

Руководство пользователя

PF_RING

Высокоскоростной захват пакетов в Linux

Версия 5.5.3
Май 2013

© 2004-13 ntop.org

1.Содержание

2. Введение.....	4
2.1. Что нового в руководстве пользователя PF_RING?.....	4
3. Добро пожаловать в PF_RING.....	6
3.1. Фильтрация пакетов.....	7
3.2. Путешествие пакета.....	7
3.3. Пакетная Кластеризация (Группирование).....	8
4. Семейство драйверов PF_Ring.....	9
4.1. PF_RING-ориентированные драйвера.....	9
4.2. TNetAPI.....	9
4.3. DNA.....	10
5. Библиотека Libzero для DNA.....	11
5.1. DNA кластер.....	11
5.2. DNA проталкиватель.....	11
6. Установка PF_RING.....	12
6.1. Установка модуля ядра Linux.....	12
7. Запуск PF_RING.....	13
7.1. Проверка конфигурации элемента PF_RING.....	14
7.2. Установка Libpfiring и Libpcap.....	14
7.3. Примеры приложений.....	15
7.4. Дополнительные модули PF_RING.....	16
8. PF_RING для разработчиков приложений.....	17
8.1. API PF_RING.....	18
8.2. Возвращаемые коды.....	18
8.3. Согласование имен устройств PF_RING.....	18
8.4. PF_RING: инициализация сокета.....	18
8.5. PF_RING: Отключение устройства.....	20
8.6. PF_RING: Чтение входящих пакетов.....	21
8.7. PF_RING: Кластерное кольцо.....	23
8.8. PF_RING: Зеркалирование пакетов.....	24
8.9. PF_RING: Выборка пакетов.....	24
8.10. PF_RING: Фильтрация пакетов.....	24
8.10.1. PF_RING: Неполная фильтрация.....	25
8.10.2. PF_RING: Конкретная фильтрация.....	26

8.10.3. PF_RING: BPF фильтрация	27
8.11. PF_RING: Фильтрация пакетов в NIC	28
8.12. PF_RING: Политика фильтрации	29
8.13. PF_RING: Передача пакетов	30
8.14. PF_RING: Прочие функции	31
8.15. Интерфейс PF_RING в C++	39
9. Библиотека libzero для DNA.....	39
9.1. Кластер DNA	39
9.1.1. Основные API (The Master API)	39
9.1.2. Подчинённые API.....	43
9.2. DNA проталкиватель	46
9.2.1. API DNA проталкивателя	46
9.3. Фрагменты кода для общих случаев.	48
9.3.1. DNA кластер: принять пакет и отложить его в сторону.....	48
9.3.2. DNA кластер: приём пакета и отправка его через механизм zero-copy	48
9.3.3. Кластер DNA: Замена стандартной функции распределения на пользовательскую функцию.	49
9.3.4. Кластер DNA: Замена стандартной функции распределения на разветвляющую функцию.	49
9.3.5. Кластер DNA: отправка входящего пакета сразу, без прохождения через подчинённый процесс или программу.....	50
10. Пишем плагины к PF_RING.....	50
10.1. Реализация плагина PF_RING	51
10.2. Плагин PF_RING: Дескриптор входящих пакетов.....	51
10.3. Плагин PF_RING: Фильтрация входящих пакетов.	53
10.4. Плагин PF_RING: Чтение статистики пакетов.	53
10.5. Использование плагина PF_RING	54
11. Структуры данных PF_RING.....	55
12. PF_RING DNA на виртуальной машине.	56
12.1. Конфигурация BIOS.....	56
12.2. Конфигурация VMware ESX.....	56
12.3. Конфигурация KVM.....	58

2. Введение.

PF_RING это высокоскоростная библиотека захвата пакетов, которая превращает PC в эффективное и дешевое сетевое устройство, ориентированное для активного анализа трафика и его управления. Кроме того, PF_RING открывает абсолютно новые рынки, так как позволяет создавать эффективные приложения распределения трафика или фильтрации пакетов несколькими строчками кода.

