PF_RING 用户指南

Linux 高速包捕获

版本 5.4.4

2012 年 7 月

©2004-12 ntop.org

目　　录

1. 介绍

PF_RING 是一个高速数据包捕获库，它把通用 PC 计算机变成一个有效且便宜的网络测量工具箱，适用于数据包和现网流量的分析和操作。并且，PF_RING 完整的开发一个市场，它可以用来创建更有效的应用程序，例如：使用一些代码可以实现流量均衡或者数据包过滤。

本手册分为两个部分：

1） PF_RING 安装和配置；

2） PF_RING SDK（软件开发工具箱）。

1.1. PF_RING 用户指南的更新列表

- 5.4.0 版本发布（2012 年 5 月）
  更新用户指南到 PF_RING 5.4.0 版本
  在 DNA 基础上增加灵活的零拷贝数据包处理库（libzero）

- 5.3.1 版本发布（2012 年 3 月）
  更新用户指南到 PF_RING 5.3.1 版本

- 5.2.1 版本发布（2012 年 1 月）
  更新用户指南到 PF_RING 5.2.1 版本
  增加管理硬件时钟和时间戳的新 API 函数
  增加内核插件调用

- 5.1 版本发布（2011 年 9 月）
  更新用户指南到 PF_RING 5.1.0 版本

- 4.7.1 版本发布（2011 年 7 月）
  更新用户指南到 PF_RING 4.7.1 版本
  描述 PF_RING 模块库和一些模块（DAG，DNA）

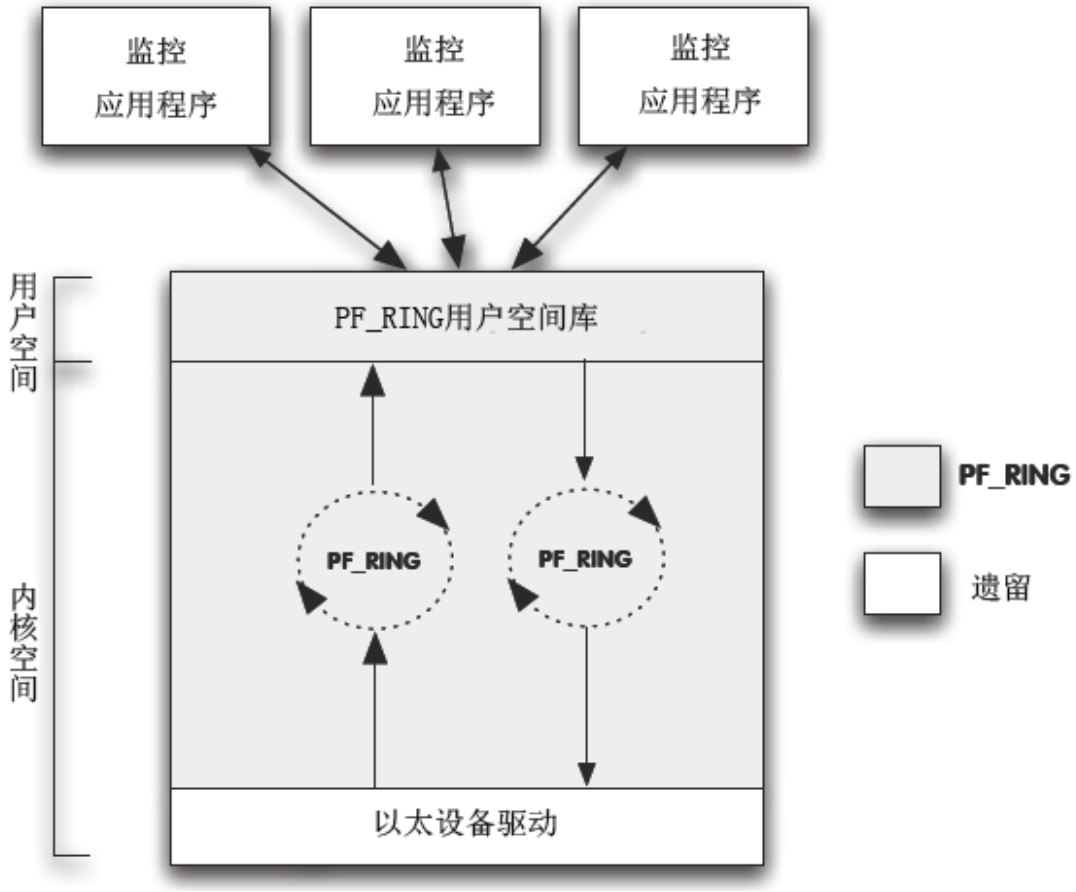
- 4.6.1 版本发布（2011 年 3 月）
  更新用户指南到 PF_RING 4.6.1 版本

- 4.6 版本发布（2011 年 2 月）
  更新用户指南到 PF_RING 4.6.0 版本

- 1.1 版本发布（2008 年 1 月）
  描述 PF_RING 插件架构

- 1.0 版本发布（2008 年 1 月）
  开始编写 PF_RING 用户指南

2. 欢迎进入 PF_RING



在上图中描述了 PF_RING 的架构，说明如下。

主要的结构单元有：

1） 加速的内核模块，提供低级数据包拷贝到 PF_RING 的环中；

2） 用户空间 PF_RING SDK 提供传输 PF_RING 支持的数据到用户空间的应用程序的功能；

3） 专用的 PF_RING 支持的驱动（可选）允许更进一步的加速数据包捕获，在不通过内核数据结构的情况下，更有效地把数据包从驱动拷贝到 PF_RING 中。请注意 PF_RING 可以使用任何的 NIC 驱动，但是必须使用这些专用的驱动以便获得最大化的性能，它们可以在 PF_RING 发行包的 kernel 目录中找到。注意，在使用 transparent_mode 参数来加载 PF_RING 内核模块时，可以选择传递数据包到 PF_RING 的驱动方式。

PF_RING 实现一种新的 socket 类型（称为 PF_RING），这样用户空间的应用程序可以与 PF_RING 内核模块进行通讯，应用程序获得 PF_RING 的句柄，然后调用本手册后面描述的 API。一个句柄可以绑定到：

1） 物理网络接口；

2） 一个 RX 队列，只是用于多队列网卡适配器；

3） 任何的虚拟接口，这意味着可以在使用的系统接口上接收/发送数据包。

正如上面描述的，你可以从一个创建时分配的内存 ring 中读取数据包。产生的数据包从内核模块中拷贝到环（ring）中，并且用户空间的应用程序可以读取它。不必执行每个数据包的内存分配和释放。一旦数据包从 ring 中读取出来，ring 中用来存储数据包的空间将会分配给后续的数据包使用。因此，应用程序需要保存一个数据包的备份，必须保留这些刚才

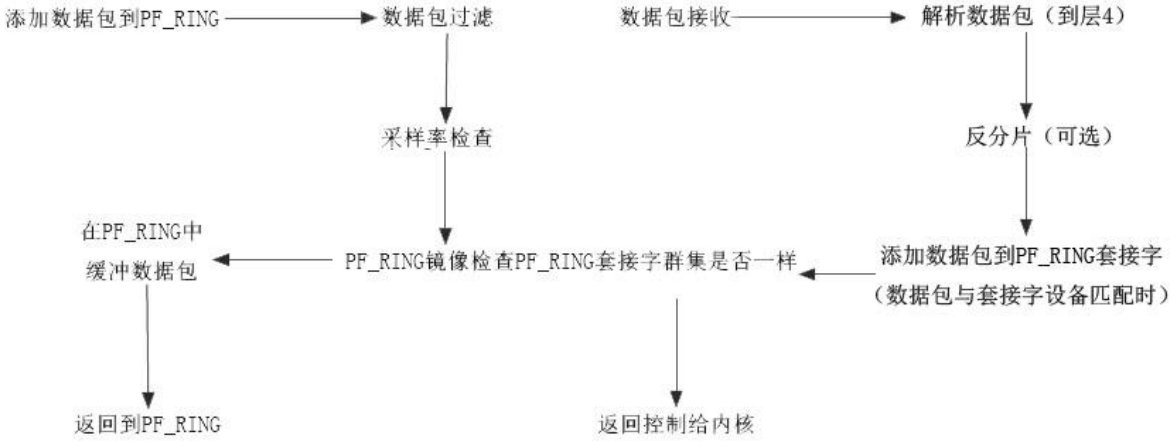
读取的数据包，因为 PF_RING 自身不会保留它们。

2.1. 数据包过滤

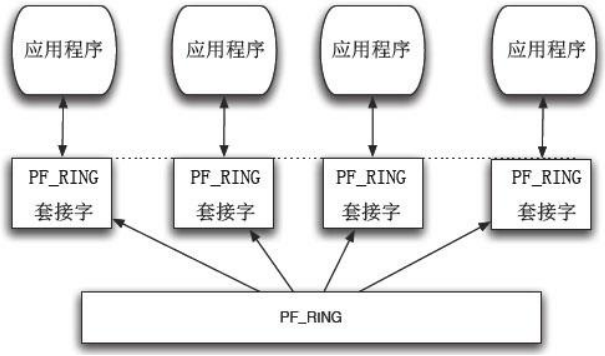
PF_RING 支持遗留（以前）的 BPF 过滤器（例如：基于 pcap 应用程序（如：tcpdump）支持的），同时也支持两种其他的过滤器类型（称为通配符和精准过滤器，这依赖于指定一些或者全部的过滤元素），这提供给开发者更多的选择。过滤器在内核中的 PF_RING 模块中进行评估。一些最新的适配器，例如：基于 Intel 82599 的时适配器，或 Silicom 重定向 NIC，PF_RING 使用专用的 API 调用也支持基于硬件过滤器（例如：pfring_add_hw_rule）。PF_RING 过滤器（除硬件过滤器以外）把一个用户指定的操作告诉 PF_RING 内核模块，在给定的数据包匹配上时过滤器时执行什么操作，操作包括通过或者不通过过滤器送到用户空间应用程序中、停止评估过滤器链、镜像数据包等。在 PF_RING 中，数据包镜像可以传输（不修改）匹配上过滤器的数据包到一个网络接口上（不包括数据包接收的接口）。整个镜像功能在 PF_RING 内核模块的内部实现，上报给用户空间应用程序的唯一动作是没有符合任何数据包处理的过滤器描述。

2.2. 数据包流程

在把数据包缓冲到 PF_RING ring 队列之前，PF_RING 中数据包流程相当长，如下图所示。



2.3. 数据包汇聚



PF_RING 也可以通过实现两个机制（称为负载均衡和汇聚）来增加数据包捕获应用程序的性能。这些机制允许应用程序（将并行处理一组数据包）来处理整个数据包流的一部分，此时发送所有的剩余数据包到另外一些 Cluster 上。这意味着不同打开 PF_RING 套接字的应用程序可以把它们绑定到执行的 Cluter 编号（利用 pfring_set_cluster）上从而集中力量来分析数据包的一部分。

通过这种方式，数据包被分配到不同的使用 Cluster 规则（例如：所有属于同一个元组 <proto, ip src/dst, port src/dst>的数据包）指定的 Cluster 套接字上。这意味着，如果你选择 per-flow 均衡，所有属于同一个流的数据包（例如 5 元组指定的）将会进入同一个应用程序，然而带有循环提示器的应用程序将接收到相同数据的数据包，但是这不能保证属于相同队列的数据包由单个应用程序接收。因此，一方面 per-flow 均衡允许你表示应用程序的逻辑，在这种情况下应用程序将接收所有数据包的一个子集，但是这些流量应该是前后一致的。另一方面，如果你有一个带有许多流量的指定流，那么处理这些流的应用程序将会被数据包益出，因此流量将不会大量的均衡。

