Version      : 6.4.1
```

```
Total rings                : 0

Standard (non ZC) Options
Ring slots                  : 4096
Slot version                : 16
Capture TX                  : Yes [RX+TX]
IP Defragment               : No
Socket Mode                 : Standard
Total plugins               : 0
Cluster Fragment Queue      : 0
Cluster Fragment Discard    : 0
```

## 6.4. 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.

## 6.5. 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      pfcountr_82599.c    pflatency.c  pfwrite.c
pcap2nspcap.c pfcountr.c          pfsend.c     preflect.c
pcount.c       pfcountr_multichannel.c  pfsystest.c
pfbridge.c     pfdump.c            pfutils.c
$ make
```

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

```
# ./pfcount -i zc:eth1
...
=====
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 zc:eth1 -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]
```

## 6.6. 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.
- ZC module.  
This module can be used to open a device in ZC mode, if you own a supported card and a PF\_RING-aware driver with ZC support. ZC dramatically increases the packet capture and transmission speed as the kernel layer is bypassed and applications can communicate directly with the card.  
Currently these ZC drivers are available:
  - e1000e
  - igb
  - ixgbe
  - i40e
  - fm10k

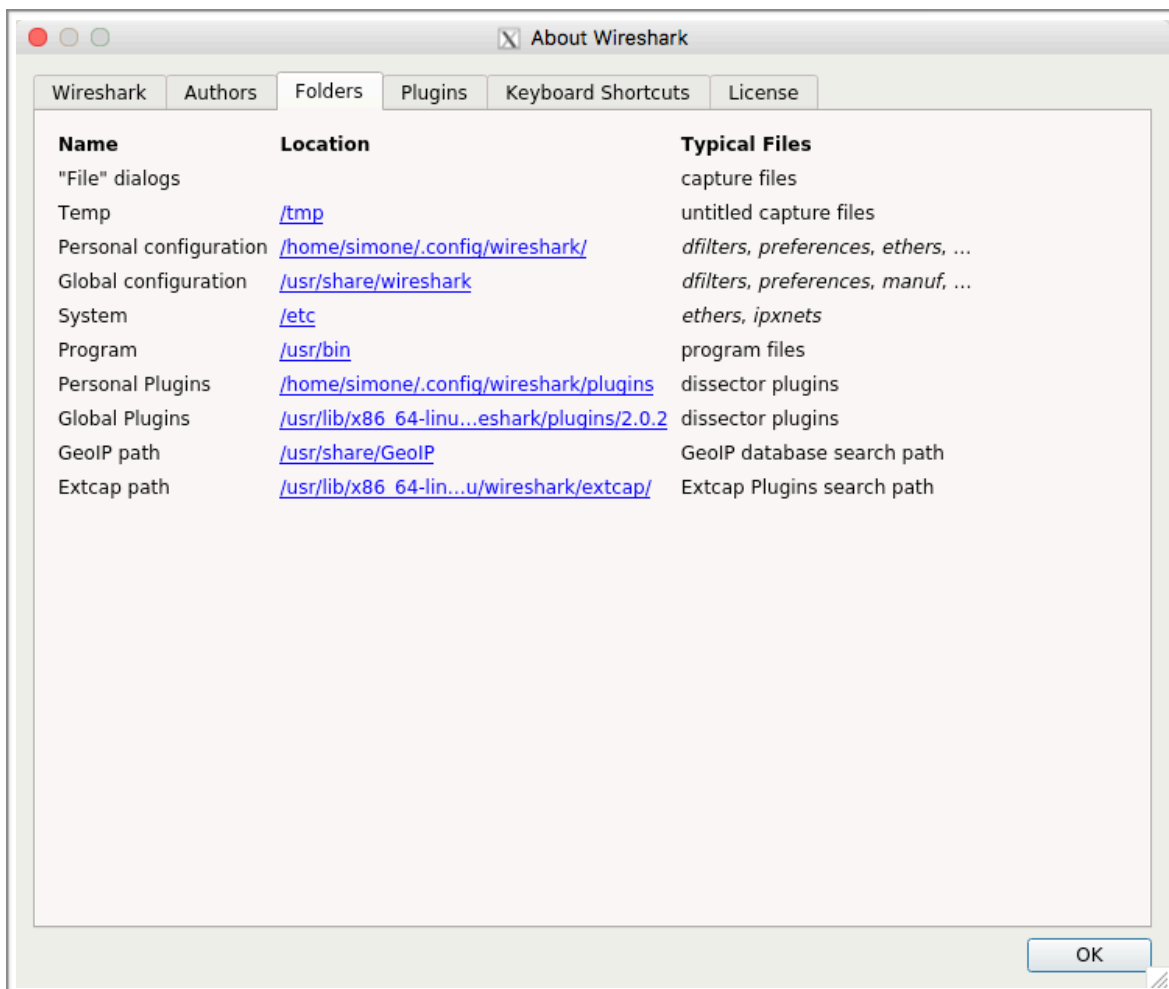
The drivers are part of the PF\_RING distribution and can be found in `drivers/` identified by the suffix `'-zc'`. With all the drivers you can achieve wire rate at any packet size, both for RX and TX. In order to open a device in ZC mode you should use the `"zc:"` prefix: `"zc:ethX"`.