Это руководство разделено на 2 части:

- Установка PF_RING и его конфигурация.
- PF_RING SDK.

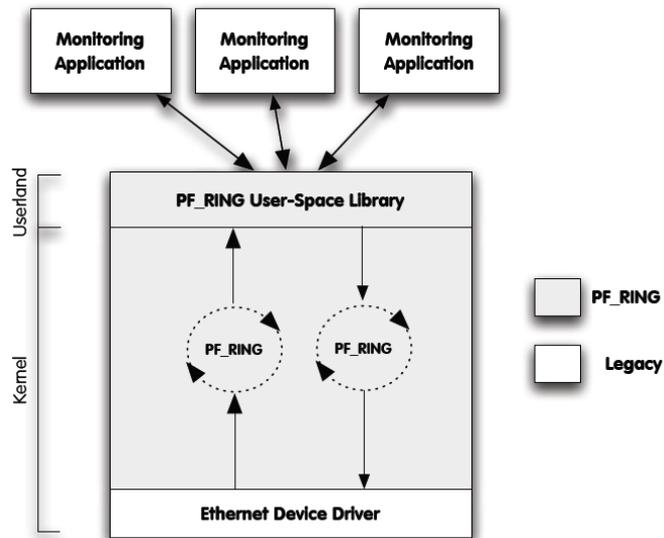
2.1. Что нового в руководстве пользователя PF_RING?

- Версия 5.5.2 (Июнь 2013)
 - Обновленное руководство в PF_RING версии 5.5.2
- Версия 5.4.6 (Авг 2012)
 - Обновленное руководство в PF_RING версии 5.4.6
- Версия 5.4.0 (Май 2012)
 - Обновленное руководство в PF_RING версии 5.4.0
 - Новая библиотека libzero с технологией zero-copу для гибкой обработки пакета напрямую с DNA.
- Версия 5.3.1 (Март 2012)
 - Обновленное руководство в PF_RING версии 5.3.1
- Версия 5.2.1 (Январь 2012)
 - Обновленное руководство в PF_RING версии 5.2.1
 - Новые функции API для управления аппаратными часами и временными метками.
 - Новые обратные вызовы модуля (расширения) ядра.

- Версия 5.1 (Сентябрь 2011)
 - Обновленное руководство в PF_RING версии 5.1.0
- Версия 4.7.1 (Июль 2011)
 - Обновленное руководство в PF_RING версии 4.7.1
 - Описаны PF_RING модульная библиотека и некоторые модули (DAG, DNA)
- Версия 4.6.1 (Март 2011)
 - Обновленное руководство в PF_RING версии 4.6.1
- Версия 4.6 (Февраль 2011)
 - Обновленное руководство в PF_RING версии 4.6.0.
- Версия 1.1 (Январь 2008)
 - Описана архитектура PF_RING модулей.
- Версия 1.0 (Январь 2008)
 - Первое руководство пользователя PF_RING.

3. Добро пожаловать в PF_RING

Архитектура PF_RING изображена на нижеприведённом рисунке.



Основными строительными блоками являются:

- Ускоренный модуль ядра, который обеспечивает низкоуровневое копирование пакета в кольца PF_RING.
- Пользовательский SDK для PF_RING, который обеспечивает прозрачную PF_RING-поддержку для пользовательских приложений.
- Специализированные PF_RING-ориентированные драйвера (дополнения), которые позволяют увеличить количество захваченных пакетов за счёт эффективного копирования пакетов из такого драйвера в PF_RING без прохождения через структуры данных ядра. Пожалуйста, имейте в виду, что PF_RING может работать с любыми драйверами сетевых карт (NIC), но для максимальной производительности используйте те специализированные драйвера, которые можно найти в директории kernel/ каталога PF_RING. Заметьте, что такая передача пакетов драйвером в PF_RING задана, когда модуль ядра PF_RING загружен вместе с параметром `transparent_mode`.