3. PF_RING 驱动家族

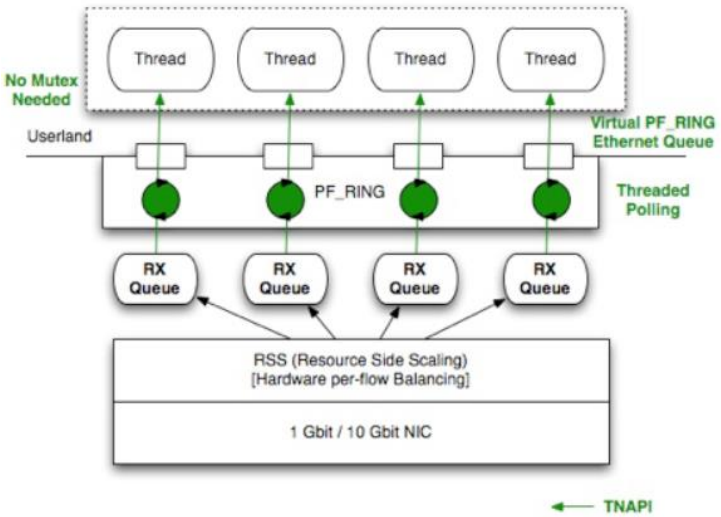
正如前面所描述的，PF_RING 可以工作在标准 NIC 驱动或者专用驱动上面。PF_RING 内核模块是相同的，但是基于使用的驱动程序，在功能和性能上会有所不同。

3.1. PF_RING 支持的驱动

这些驱动（在 PF_RING/driver/PF_RING-aware 中可用）设计用于提高数据包捕获，它把数据包直接放入到 PF_RING 中，不经过标准 Linux 数据包分发机制。使用这些驱动程序，你能使用带有 1 或 2 值的 trasparent_mode。（更多信息请参见文档的下面部分）。

除了 PF_RING 支持的驱动程序（对一些可选的适配器）以外，使用其他驱动程序类型来进一步改善数据包捕获是可能的。

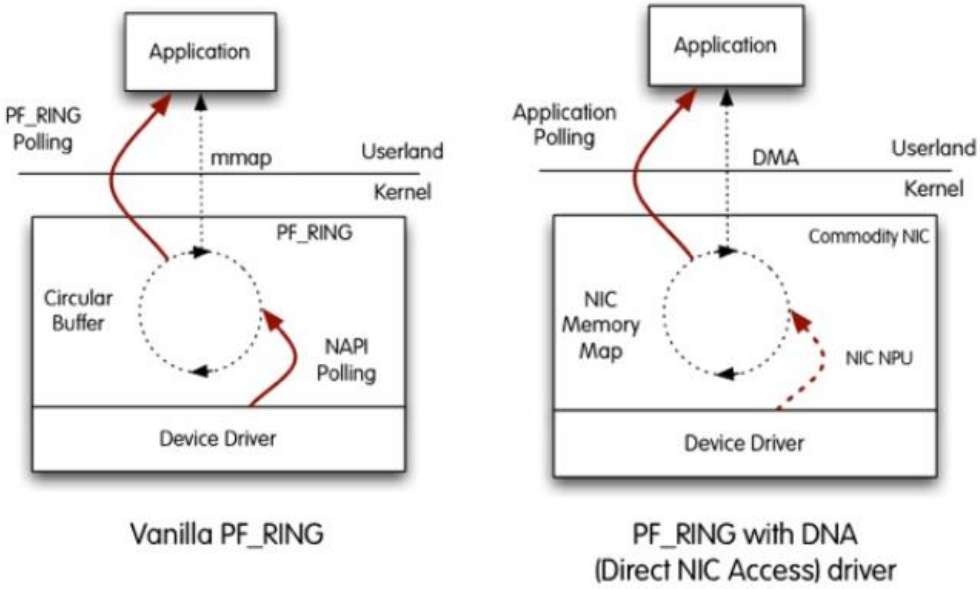
3.2. TNAPI



驱动程序第一个系列称为 TNAPI（线程 NAPI），它们允许数据包使用 TNAPI 驱动直接激活的内核线程从而更加有效的存放到 PF_RING 中。TNAPI 驱动设计用于改进数据包捕获，但是它们不能用于发送数据包，因为 TX 方向上是禁用的。

3.3. DNA

对于那些希望在 CPU 利用率为 0%（拷贝包到主机）情况下需要最大数据包捕获速度的用户（例如：NAPI 轮询机制不能使用）来说，使用不同类型的驱动（称为 DNA）也是可能的，它允许数据直接从网络接口上读取，它以零拷贝的方式同步的从 Linux 内核传递到 PF_RING 模块中。



在 DNA 中同时支持 RX 和 TX 操作。当内核旁路的时候，一些 PF_RING 的功能会失效，它们包括：

1） 内核包过滤（BPF 和 PF_RING 过滤器）；

2） PF_RING 内核插件无效；

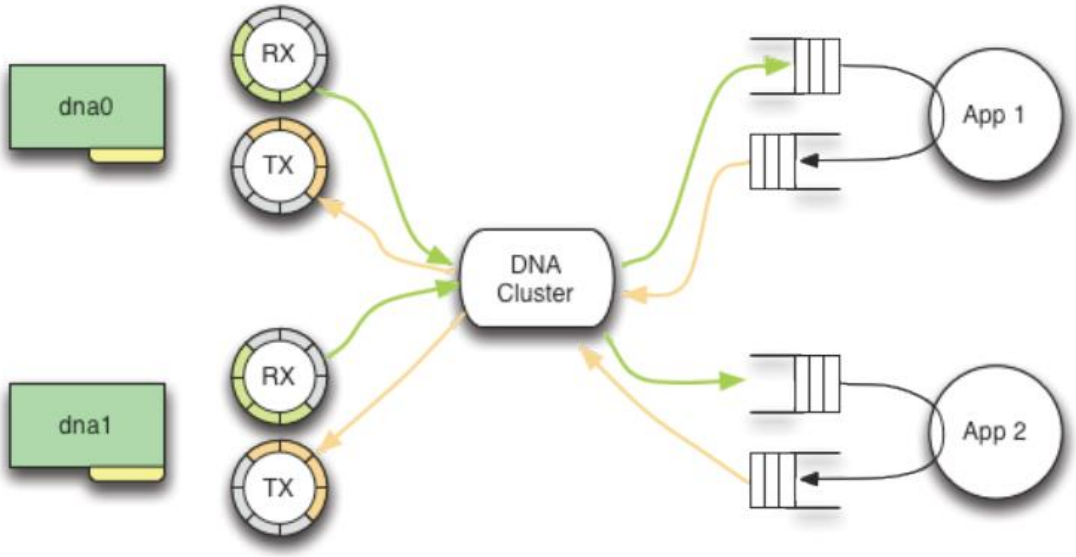
4. DNA 的 Libzero 库

许多应用程序需要复杂的包处理特征，从 PF_RING 5.4.0 开始，称为 libzero 的库被介绍，它位于低级 DNA 接口之上，实现了零拷贝数据包处理。libzero 提供两个主要组件：DNA Cluster 和 DNA Bouncer。

4.1. DNA Cluster

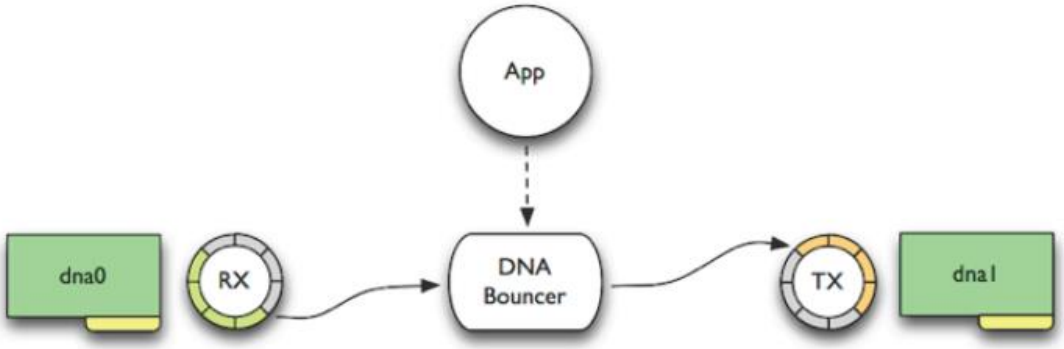
DNA Cluster 实现包的汇聚，因此所有属于相同 Cluster 的应用程序可以共享输入数据包，使用一个简单的负载均衡函数，以零拷贝的方式传输所有的数据包。RSS 的定制实现允许把数据报文分流道网络适配器的多个队列中。Cluster 允许用于定义过滤、分发

和复制数据包的分发函数，可以对应到多个线程和应用程序。



4.2. DNA Bouncer

DNA Bouncer 以零拷贝的方式交换报文到两个接口上，让用户可以指定一个决定函（报文到报文）给定的数据包是否必须传输到接口的函数。



对于一个单向的传递，此时，如果你希望实现桥接的话，两个 Bouncer（每个方向一个）需要被实例化。

5. PF_RING 安装

当你下载 PF_RING 时，你获得下面这些组件：

1） PF_RING 用户空间 SDK；

2） Libpcap 库的加强的版本，安装了 PF_RING 的话，传输能够利用 PF_RING 的优点。如果没有安装 PF_RING，退回到标准的 libpcap 功能。

3） PF_RING 内核模块；

4） 支持各种提供商的不同芯片的 PF_RING 驱动程序。

PF_RING 可以通过 SVN 下载到，网址：http://www.ntop.org/get-started/download/。

PF_RING 源代码的层次结构如下：

doc/
drivers/
kernel/
Makefile
README
README.DNA
README.FIRST
userland/
vPF_RING/
你可以在主目录中输入 make 命令编译整个树（通常，非 root 用户）。

5.1. Linux 内核模块安装

为了编译 PF_RING 内核模块，你需要有一个安装过的 linux 内核头文件（或者内核源代码）。

$ cd <PF_RING PATH>/kernel
$ make

注意：

1） 内核模块安装（利用 make install）要求超级用户（root）权限；
2） 对于一些 Linux 发行版，需要提供一个内核安装和兼容性包；
3） PF_RING 4.x 以后的版本不再需要给 linux 内核打补丁，以前的版本需要。

6. 运行 PF_RING

在使用任何 PF_RING 应用程序之前，pf_ring 内核模块应该以超级用户加载：

# 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]

注意：

1） transparent_mode = 0 (缺省)
   数据包利用标准的 Linux 接口接收，任何驱动可以使用该模式。
2） transparent_mode = 1 （普通和 PF_RING 支持的驱动）
   数据包 memcpy()到 PF_RING 中，也通过标准的 Linux 路径。
3） transparent_mode = 2 (PF_RING 支持的驱动)
   数据包只 memcpy()到 PF_RING 中，不通过标准的 Linux 路径（例如：tcpdump 不能看到如何数据）。