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`).

- 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".
- 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:ethX" where ethX 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_recv()` has to be used.

## 6.7. PF\_RING ntopdump: A Wireshark Extcap Compatible Tool

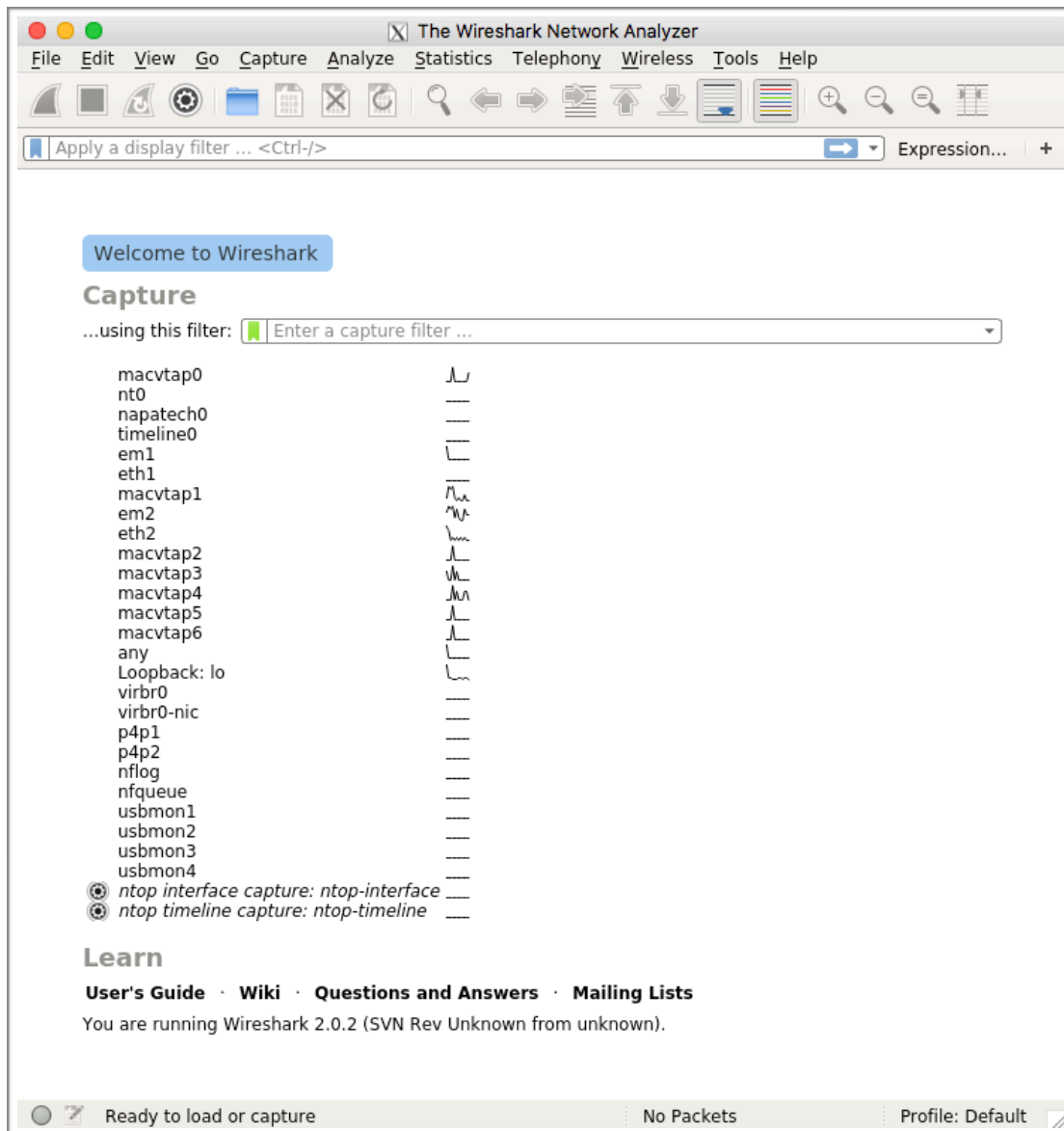
PF\_RING provides `ntopdump`, a tool that implements the Wireshark extcap interface. The extcap interface is a Wireshark plugin interface that allows external binaries to act as capture interfaces directly in Wireshark. By implementing the extcap interface, `ntopdump` is able capture from a PF\_RING interface, delivering the captured packets directly to Wireshark.



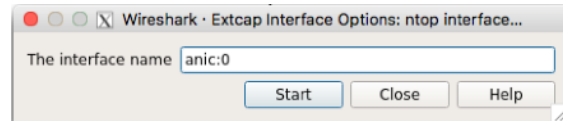


In order for Wireshark to use it, `ntopdump` must be placed under the Wireshark extcap lookup folder. The path of that folder can be determined by selecting "Folders" tab of the Wireshark "Help" -> "About" menu. Packaged versions of PF\_RING will automatically place `ntopdump` executable in the right place. Users who build PF\_RING from sources should manually place it in the extract lookup folder.

Starting Wireshark with the `ntopdump` binary in place, will yield to new entries in the interfaces list, namely `ntopdump-interface` and `ntopdump-timeline`. The former is the actual entry that has to be selected in order to capture from PF\_RING interfaces. The latter is used to read pcap files from the traffic recorder `n2disk` and falls outside the scope of this manual.



A small wheel on the left of the entry name allows to specify the capture settings, including the interface name (e.g., `anic:0`) and the BPF filters that PF\_RING will try to inject as deep as possible, possibly reaching the hardware (BPF filters input field is only available in recent Wireshark versions).



Once the interface name has been specified, a double click on the entry will open the familiar Wireshark packets window and, if everything has been set up properly, packets should start flowing on that window.

## 7. 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.

### 7.1. 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.

### 7.2. 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.

### 7.3. 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 (e.g. an Intel 10 Gbit adapter), PF\_RING allows you to capture packet from a specific RX 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 since version 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. `zc:ethX@Y` for the ZC module, `dag:dagX:Y` for the dag module, `"multi:ethA@X;ethB@Y;ethC@Z"` for the Link Aggregation module).

## 7.4. 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.

## 7.5. 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.

## 7.6. 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 `queued`, such packet is discarded thus acting as a hardware firewall filter.