重要：对于普通的驱动，请不要使用 transparent_mode 为 1 和 2，它会造成没有数据包捕获到。

当 transparent_mode 值越高，那么获得数据包捕获就越快。

其他的参数：

1） min_num_slots
   ringslot 的最小数量（缺省— 4096）。
2） enable_tx_capture

设置 1 捕获输出的数据包，设置 0 不捕获输出的数据包，缺省— RX+TX。

3） enable_ip_defrag

设置 1 启用 IP 反分片功能，只有 rx 上的流量会反分片。

4） quick_mode

设置 1 在全速下运行，但是每个接口只能使用一个 socket，不能运用于多个应用程序捕获同一个接口。

6.1. 检查 PF_RING 的设备配置

当 PF_RING 激活时，会创建一个新的入口/proc/net/pf_ring。

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

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

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

PF_RING 允许用户安装插件来处理定制的流量。这些插件也在 pf_ring/proc 树中被注册，它们可以通过输入 plugins_info 文件罗列出来。

6.2. Libpfring 和 Libpcap 安装

Libpfring（用户空间 PF_RING 库）和 libpcap 都以源码的方式发布。它们可以通过下面命令进行编译：

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

注意：库是可重入的，因此它必须链接到你的 PF_RING 使能的应用程序中，也需要添加

-lpthread 库。

重要：遗留的基于 pcap 应用程序需要使用新的 libpcap 库进行重新编译，并且与 PF_RING 使能的 libpcap.a 库链接，这是为了获得 PF_RING 的优势。如果没有重新编译你已有的应用程序，那就不要希望使用 PF_RING。

6.3. 应用程序实例

如果你是 PF_RING 的新手，你可以从一些例子开始。userland/examples 目录中有一些准备使用的 PF_RING 应用程序。

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

例如：pfcout 允许你接收数据包，并且打印出一些统计。

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

另外一个例子是 pfsend，它允许你以给定的速率发送数据包（合成的数据包，或者一个可用的.pcap 文件）：

```
# ./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]
```

6.4. PF_RING 其他模块

从 4.7 版本以后，PF_RING 库中有一个新的模块架构，使得它可以使用其他的组件，而不只是标准的 PF_RING 内核模块。这些组件在库的内部编译，根据配置脚本检查到的情况。

当前，其它模块集合包括：

1） DAG 模块

在 PF_RING 中，该模块增加对 Endace DAG 卡的本地支持。为了使用这个门槛，必须安装 dag 库（4.x 以后版本），并且通过-ldag 符号链接到 PF_RING 使能的应用程序中。

2） DNA 模块

该模块常用于以 DNA 模式打开一个设备，条件是你拥有一个支持的卡和 DNA 驱动。请注意 PF_RING 内核模块必须在 DNA 驱动之前加载。使用 DNA，你可以显著的增加包捕获和传输速率，内核层被旁路，应用程序能直接从驱动程序获得数据报文。

目前这些 DNA 支持的驱动程序可用：e1000e/igb（千兆）/ixgbe（万兆）。

驱动程序是 PF_RING 分布的一个部分，可以在 drivers/DNA 中找到。

使用所有的驱动，你可以完成在任何包大小下 RX 和 TX 方向上线速的捕获。你可以使用 pfcount 应用程序测试 RX，使用 pfsend 应用程序测试 TX。

注意：在 TX 情况下，传输速率受到 RX 性能的限制。这是因为当接收不能跟上捕获的速率时，以太 NIC 发送以太 PAUSE 帧到发送程序来减少速率。如果你希望忽略这些帧，然后以全速发送，你需要关闭掉自动协商并且忽略它们（ethtool -A danX autoneg off rx off tx off）。

3） 链路汇聚（“multi”）模块

该模块可以用于汇聚多个接口，为了从多个打开的单个 PF_RING socket 上捕获数据包。例如，使用设备名称“multi:ethX;ethY;ethZ”来打开一个 ring 是可能的。

4） 用户空间 RING（“userspace”）模块

该模块允许一个应用程序发送数据包到另外一个进程，从而通过创建虚拟设备（例如：usrX，这里 X 是用户空间 ring 的唯一标识）来调节标准的 PF_RING API。为了使用它，发送应用程序必须使用设备名称“userspace:usrX”（这里“userspace:”标识用户空间 RING 模块）来打开 ring，此时接收应用程序必须使用设备名称“usrX”以标准方式打开一个 ring。

5） Libzero 消费者（“dnacluster”）模块

这个模块可以用于依附于 DNA Cluster，允许应用程序以标准 PF_RING API 来发送和接收数据包。发送应用程序必须使用“dnacluster:X@Y”（这里 X 是 Cluster 标示符，Y 是消费者标识符）或者“dnacluster:X”（自动分配消费者标识符）设备名称打开一个 ring。

7. PF_RING 应用程序开发者

概念上讲，PF_RING 是一个简单也强大技术，它能够让开发者在很短的时间内创建一个高速流量监控和管理应用程。这是因为 PF_RING 为开发者隐藏了内核详情，通过库和内核驱动来进行处理。在这种方式下，开发者显著的节约开发时间，只要关注他们开发的应用程序本身，而不要关注发送和接收数据包的方式。

本章包括：

1） PF_RING 的 API；

2）扩展 libpcap 库，支持遗留的应用程序。

7.1. PF_RING API

PF_RING 内部数据结构应该对用户是隐藏的，用户可以只通过定义在 pfring.h 头文件（与 PF_RING 一起发布）中的可用 API 来操作数据包和设备。

7.2. 返回码

按照惯例，库返回负值表示出现错误和异常。非零返回码表示成功。这种情况下，返回码有另一种含义，它们与相关的函数中进行描述。

7.3. PF_RING 设备名称约定

PF_RING 的设备名称与 libpcap 和 ifconfig 是相同的。因此 eth0 和 eth5 是可用的名称，你可以使用在 PF_RING 中。你也能指定称为“any”的虚拟设备，通知 PF_RING 从所有可用的网络设备上捕获数据包。

正如前面解释的，使用 PF_RING，你能够使用 Linux 发行版一起的驱动（不是 PF_RING 指定的），或者一些 PF_RING 支持的驱动（你可以在 PF_RING 的 drivers 目录中找到），它比普通的驱动更佳有效的捕获 PF_RING 数据包。如果你有最新的多队列 NIC 与 PF_RING 支持驱动一起运行（例如：Intel 10G 适配器），PF_RING 允许你从整个设备上捕获数据包（例如：捕获不管是接收数据包的 RX 队列的数据包，比如 ethX）。或者来自指定的队列（例如：ethX@Y）。假设你有一个带有 Z 队列的适配器，队列 ID Y，必须在 0 到 Z-1 范围内。这种情况下，你指定一个不存在的队列，没有数据包会捕获到。

正如先去章节描述的，PF_RING 4.7 有一个模块化的架构。为了说明我们使用那个模块的库，可能要把模块名称预先赋值给设备，通过分号分割开（例如：DNA 模块使用 dna:dnaX@Y，dag 模块使用 dag:dagX:Y，链路汇聚模块使用 multi:ethA@X;ethB@Y;ethC@Z，Cluster 模块使用 dnacluster:A@X。

7.4. PF_RING：SOCKET 初始化

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

这个调用用于初始化一个 PF_RING 套接字，然后获得一个 pfring 结构的句柄，它可以用于后续的调用。注意：

1） 你可以使用物理（例如：ethX）和虚拟（例如：tapX）设备，Rx-队列（例如：ethX@Y）和其他模块（例如：dna:dnaX@Y，dag:dagX:Y，multi:ethA@X;ethB@Y;ethC@Z，dnacluster:A@X）。

2） 为了打开设备，你需要超级用户的权限。

输入参数：

device_name

尝试打开的 PF_RING 支持设备的符号名称，例如：eth0

caplen

最大包捕获长度，也可以看做是 snaplen。

flags

它允许以一种基于位图的紧凑格式指定几个选项：

1) PF_RING_REENTRANT

以可重入的模式打开设备，通过信号量实现它，它会导致性能很差。进在多线程的应用程序中使用可重入模式。

2) PF_RING_LONG_HEADER

如果没有设置，PF_RING 不填充 pfring_pkthdr 结构中的 extended_hdr 字段。如果设置，

extended_hdr 字段也会适当的被填充。在不需要扩展信息的情况下，设置该值为 0 为了加速操作。

3) PF_RING_PROMISC

以混杂模式打开设备。

4) PF_RING_DNA_SYMMETRIC_RSS

设置硬件 RSS 函数为对称模式（相同方向的流进入相同的硬件队列）。只有 DNA 驱动才支持。

返回值：

成功返回一个句柄，否则返回 NULL 空。

```
u_int8_t pfring_open_multichannel(char *device_name,
        u_int32_t caplen, u_int flags,
        pfring* ring[MAX_NUM_RX_CHANNELS])
```

该调用类似于带有异常的 pfring_open()，当一个多 RX 队列 NIC 时，对整个驱动替换掉打开的单 ring，打开几个单独的 ring（每个 RX 队列上一个）。

输入参数：

device_name

尝试打开 PF_RING 支持设备的符号名称，例如：eth0。没有队列名称哈希到指定的位置，但是只有一个主设备名称。

caplen

最大报文捕获长度（也称为 snp 冷）

flags

参见 pfring_open 函数查看详情

ring

ring 数组的的指针，它们包括已打开的 ring 报文。

返回值：

Ring 数组（包含一个可用的 ring 指针）中最后一个索引。

7.5. PF_RING：设备关闭

```
void pfring_close(pfring *ring)
```

该函数用于关闭一个以前打开的 PF_RING 设备。注意，在你退出应用程序之前必须总是关闭掉设备。如果不确定，你可以从一个信号处理器中关闭一个设备。

输入参数：

ring

我们试图关闭的 PF_RING 句柄

7.6. PF_RING：读取输入数据包

```
int pfring_recv(pfring *ring, u_char** buffer, u_int buffer_len, struct pfring_pkthdr *hdr,
        u_int8_t wait_for_incoming_packet)
```

当有数据包时，该函数返回一个输入数据包。

输入参数：

ring

我们执行检查的 PF_RING 句柄

buffer

有调用者分配的一个内存区域，接收的数据包存储在该内存内。注意该参数是一个指向指针的指针，为了启用零拷贝实现（buffer_len 必须设置为 0）。

buffer_len

上面的内存区域的长度，注意：如果接收的数据包比分配的范围长的话，那么它会被截短。当可以使用零拷贝时，长度 0 表示使用零拷贝进行优化。

hdr

一个内存区域，数据包头部拷贝到该区域。

wait_for_incoming_packet

如果是 0，我们只是检查数据包是否可用，否则该调用会阻塞到一个数据包可用为止。

返回值：

如果没有包被接收到（非阻塞方式下）返回 0，1 表示接收到数据包，-1 表示发生错误。

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

pfring_recv()函数一样，添加其它的参数强迫数据包解析。

输入参数没有在 pfring_recv()函数中出现的有：

level

停止解析的头部级别，数据包解析到哪层。

add_timestamp

添加时间戳

add_hash

就算基于 IP 双向的哈希值

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

该调用处理数据包直到 pfring_breakloop()被调用或者一个错误发生。

输入参数：

ring

PF_RING 句柄

Looper

一个对每个接收到的数据包调用的回调函数。传递给该函数的参数有：一个 struct pfring_pkthdr 的指针，一个数据包内存指针和一个 user_bytes 的指针。

user_bytes

一个用户数据的指针，它传递给回调函数。

wait_for_packet

如果 0，启动等待，常用于检查数据包是否可用。

返回值：

如果 pfring_breakloop()函数被调用，返回一个非负数，如果错误返回一个负数。

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

当可用时，该调用返回下一个 income 数据包到达的时间。

输入参数：

ring

执行检查的的 PF_RING 句柄。

ts

用于存储时间的结构

返回值：

如果成功返回 0，如果错误返回一个负数。

int pfring_next_pkt_raw_timestamp(pfring *ring, u_int64_t *timestamp_ns)

该调用返回下一个接收数据包的原始时间戳，这只适用于 RX 硬件时间戳的适配器。

输入参数：

ring

执行检查使用的 PF_RING 句柄。

timestamp_ns

用于存储时间戳的内存。

返回值：

如果成功返回 0，如果发生错误返回一个负值。

7.7. PF_RING：ring Clusters

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

该调用允许一个 ring 添加到 Cluster，它可以交换地址空间。当二个以上的 socket 被 Cluster 时，它们共享以 per-flow 方式平衡的输入报文。这个技术对于利用多核系统中多线程情况下在同一个地址空间上共享数据包是有用的。

输入参数：

ring

用于 Cluster 的 PF_RING 句柄

clusterId

Culster 的数字标识符，指定那个 ring 被绑定。

the_type

Cluster 类型（Per-flow 或者是 round robin）。

返回值：

如果成功返回 0，否则返回负数值。

int pfring_remove_from_cluster(pfring *ring);

该调用允许从一个先前合并的 Cluster 中移除一个 ring。

输入参数：

ring

汇聚的 PF_RING 句柄。

clustered

ring 绑定到 Cluster 的一个数字标识符。

返回值：

如果成功返回 0，否则返回负数值。

7.8. PF_RING：数据包镜像

数据包映射可以在内核中桥接数据包，不需要发送它们到用户空间。你可以在过滤过则

中指定包映射。

typedef struct {

...

char reflector_device_name[REFLECTOR_NAME_LEN];

...