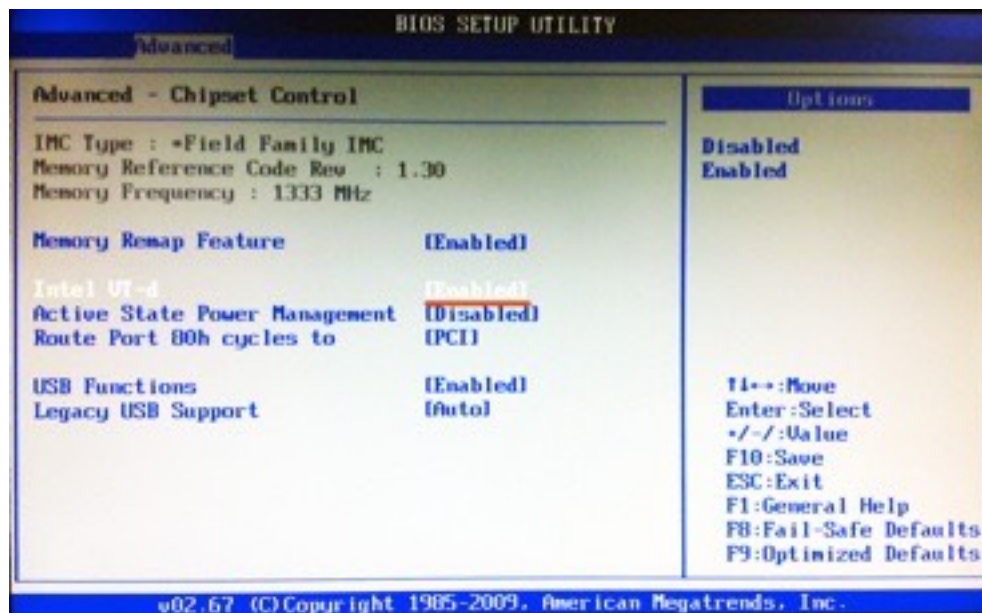
NOTE: Kernel packet filtering is not supported by ZC.

## 8. PF\_RING ZC Device Drivers On Virtual Machines

Section 4 contains a brief introduction to the PF\_RING ZC drivers, which allows you to manipulate packets at 10 Gbit wire speed for any packet size. Thanks to Virtualisation Technologies based on IOMMUs (Intel VT-d or AMD IOMMU), it is possible to assign a device to a given guest operating system, benefiting from the PF\_RING ZC drivers acceleration within a VM (Virtual Machine). The following sections show how to configure VMware and KVM (the Linux-native virtualisation system). XEN users can use similar system configurations.

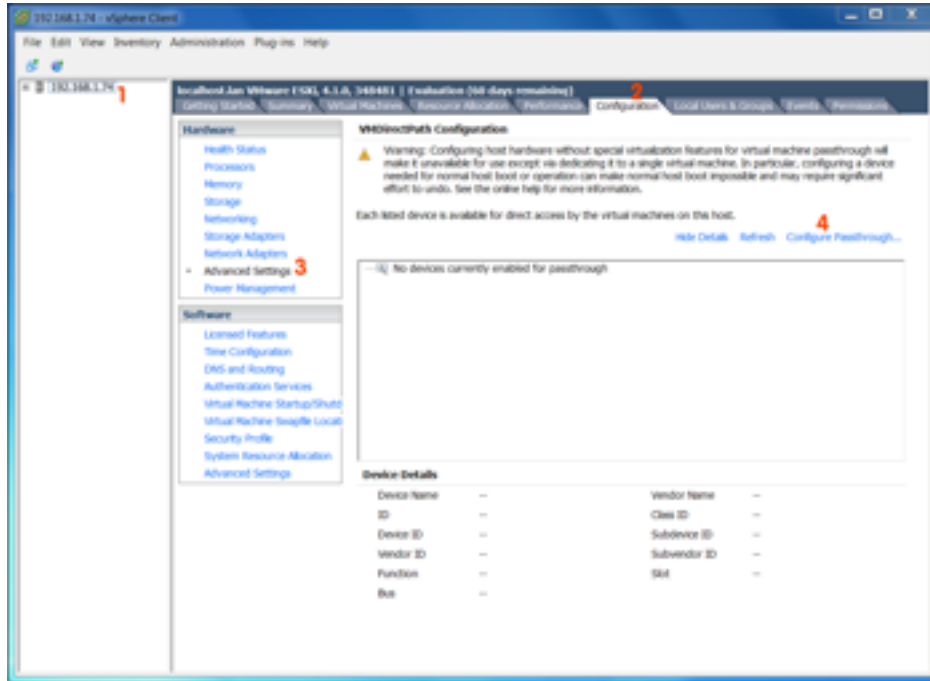
### 8.1. BIOS Configuration

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

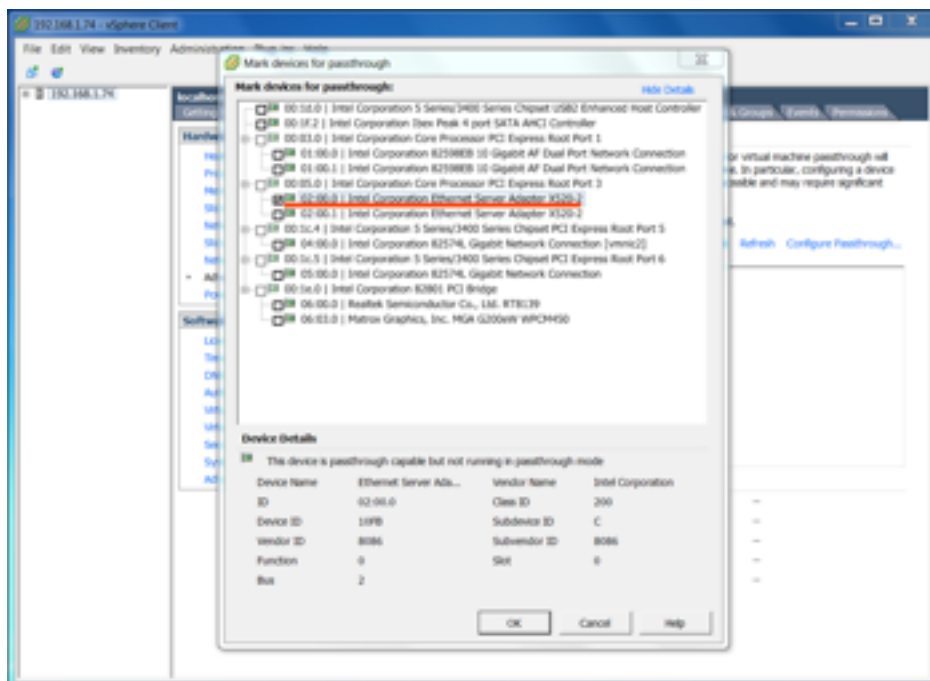


## 8.2.VMware ESX Configuration

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".