} filtering_rule;

在 reflector_device_name 中，你需要指定一个设备名称（例如：eth0），在数据包符合过滤规则的时候被映射。确保没有指定镜像设备与包捕获的设备名称一样，否则你将会创建一个数据包循环。

7.9. PF_RING：包采样

int pfring_set_sampling_rate(pfring *ring, u_int32_t rate)

直接在内核中实现数据包采样。注意：该解决方法比在用户空间中实现更加高效。采样的数据包只有那些经过所有过滤器的包。

输入参数：

ring

需要应用采样的 PF_RING 句柄。

rate

采样率，X 的比例意味 X 中的一个数据包被传递，这意味着采样率为 1 时禁用采样功能。

返回值：

如果成功返回 0，否则返回负数值。

7.10. PF_RING：包过滤

PF_RING 允许以两种方式过滤数据包：精确过滤（比如：哈希过滤器）或者通配符过滤。精确过滤用于跟踪一个精确 6 元组连接<vlan id, protocol, srcip, srcport, dst ip, dst port>是很有必要的。通配符过滤常用于替代一个过滤器在一些字段上有通配符（例如：符合所有 UDP 报文，不管他们的目的地址）。如果一些字段设置为 0，在过滤计算中它将会不参与。

7.10.1. PF_RING：通配符过滤

int pfring_add_filtering_rule(pfring *ring, filtering_rule* rule_to_add)

添加一个过滤规则到已存在的 ring 上。每个规则有一个唯一的规则 ID 号，例如：两个 ring 可以有相 ID 的规则。

输入参数：

ring

添加过滤规则的 PF_RING 句柄。

rule_to_addr

添加定义在本文档最后一章节中的规则。

返回值：

如果添加成功返回 0，否则返回负数值。

int pfring_remove_filtering_rule(pfring *ring, u_int16_t rule_id)

删除一个以前添加的过滤规则。

输入参数：

ring

删除过滤规则所在的PF_RING句柄。

Rule_id

将要删除的优先添加的过滤规则的规则ID。

返回值：

如果删除规则成功返回0，否则返回负数值（例如：规则不存在）。

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

读取哈希过滤规则的统计状态。

输入参数：

Ring

读取状态的PF_RING句柄。

Rule_id

读取统计信息的规则所指定的ID编号。

Stats

用户分配的一个缓冲区，可以包含规则统计信息，请确保缓冲区足够包含统计信息。这个缓冲区将包含接收和丢弃掉的数据包数量。

Stats_len

状态统计缓冲区的长度（字节）。

返回值：

如果成功返回0，否则返回一个负值（例如：规则不存在）。

int pfring_purge_idle_rules(pfring *ring, u_int16_t inactivity_sec);

删除与给定秒数不活动的规律规则。（超时）

输入参数：

Ring

移除规则所在的PF_RING句柄。

Inactivity_sec

不活动的门限值（秒）

返回值：

如果成功返回0，否则返回负值。

7.10.2. PF_RING:哈希过滤

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

添加或者删除一个哈希过滤规则。

输入参数：

Ring

用于读取状态的PF_RING句柄。

Rule_to_add

定义在本文档最后章节中的添加或者删除的规则。所有的参数应该定义在过滤规则中（不

是通配符）。

Add_rule

如果设置一个整数值，添加规则，如果为0则删除规则。

返回值：

如果成功返回0，否则返回一个负值（例如：删除掉不存在的过滤规则）。

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

读取哈希过滤规则的统计信息。

输入参数：

Ring

添加或者删除规则所在的PF_RING句柄。

Rule

读取统计信息的规则，这需要是以前已经添加的相同规则。

Stats

用户分配的缓冲区，用于包括规则的统计。请确保缓冲区足够大来包含统计信息。这些缓冲区可以包含接收和丢弃的数据包数量。

Stats_len

统计缓冲区的大小（字节为单位）。

返回值：

如果成功返回0，否则负值（例如：移除掉不存在的规则）。

int pfring_purge_idle_hash_rules(pfring *ring, u_int16_t inactivity_sec);

删除掉哈希过滤规则，在给定数量时间之内不激活时。

输入参数：

Ring

删除规则的PF_RING句柄。

Inactivity_sec

不活动的门限值。

返回值：

成功返回0，否则返回负值。

7.10.3. PF_RING：BPF 过滤

正如版本5.1ZHONG，通过PF_RING API来设置BPF过滤器是可以的。为了完成这个，必须是在编译时候选择BPF支持（使用，./configure –enable-bpf;make），并且把-lpcap库链接到PF_RING使能的应用程序上。

int pfring_set_bpf_filter(pfring *ring, char* filter_buffer)

设置BPF过滤条件到已经存在的ring上。

输入参数：

Ring

过滤器设置的PF_RING句柄。

Filter_buffer

过滤器设置为0.

返回值：

如果成功返回0，否则返回负值。

int pfring_remove_bpf_filter(pfring *ring)

删除掉BPF过滤器。

输入参数：

Ring

PF_RING句柄

返回值：

如果成功返回0，否则返回负值。

7.11. PF_RING：NIC 上包过滤

一些多队列的现代网络适配器具有“包控制”能力的特征。使用它们可以给硬件NIC发送一个指令让选择的报文分配到一个指定的RX队列中。如果指定的队列有一个超过最大队列ID的编号，这样的包将丢弃掉，因此可以作为硬件防火墙使用。

注意：DNA不支持内核报文过滤。

int pfring_add_hw_rule(pfring *ring, hw_filtering_rule *rule)

设置一个指定过滤规则给NIC，注意：不是添加PF_RING的过滤器，只是一个NIC过滤。

输入参数：

Ring

添加规则的PF_RING句柄

Rule

在NIC中设置的过滤规则，定义在本文档的最后章节。所有的规则参数应该被定义，并且如果设置为0，他们不参加过滤。

返回值：

如果成功返回0，否则返回负值。（流入：添加的规则格式有问题，或者如果ring绑定的NIC不支持硬件过滤。）。

int pfring_remove_hw_rule(pfring *ring, hw_filtering_rule *rule)

从NIC上删除掉指定规律规则。

输入参数：

Ring

删除过滤规则的PF_RING句柄

Rule

需要从NIC中删除的过滤规则。

返回值：

如果成功返回0，否则返回负值。

int pfring_set_filtering_mode(pfring *ring, filtering_mode mode)

设置过滤模式(只是软件/只用硬件/使用软件和硬件)，为了使用与软件（通配符和哈希）规则一样的API函数应用添加和删除硬件过滤。

输入参数：

Ring

设置过滤规则模式的PF_RING句柄

Mode

过滤模式

返回值：

如果成功返回0，否则为负值。

7.12. PF_RING：过滤策略

int pfring_toggle_filtering_policy(pfring *ring, u_int8_t rules_default_accept_policy)

设置缺省的过滤策略。这意味着如果没有规则符合输入的报文，缺省的策略决定报文是否传送到用户空间或者丢弃。注意：过滤策略受限于 ring 上，因此每个 ring 有不同的规则集合和缺省的策略。

输入参数:

Ring

添加和删除规则所用的 PF_RING 句柄。

Rules_default_accept_policy

如果设置为正值，那么接受缺省的规则（例如：传送报文到用户空间）否则丢弃掉。

返回值：

如果成功返回 0，否则为负值。

7.13. PF_RING：报文发送

依赖于所使用的驱动，报文传输可能会不同：

通用和 PF_RING 支持的驱动：FP_RING 不能加速 TX，由于标准 LINUX 发送机制被使用。当在这种模式下使用 PF_RING 不要希望速度优势。

TNAPI：包传输不支持；

DNA：支持线速的发送。

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

虽然 PF_RING 已经优化 RX，它也可以发送报文（TX）。该函数也许发送一个原始报文（例如：它发送到指定的网卡上）。该包必须全部指定（MAC 地址以上）并且它作为一个没有任何进一步操作情况下发送。

输入参数：

Ring

用于发送报文的 PF_RING 句柄。

Pkt

包括发送报文的缓冲区。

Pkt_len

报文缓冲区的长度

Flush_packet

1=刷新可能的发送缓冲区，如果设置为 0，你会减少 CPU 利用率，但是会付出发送报文的延时更大的代价。

返回值：

如果成功返回发送的字节数，否则返回负值。

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

与 pfring_send()相同，可以指定输出的接口索引。

在 pfring_send()中没有出现的输入参数：

If_index

赋值给输出设备的接口索引。

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

函数允许发送一个原始报文返回一个精确的时间（ns）它也会发送到网络上。注意：当适配器支持 TX 硬件时间戳的时候才能使用，并且这会影响到性能。

没有在 pfring_send（）中出现的输入参数：

Ts

保存 TX 时间戳的结构。

int pfring_send_last_rx_packet(pfring *ring, int tx_interface_id)

发送最后接收报文到指定的设备，这是一种只能工作在标准 PF_RING 上的优化的方式。

输入参数：

Ring

接收报文所在的 PF_RING 句柄。

Tx_interface_id

输出接口索引值

返回值

发送成功返回 0，否则为负值。

7.14. PF_RING：其它函数

int pfring_enable_ring(pfring *ring)

当创建一个ring时，它没有启用（流入：输入报文会被丢弃）直到调用上面的函数。

输入参数：

Ring：

启用的PF_RING句柄

返回值：

如果成功返回0，否则返回负值（流入：ring不能启用）。

int pfring_disable_ring(pfring *ring)

停止一个ring

输入参数：

Ring

停用的PF_RING句柄。

返回值：

如果成功返回0，否则返回负值。

int pfring_stats(pfring *ring, pfring_stat *stats)

读取ring统计信息（接收和丢弃的数据包）。

输入参数：

Ring

启用的PF_RING句柄

Stats

用户分配的缓冲区，统计信息（接收和丢弃的报文数量）将保存其中。

返回值：

如果成功返回0，否则返回负值。

int pfring_version(pfring *ring, u_int32_t *version)