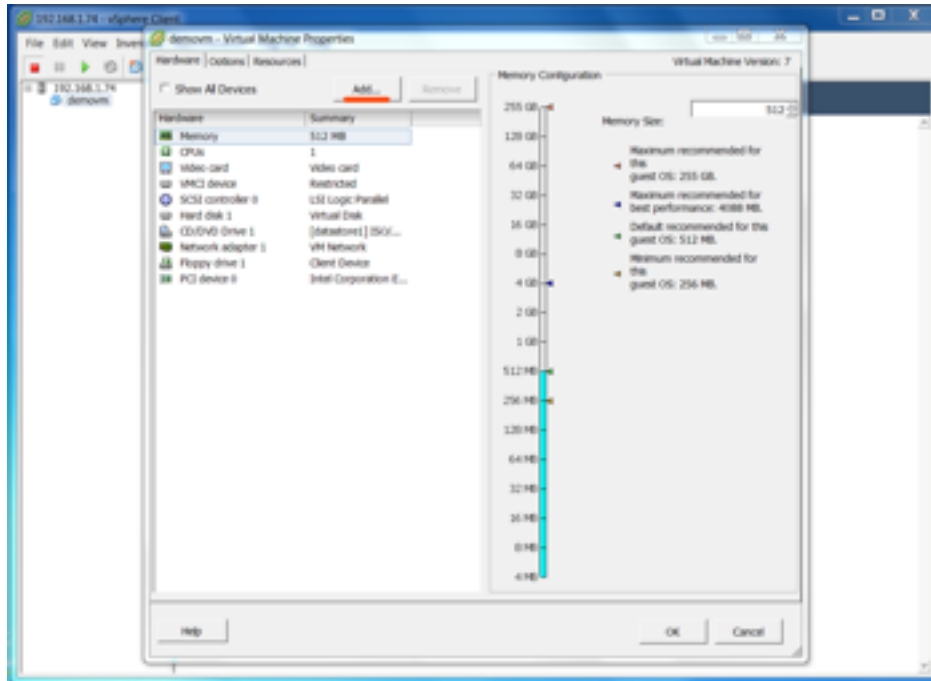


Select the devices you want to assign to the VMs.

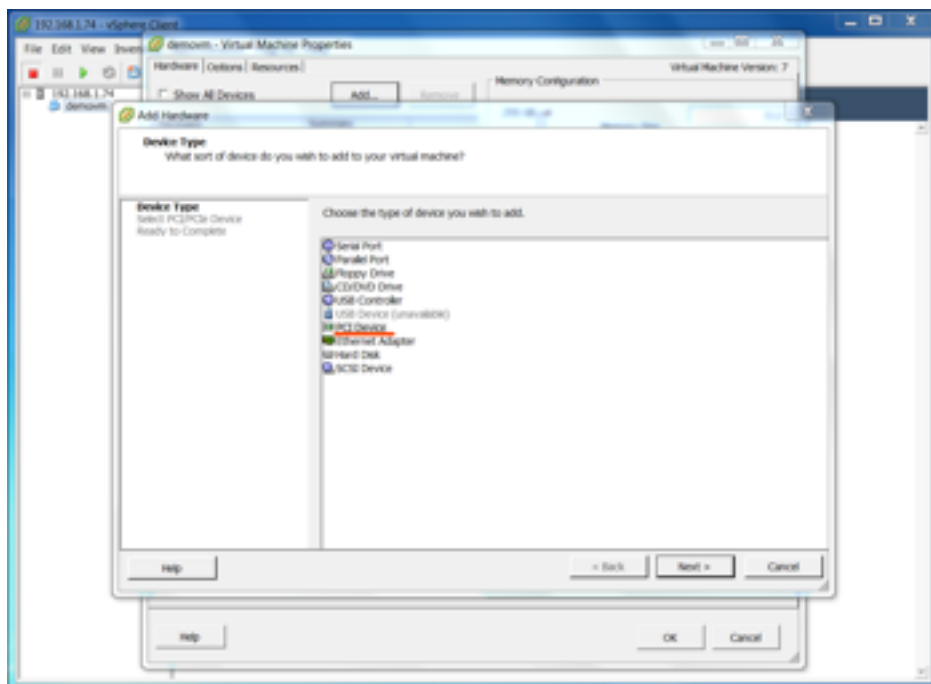


Reboot the server.