读取ring的版本，注意：ring的版本3.7时返回ring版本为0x030700。

输入参数：

Ring

启用的PF_RING句柄

Version

用户分配的缓冲区，ring版本会拷贝到该缓冲区中。

返回值：

如果成功返回0，否则返回负值。

int pfring_set_direction(pfring *ring, packet_direction direction)

告诉PF_RING只考虑符合指定方向的数据包。如果应用程序没有调用该函数，所有的数据包（不管是TX或者RX方向）都被返回。

输入参数：

Ring

启用的PF_RING句柄

Direction

数据包的方向（RX，TX或者RX和TX）

返回值：

如果成功返回0，否则返回负值。

int pfring_set_socket_mode(pfring *ring, socket_mode mode)

告诉PF_RING应用程序是否需要从socket上发送或者接收数据包。

输入参数：

Ring

启用的PF_RING句柄。

Mode

套接字模式（发送，接收或者接收和发送）。

返回值：

如果成功返回0，否则返回负值

int pfring_poll(pfring *ring, u_int wait_duration)

在PF_RING套接字上执行被定等待，类似于标准的poll()，注意数据结构的同步。

输入参数：

Ring

轮询的PF_RING套接字。

Wait_duration

轮询的超时（单位：msec）

返回值：

如果成功返回0，否则返回负值。

int pfring_set_poll_watermark(pfring *ring, u_int16_t watermark)

用户空间应用程序必须等待直到输入数据包到达，它也可以让PF_RING不从poll（）调用处返回，除非至少“水印”包围被接收到。一个低级的水印值（例如：1）减少poll()的延时，但是增加了poll()调用的次数。一个高的水印（它不能扩充掉ring大小的50%，否则PF_RING内核模块将顶掉该值）可以减少poll()调用的次数，但是会增加报文的延时。缺省的水印值为128（例如：用户空间应用程序是否使用该调用操作该值）。

输入参数：

Ring
启用的PF_RING句柄
Watermakar
报文轮询的水印。
返回值：
如果成功返回0，否则为负值。

int pfring_set_tx_watermark(pfring *ring, u_int16_t watermark)

在发送到网卡上之前，设置缓冲在流出队列中的数据包数量。
输入参数：
Ring
启用的PF_RING句柄
Watermark
TX的水印
返回值：
如果成功返回0，否则负值。

int pfring_set_poll_duration(pfring *ring, u_int duration)

当使用被动等待时，设置轮询的超时时间。
输入参数：
Ring
启用的PF_RING句柄
Duration
轮询的超时，单位msec。
返回值：
如果成功返回0，否则为负值。

int pfring_set_application_name(pfring *ring, char *name)

告诉PF_RING使用这个ring的应用程序的名称，通常使用argv[0]。该信息用户标识应用程序，访问PF_RING/proc文件系统中的文件。例如：

> 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

输入参数：
Ring
启用的PF_RING句柄
Name
使用ring的应用程序名称
返回值：
如果成功返回0，否则为负值。

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

返回绑定到套接字的设备的MAC地址。
输入参数：

Ring
查询的PF_RING句柄
Mac_address
MAC地址拷贝的内存区域。
返回值：
成功返回0，否则为负值。

int pfring_get_bound_device_id(pfring *ring, int* device_id)

返回绑定到套接字的设备编号
输入参数：
ring
查询的PF_RING句柄
Device_id
存放设备的编号的内存区域
返回：
成功返回0，否则为负值。

u_int8_t pfring_get_num_rx_channels(pfring *ring)

返回ring绑定的以太接口的RX通道的数量（也看作为RX队列）。
输入参数：
Ring
查询的PF_RING句柄。
返回值：
RX通道的数量，或者1（缺省），在这种情况下信息未知。

int pfring_get_selectable_fd(pfring *ring)

返回与指定ring相关的文件描述符。该数值可以用于函数调用，例如：poll()和select()用于被动等待输入的报文。
输入参数：
Ring
查询的PF_RING句柄
返回值：
一个可以用于引用该ring的数值，在函数调用中需要一个可选的文件描述符。

int pfring_enable_rss_rehash(pfring *ring)

告诉PF_RING使用双向的哈希函数重新哈希输入的报文。
输入参数：
Ring
查询的PF_RING句柄
返回值：
成功返回0，否则为负值。

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

从设备硬件时钟上读取时间，当适配器支持硬件时间戳时。
输入参数：
Ring
PF_Ring句柄
Ts
存储时间的结构

返回值：

成功返回0，否则为负值。

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

设置设备硬件时钟时间戳，当适配器支持硬件时间戳时。

输入参数：

Ring

PF_RING句柄

Ts

设置的时间戳

返回值：

成功返回0，否则为负值。

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

当适配器支持硬件时间戳时， 使用一个偏移量调整设备硬件时钟的时间。

输入参数：

Ring

PF_RING句柄

Offset

时间的偏移量

Sing

偏移的符号

返回值：

如果成功返回0，否则负值。

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

当适配器支持的时候，启用RX和TX硬件时间戳。

输入参数：

Ring

PF_RING句柄

Device_name

启用时间戳的设备名称。

Enable_rx

启用RX时间戳的标志

Enable_tx

启用TX时间戳的标志。

返回值：

成功返回0，否则为负值。

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

解析一个报文

输入参数：

Ptk

报文缓冲区

Hdr

填充的头部

Level
停止解析的头部级别
Add_timestamp
添加时间戳
Add_hash
计算基于IP的双向哈希。
返回值：
如果成功返回一个非负的数值表示那个级别的头部被解析，否则返回负值。

7.15. C++ PF_RING 接口

C++接口（参见：PF_RING/userland/c++/）等价于 C 接口。没有做任何主要的改变，所有使用的方法都与 C 的相同，例如：

1） C：int pfring_stats(pfring* ring, pfring_stat *stats);

2） C++: inline int get_stats(pfring_stat *stats)

8. DNA 的 libzero 库

该库实现一个零拷贝内部处理通信，它常用于在多线程和多处理器应用程序中。正如在介绍中说明的，它提供两个主要的组件：DNA Cluster 和 NDA Bouncer。

8.1. DNA Cluster

DNA Cluster 实现数据包的汇聚，所有属于同一个 Cluster 的应用程序可以使用用户定义的负载函数以零拷贝的方式共享输入的数据包。应用程序也已零拷贝的方式发送数据包。每个应用程序读取/发送数据包到一个“从”socket 上。

一个主线程/应用程序用于分配输入数据包到使用一个用户定义均衡函数的 Slave 上（缺省是一个双向基于 IP 的哈希函数）。它也作为 fan-out（扇出使用，发送一些数据包到多个从线程/应用程序上，在没有最慢的消费者影响最快的消费者的情况下。

Cluster 允许应用程序处理“不全的“数据包（例如：不能顺序的处理数据包）。移动下一个输入报文，甚至也不管在前一个报文没有被处理完。

8.1.1. 主 API

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

创建一个新的 DNA Cluster 句柄，Cluster 只是创建，没有 ring 与之相关。

输入参数：

Cluster_id

Cluster 的标示符。

Num_apps

从应用程序/线程的数量。

Flags

启动其它扩展的掩码，例如根据分发函数返回的指来传输接收数据包给输出结构的“方向传送“。

返回值：

Cluster 的句柄。

int dna_cluster_register_ring(pfring_dna_cluster *handle, pfring *ring)

添加一个 PF_RING 套接字给 DNA Cluster。

输入参数：

Handle

DNA Cluster 的句柄

Ring

PF_RING 句柄

返回值：

如果成功返回 0，否则为负值。

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

绑定 RX 和 TX 主线程到给定的内核 ID 上。在 Cluster 中，线程用于轮训 RX 和发送（TX）Cluster 的数据包。该函数用于指定线程与 CPU 核的亲和力。

输入参数：

Handle

DNA Cluster 句柄。

Rx_core_id

RX 线程内核 ID。

Tx_core_id

TX 线程的内核 ID。

int dna_cluster_set_mode(pfring_dna_cluster *handle, socket_mode mode)

设置 Cluster 模式：接收（只有 TX），发送（TX），或者接收和发送（TX 和 TX）。

输入参数：

Handle

DNA Cluster 句柄

Mode

Cluster 的模式。

返回值：

如果成功返回 0，否则返回为负值。

void dna_cluster_set_distribution_function(pfring_dna_cluster *handle,
pfring_dna_cluster_distribution_func func)

设置包分发函数（确实的函数是一个双向基于 IP 的哈希函数）。该调用允许开发者指定它们自己的函数。

输入参数：

Handle

DNA Cluster 句柄

Func

分发函数

void dna_cluster_set_wait_mode(pfring_dna_cluster *handle, u_int32_t active_wait)

设置进入数据包的等待模式：被动（轮询）或者主动等待。

输入参数：

Handle

DNA Cluster 句柄

Active_wait

布尔值：0 是被动模式，1 是租主动模式。

返回值：

如果成功返回 0，否则为负值。

int dna_cluster_enable(pfring_dna_cluster *handle)

启用 Cluster

输入参数：

Handle

DNA Cluster 句柄。

返回值：

如果成功返回 0，否则为负值。

int dna_cluster_disable(pfring_dna_cluster *handle)

停止 Cluster

输入参数：

Handle

DNA Cluster 句柄。

返回值：

如果成功返回 0，否则为负值。

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

返回 Cluster 的状态。

输入参数：

Handle

DNA Cluster 句柄。

Tot_rx_packets

接收报文的总数量。

Tot_tx_packets

发送报文的总数量。

Tot_rx_processed

有 slave 处理的报文总数量。

返回值：

如果成功返回 0，否则为负值。

void dna_cluster_destroy(pfring_dna_cluster *handle)

销毁 Cluster，并且关闭绑定的 PF_RING 套接字。

输入参数：

Handle

DNA Cluster 句柄。

返回值：

如果成功返回 0，否则为负值。

8.1.2. Slave API

DNA Cluster Salve 线程/应用程序使用的 PF_RING API 的超集，保证向后兼容所有现有的应用程序。在该文档的后面，你可以发现在两个集合之间的不同。

pfring_pkt_buff* pfring_alloc_pkt_buff(pfring *ring)

返回一个包缓冲区句柄，由 PF_RING 分配的内存到内核，并且它由 PF_RING 管理（例如：不能释放 free()这块内存），只有语 pfring_xxx_xxx 调用。

输入参数：

Ring

PF_RING 句柄

返回值：

只返回缓冲区的句柄。

void pfring_release_pkt_buff(pfring *ring, pfring_pkt_buff *pkt_handle)

释放先前有 pfring_alloc_pkt_buf 分配的报文缓冲区句柄。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄。

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

接收由 pkt_handle 指向的填充的缓冲区报文，替代返回一个新的缓冲区。在小容器中，返回报文放置在传递函数参数中。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄，这里用于放置输入数据包。

Hdr

PF_RING 头部

Wait_for_incoming_packet

如果 0，简单检查数据包是否可用，否则调用阻塞到一个数据包可用。

返回值

没有包接收到时返回 0（非阻塞），如果成功返回 0，错误发生返回-1。

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

发送 pkt_handle 缓冲区句柄指向的数据包。注意：该函数重设置缓冲区句柄的内容，因此你需要保持它的内容，保证你在调用它之前拷贝数据。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄