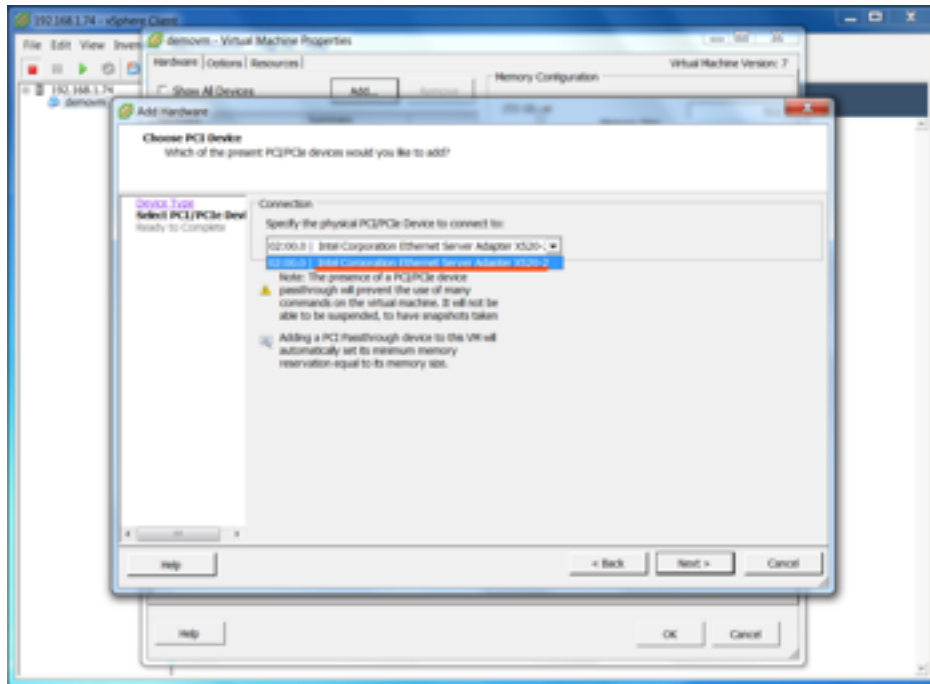
After the reboot, make sure that the VMs where the PCI device will be assigned is in the off state. Open the VM settings, and click on "Add..." in the "Hardware" tab.



Select "PCI Device".



Select the device to assign to the VM.



Boot the VM and install PF\_RING with the ZC driver as in the native case.



### 8.3. 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

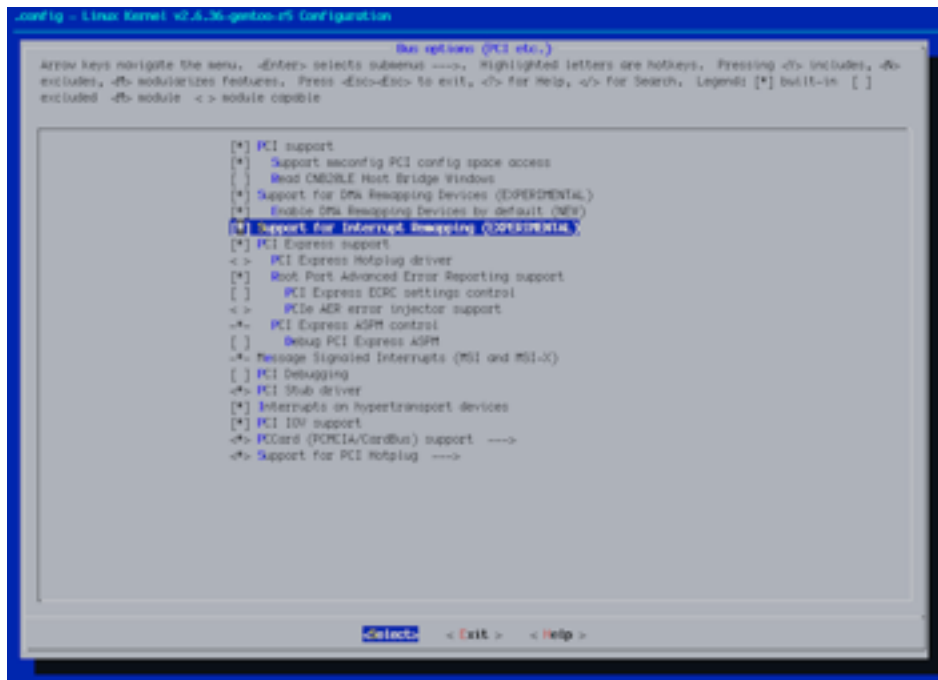
[\*] Enable DMA Remapping Devices

[\*] 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 ZC driver as in the native case.