Flush_packet

刷新可能传输的队列

返回值：

如果成功返回发送的字节数，否则返回负值。

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

返回报文缓冲区句柄指向的缓冲区指针。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄

返回值：

只想数据包缓冲区。

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

设置报文长度，除非你希望自定义设置报文的长度，该函数调用不需要，它替代从接收数报文所使用的大小。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄

Len

报文长度。

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

绑定缓冲区句柄（处理一个报文）到接口 ID 上，该函数调用对于指定出接口索引时有用的。

输入参数：

Ring

PF_RING 句柄

Pkt_handle

缓冲区句柄

If_index

接口的 ID 号

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

添加一个接口 ID 到缓冲区句柄的绑定的接口 ID 上。该函数用于指定数据包缓冲区的输出接口（扇出）。

输入参数：

Ring

PF_RING 处理

Pkt_handle
缓冲区句柄
If_index
接口的 ID 号

8.2. DNA Bouncer

DNA Bouncer 把数据包以零拷贝的方式交换给两个接口，让用户有能力指定一个函数来决定，报文到报文，给定的报文是否必须传递。Bouncer 是有方向的，意味着数据包只拷贝到一个方向（流入到流出的 ring）。如果你需要一个双向的拷贝，你需要创建两个 Bouncer。

8.2.1. DNA Bouncer API

pfring_dna_bouncer *pfring_dna_bouncer_create(pfring *ingress_ring, pfring *egress_ring)
创建一个新的 DNA Bourcer 句柄
输入参数：
Ingress_ring
数据包被读取的套接字
Egress_ring
数据包被发送的套接字
返回值：
DNA Bouncer 句柄

int pfring_dna_bouncer_set_mode(pfring_dna_bouncer *handle, dna_bouncer_mode mode)
设置DNA Bouncer模式为单路（确实）或者双路。
输入参数：
Handle
DNA Bouncer句柄
Mode
模式：one_way_mode或者two_way_mode。
返回值：
如果成功返回0，否则为负数。

int pfring_dna_bouncer_loop(pfring_dna_bouncer *handle ,pfring_dna_bouncer_decision_func func, const u_char *user_bytes, u_int8_t wait_for_packet)
启用DNA Bouncer。
输入参数：
Handle
DNA Bouncer句柄
Func
数据包处理函数
User_bytes

用户数据指针
Wait_for_packet
如果0，主动等待用户检查数据包是否可用。
返回值：
如果成功返回0，否则返回负值。
void pfring_dna_bouncer_breakloop(pfring_dna_bouncer *handle)
停止 Bouncer
输入参数：
Handle
DNA Bouncer 句柄。
void pfring_dna_bouncer_destroy(pfring_dna_bouncer *handle)
销毁 Bouncer，并且关闭 PF_RING 套接字。
输入参数：
Handle
DNA Bouncer 句柄。

8.3. 常用的实例的代码片段

8.3.1. DNA Cluster:接收数据包和输出

```
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);
```

8.3.2. DNA Cluster：接收一个数据包并且以零拷贝发送

```
pfring_pkt_buff *pkt_handle = pfring_alloc_pkt_buff(ring);
if (pkt_handle != NULL) {
rc = pfring_recv_pkt_buff(ring, pkt_handle, &hdr, wait_for_packet);
if (rc > 0) {
if (forward_packet_to_another_interface) {
pfring_set_pkt_buff_ifindex(ring[thread_id], pkt_handle, if_index);
} else {
```

```
/* bounce packet on the rx interface (already set in pkt_handle) */
}
pfring_send_pkt_buff(ring[thread_id], pkt_handle, 0);
}
}
```

8.3.3. DNA Cluster:使用自定义函数替代缺省负载均衡函数

```
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);
```

8.3.4. DNA Cluster:使用扇出函数替代确实的负载均衡函数

```
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);
```

8.3.5. DNA Cluster:不通过 Slave 直接发送一个输入数据包

```
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);
```

9. 编写 PF_RING 插件

自从 3.7 版本，开发者可以编写插件，从而委派 PF_RING 的活动，如下：

数据包负载解析；

数据包内容过滤；

内核流量统计计算。

为了澄清这个概念，想象下你需要开发一个 VoIP 流量监控的应用程序。在这种情况下必须：

解析信令包（例如：SIP 或 IAX），以至于只有属于感兴趣的对的数据包通过；

计算语音统计到 PF_RING 中，并且只把统计信息报告给用户控件，而不是数据报文。

在这种情况下，开发者可以编写两个插件，以至于 PF_RING 可以作为高级流量过滤器使用，当不需要的时候通过避免数据包经过内核边缘的方式加快数据包的处理。

本章的下面部分解释如何实现插件和如何从用户空间中调用。

9.1. 实现 PF_RING 插件

在 kernel/net/ring/plugins 目录中，这里有一个简单的插件调用 dummy_plugin，它显示如何实现一个简单的插件。让我们来剖析这个代码。

每个插件作为一个 Linux 内核模块实现，每个模块必须有两个入口点，module_init 和 module_exit，当模块插入和删除的时候调用它们。在 dummy_plugin 例子中的 module_init 函数使用 dummy_plugin_init()函数实现，通过调用 register_plugin()函数来注册该插件。传递给注册函数的参数是 struct pfring_plugin_registration 的数据结构，包含：

pulgin_id

一个唯一的整数插件 ID。

pfring_plugin_filter_skb

函数指针，在包需要过滤的时候调用，该函数在 pfring_plugin_handle_skb()之后调用。

Pfring_plugin_get_stats

函数指针，它在用户希望读取过滤规则统计的时候调用，作为插件的一个动作。

Pfring_plugin_purge_idle

函数指针，用户希望插入过滤过则的时候被调用，作为插件的一个动作。

Pfring_plugin_free_rule_mem

函数指针，当过滤规则被删除的时候被调用，作为插件的一个动作。

Pfring_plugin_free_ring_mem

函数指针，当插件取消注册（rmmod）或者插件使用的 ring 删除的时候调用。释放所有在操作期间插件分配的全局内存。

Pfring_plugin_add_rule

函数指针，当用户设置该插件带有forward_packet_add_rule_and_stop_rule_evaluation行为的过滤规则的时候调用。在数据包匹配的时候，该函数被调用。

Pfring_plugin_add_rule

函数指针，当用户设置插件带有forward_packet_del_rule_and_stop_rule_evaluation行为的过滤规则时候调用。

开发者可以选择不实现所有上面的函数，但是在这种情况下插件将会被功能所限制。（例如：如果pfring_plugin_filter_skb为NULL，不支持过滤）。

9.2. PF_RING 插件：处理输入数据包

```
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)
```

该函数在接收数据包（RX 或者 TX）时候被调用。该函数通常更新规则的统计。注意如果开发者设置插件作为过滤插件，那么数据包：

已经被解析；

如果设置，传递一个规则负载过滤器。

输入参数：

Rule

指向通配符规则的指针（如果插件已经设置为一个通配符规则）或者 NULL（如果插件设置为一个哈希规则）。

Has_rule

指向哈希规则的指针（如果插件设置为一个哈希规则）或者 NULL（如果插件已经设置为通配符规则）。注意 rule 为 NULL 时，没有 hash_rule，反之亦然。

Hdr

指向接受数据包的 pcap 数据包头的指针，请注意：

数据包已经被解析；

头部时 pcap 头部的扩展，包含解析包头部元数据。

Skb

一个 sk_buff 的数据结构，在 linux 内核中用来承载数据包。

Filter_plugin_id

已经解析数据包负载的插件的ID（不包含已经存储在hdr的头部）。如果filter_plug_id与dummy_plugin的ID以中央，那么包已经被插件解析，并且参数filter_rule_memory_storage指向负载的解析内存上。

Parse_memory

指向数据结构，包含解析数据包负载的信息，它已经被有参数filter_plugin_id指定的插件所解析。注意：

只有一个插件解析一个数据包；

解析内存由plugin_filter_skb动态分配（例如：利用kmalloc），并且有PF_RING内核释放。

返回值：

如果成功返回 0，否则返回一个负值。

9.3. PF_RING 插件：过滤输入数据包

```
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)
```

该函数在先前解析数据包（利用 plugin_handle_skb）时被调用，接收的数据包（RX 或者 TX）需要被过滤。在这时，数据包被解析，返回解析信息，返回值表示数据包是否通过过滤器。

输入参数：

Rule

指向通配符规则的指针，包括应用到数据包的负载过滤器。

Hdr

指向接收数据包的 pcap 数据包的头部，请注意：

数据包已经被解析

头部时一个 pcap 头部的扩展，包括解析数据包头部的原数据。

Skb

一个 Linux 中使用 sk_buff 数据结构，用于在内核中承载数据包。

输出参数：

Parse_memory

有函数分配的内存指针，它包括关于解析数据包负载信息。

返回值：

如果包不符合规律规则返回 0，否则返回负值。

9.4. PF_RING 插件：读取数据包统计

```
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)

该函数在用户空间的应用程序希望读取过滤规则的统计时调用。

输入参数：

指向通配符规则的指针（如果插件已经设置为一个通配符规则）或者 NULL（如果插件设置为一个哈希规则）。

Has_rule

指向哈希规则的指针（如果插件设置为一个哈希规则）或者 NULL（如果插件已经设置为通配符规则）。注意 rule 为 NULL 时，没有 hash_rule，反之亦然。

Stats_buffer

统计信息被拷贝的缓冲区指针。

Stats_buffer_len

Stats_buffer 的字节长度。

返回值：

规则统计的长度，如果发生错误返回 0。

9.5. 使用 PF_RING 插件

一个基于 PF_RING 的应用，当过滤规则设置时可以利用插件的优点。Filtering_rule 的数据结构用于设置规则和指定与之有关的插件。

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

当设置plugin_action.plugin_id时，只要符合规则头部部分的数据包，MY_PLUGIN_ID(如果注册)插件被调用，plugin_filter_skb () 和plugin_handle_skb()也被调用。

如果开发者希望在plugin_filter_skb () 调用之前过滤一个数据包，那么扩展的filtering_rule字段需要被设置。例如：假设实现一个SIP过滤插件，并且使用它，为了只返回INVITE的数据包。下面几行代码显示如何完成：

```
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 */
```

正如以前解释的，pfring_add_filtering_rule()函数用于注册过滤规则。

10. PF_RING 数据结构

下面描述一些与 PF_RING 相关的数据结构。

```
typedef struct {
    u_int16_t rule_id; /* Rules are processed in order from
```

```
                lowest to higest id */
        rule_action_behaviour rule_action; /* What to do in case of match */
        u_int8_t balance_id, balance_pool; /* If balance_pool > 0, then pass the
                                              packet above only if the
                                              (hash(proto, sip, sport, dip, dport) %
                                              balance_pool) = balance_id */
        u_int8_t locked; /* Do not purge */
        u_int8_t bidirectional; /* Swap peers (Default: mono) */
        filtering_rule_core_fields core_fields;
        filtering_rule_extended_fields extended_fields;
        filtering_rule_plugin_action plugin_action;
        char reflector_device_name[REFLECTOR_NAME_LEN];
        filtering_internals internals; /* PF_RING internal fields */
} filtering_rule;

typedef struct {
        u_int8_t smac[ETH_ALEN], dmac[ETH_ALEN]; /* Use '0' (zero-ed MAC address) for
        any MAC address. This is applied
        to both source and destination */
        u_int16_t vlan_id; /* Use '0' for any vlan */
        u_int8_t proto; /* Use 0 for 'any' protocol */
        ip_addr shost, dhost; /* User '0' for any host. This is applied
        to both source and destination. */
        ip_addr shost_mask, dhost_mask; /* IPv4/6 network mask */
        u_int16_t sport_low, sport_high; /* All ports between port_low...port_high
        means 'any' port */
        u_int16_t dport_low, dport_high; /* All ports between port_low...port_high
        means 'any' port */
} filtering_rule_core_fields;

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

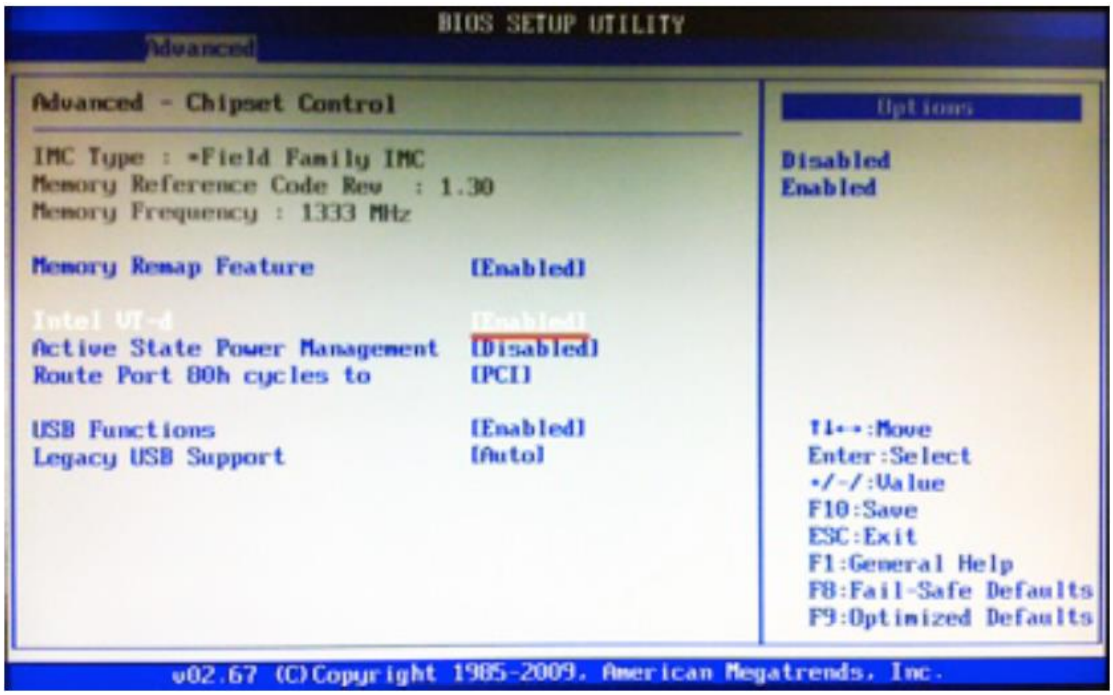
```
typedef enum {
    forward_packet_and_stop_rule_evaluation = 0,
    dont_forward_packet_and_stop_rule_evaluation,
    execute_action_and_continue_rule_evaluation,
    execute_action_and_stop_rule_evaluation,
    forward_packet_add_rule_and_stop_rule_evaluation,/* auto-filled hash rule or
    via plugin_add_rule() */
    forward_packet_del_rule_and_stop_rule_evaluation,/* plugin_del_rule() only */
    reflect_packet_and_stop_rule_evaluation,
    reflect_packet_and_continue_rule_evaluation,
    bounce_packet_and_stop_rule_evaluation,
    bounce_packet_and_continue_rule_evaluation
} rule_action_behaviour;

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

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

11. 虚拟机上的 PF_RING DNA

第 4.4 节包括一个 PF_RING DNA 模块的简单介绍，它允许你捕获任意包大小的 10G 线速数据包。感谢基于 IOMMU（Intel VT-d AMD IOMMU）的虚拟技术，它现在可以把设备添加到给定客户操作系统，在 VM（虚拟机）中获得 PF_RING DNA 模块的好处。下面章节显示如何配置 VMware 和 KVM（Linux 本地虚拟系统）。XEN 用户可以使用类似系统的配置。

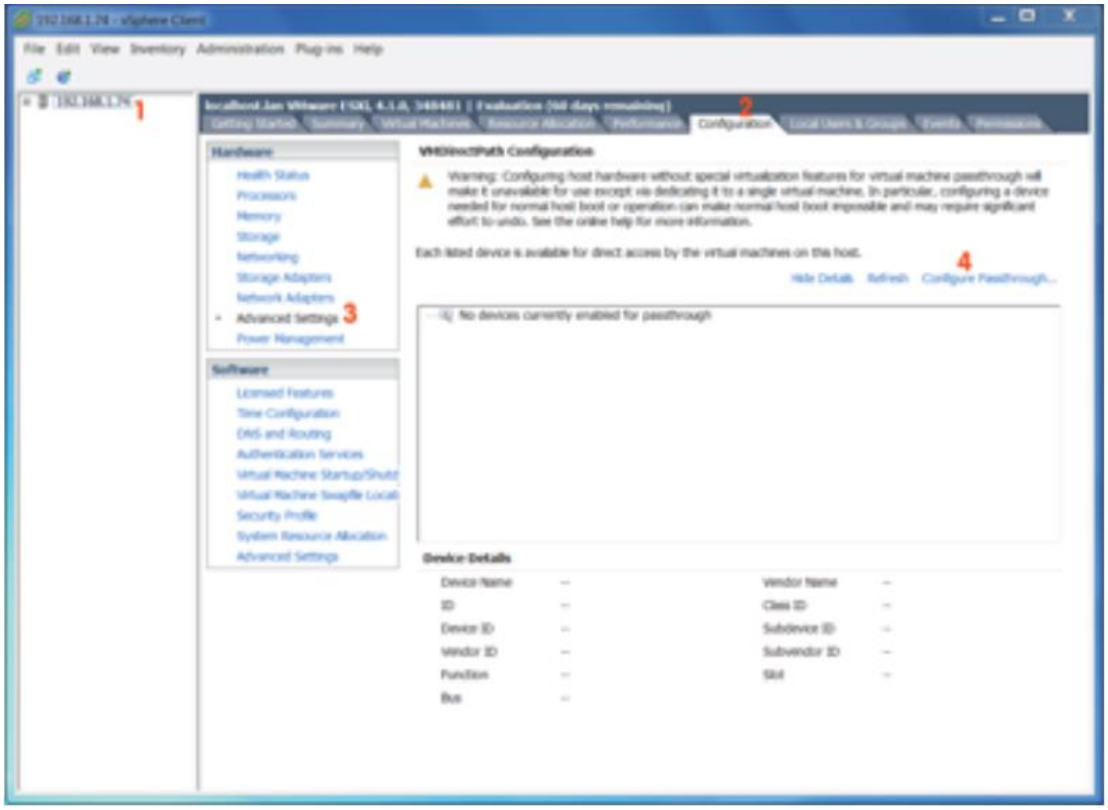
11.1. BOIS 配置

首先，确保你的主板支持 PCI 旁路，并且检查 BOIS 设置中是否启用。

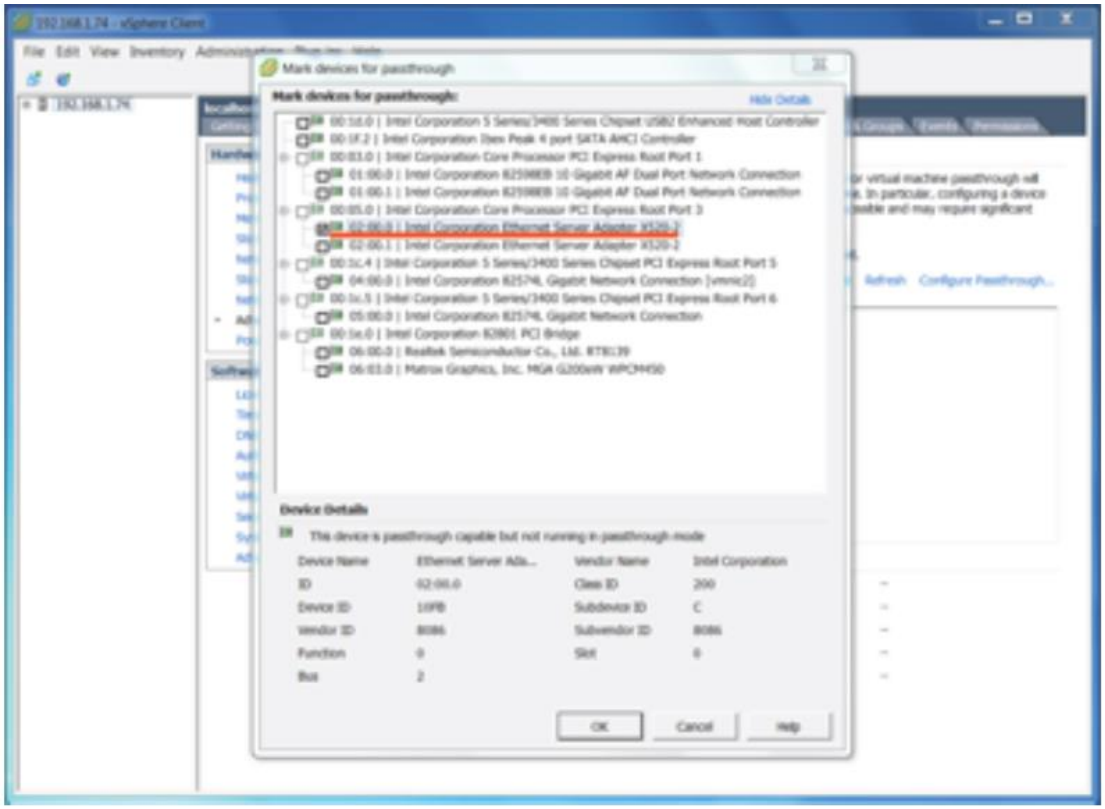


11.2. VMware ESX 配置

为了在 VMare 中配置 PCI 旁路，打开 vSphere 客户端并且连接到服务器。选择服务器，到“配置”，“高级设置”，“配置旁路”。

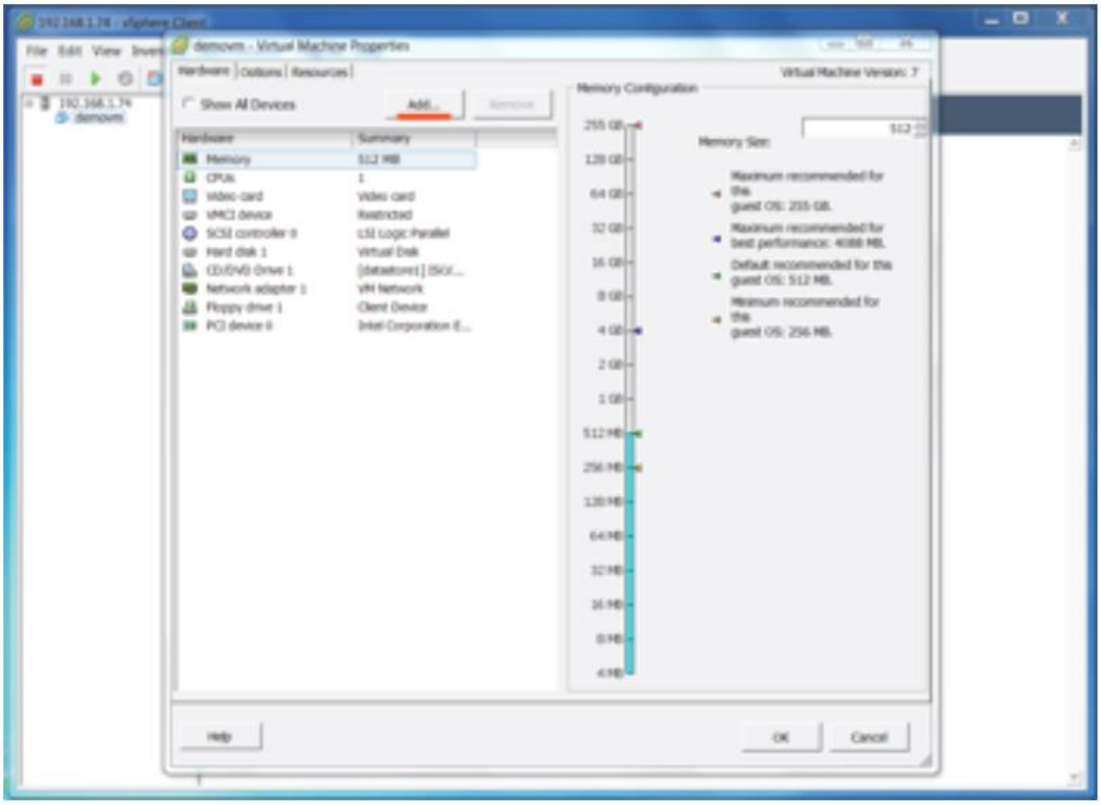


选择你希望赋值的 VM 设备。

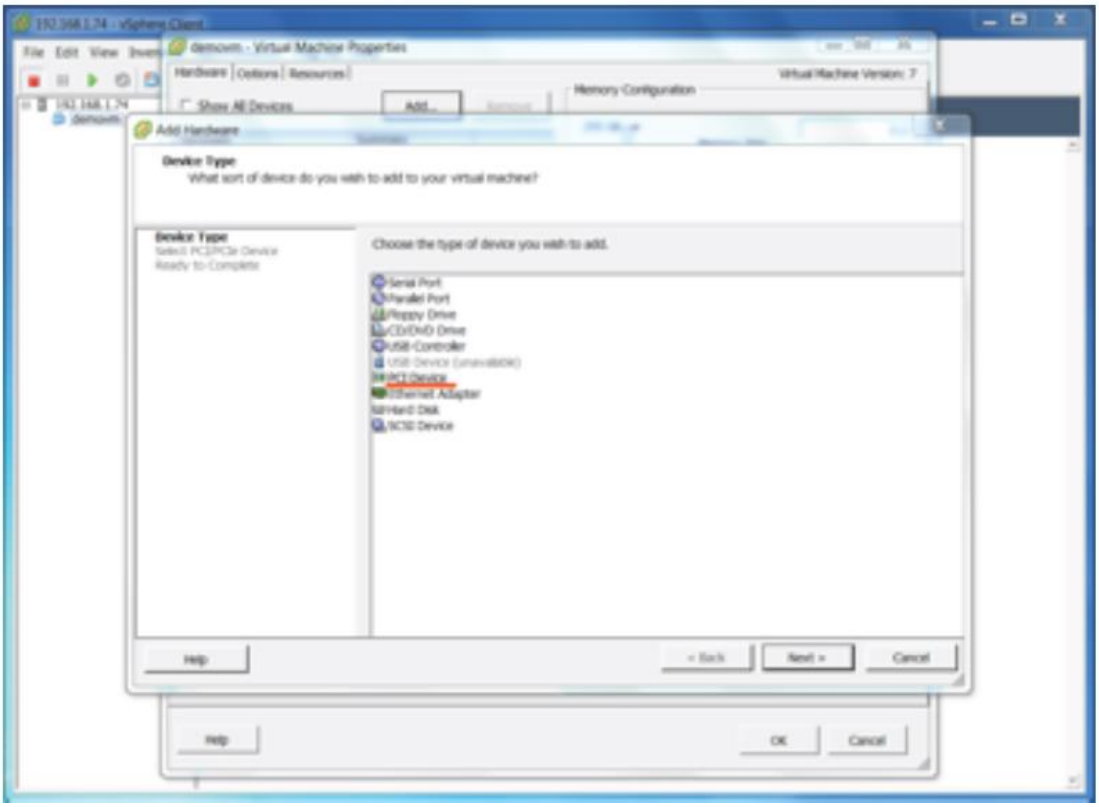


重启服务器。

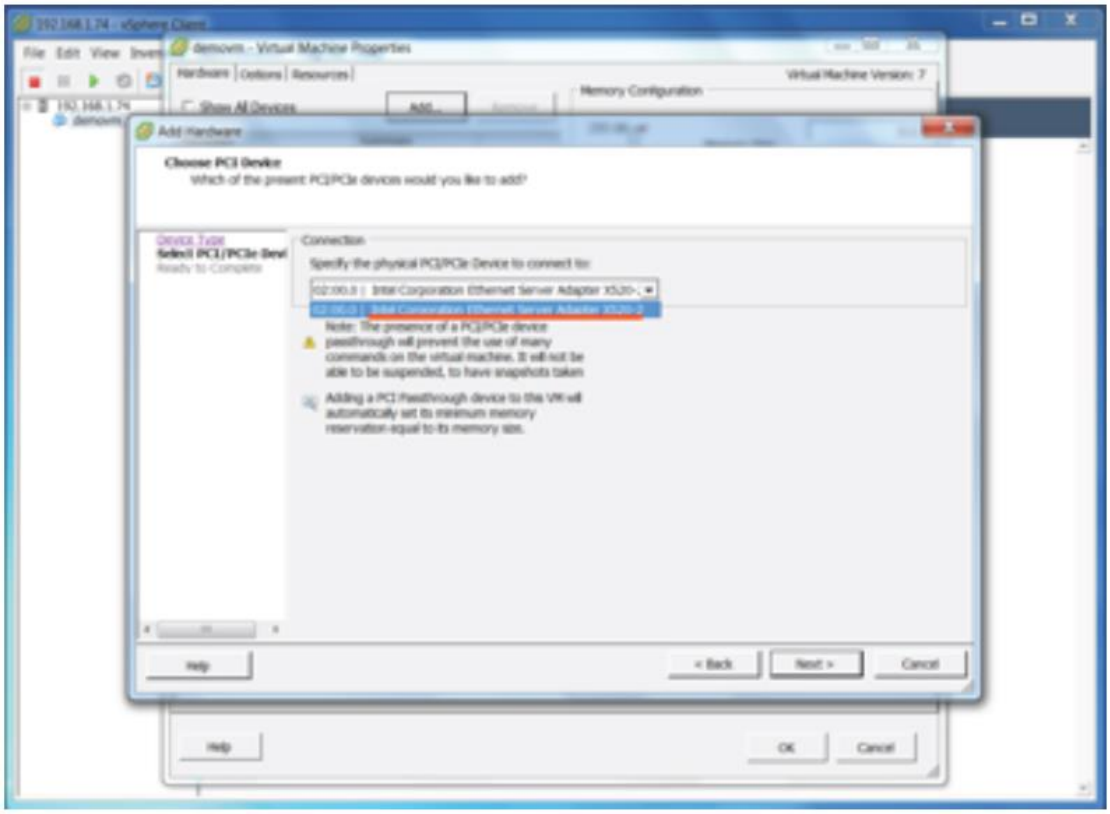
在重启之后，确保 PCI 设备所在的 VM 处于 off 状态。打开 VM 设置，点击“硬件“标签中的“Add…”。



选择“PCI 设备“



选择设备赋值给 VM。



VM 和安装带有 DNA 驱动的 PF_RING 都作为本地情况。

11.3. KVM 配置

为了配置 KVM 的 PCI 旁路，保证在内核中已经启用下面选项。

总线选项（PCI 等）：

[*] Support for DMA Remapping Devices

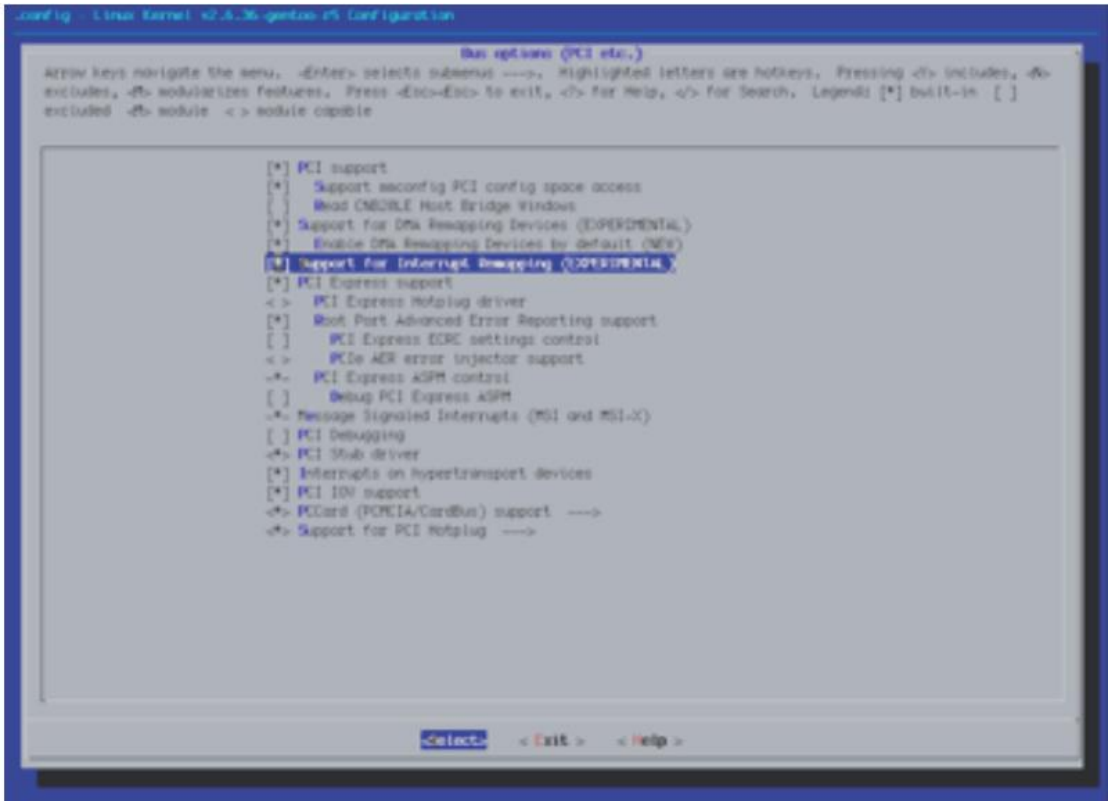
[*] Enable DMA Remapping Devices

[*] Support for Interrupt Remapping

<*> PCI Stub driver

$ cd /usr/src/linux

$ make menuconfig



$ make

$ make modules_install

$ make install

(或者使用你的发行版特有的方式)。

传递“intel_iommu=on”作为内核参数。例如，如果你使用 grub，编译/boot/grub/menu.lst 方式：

title Linux 2.6.36

root (hd0,0)

kernel /boot/kernel-2.6.36 root=/dev/sda3 intel_iommu=on

不要绑定你提供给 VM 的设备到主机内核驱动上。

$ 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

装载 KVM 并且启动 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
```

在本地情况下安装和运行带有 DNA 驱动的 PF_RING。