PF_RING реализует новый тип сокета (называемый PF_RING), через который приложение пользователя может взаимодействовать с модулем ядра PF_RING. Приложение может получить дескриптор PF_RING, и выполнить API вызовы, описанные ниже в данном руководстве. Дескриптор может быть связан с:

- Физическим сетевым интерфейсом.
- RX очередью, только на адаптерах с поддержкой нескольких очередей.

- Любым виртуальным интерфейсом, связанным с интерфейсом системы, который может принимать/отправлять пакеты.

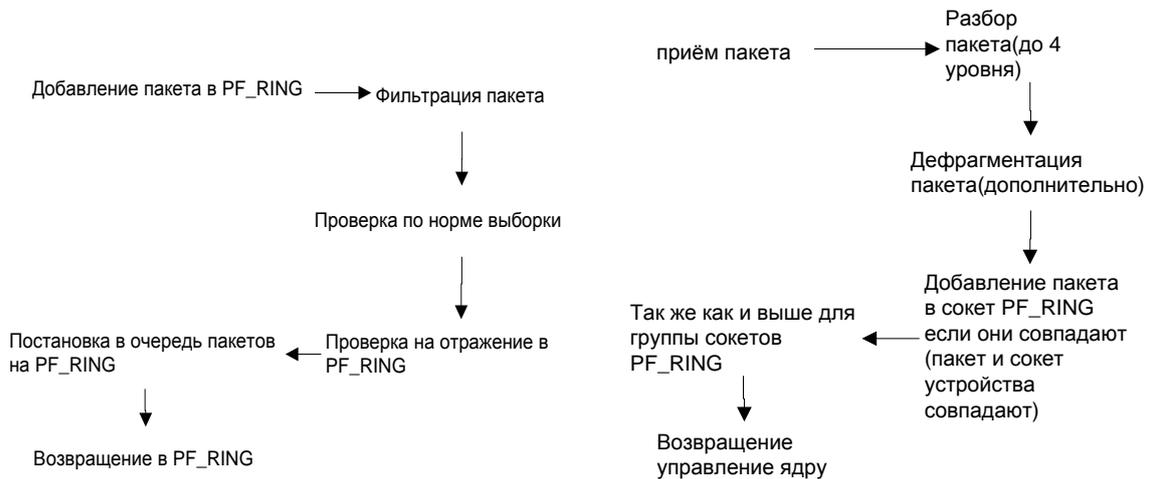
Как указано выше, пакеты читаются из памяти кольца выделенной на время его существования. Входящие пакеты копируются модулем ядра в кольцо и читаются приложением пользователя. Под каждый пакет не осуществляется выделение/освобождение памяти. Как только пакет был прочитан из кольца, используемое пространство в кольце для хранения пакета будет использовано для размещения будущих пакетов. Это означает, что приложения согласны, после прочтения, хранить пакеты у себя, так как PF_RING не будет сохранять их.

3.1. Фильтрация пакетов.

PF_RING поддерживает оба существующих BPF фильтра (т.е. те, которые поддерживаются в основанных на рсар приложениях, таких как tcpdump) и еще 2 дополнительных типа фильтров (конкретный и неточный фильтры, в зависимости от того, что задано – все элементы фильтра или только их часть), которые предоставляют разработчикам широкий выбор. Фильтры работают внутри модуля PF_RING, т.е. в ядре. Некоторые современные адаптеры, например основанные на сетевых контроллерах Intel 82599 или сетевые адаптеры компании Silicom, поддерживают аппаратную фильтрацию, которая также поддерживается PF_RING-ом через специальные вызовы API (напр. pfring_add_hw_rule). Фильтр PF_RING (за исключением аппаратных фильтров) может иметь специальную операцию, которая сообщит модулю ядра PF_RING, какое действие необходимо выполнить, когда данный пакет совпадет с фильтром. К действию относятся передача или отброс отфильтрованных пакетов приложению пользователя, отключение фильтрации или отражение пакета. В PF_RING, отражение пакета это способность передачи (не модифицированного) удовлетворяющему фильтру пакета на сетевой интерфейс (за исключением интерфейса на котором он был получен). Все операции, связанные с отражением, выполняются внутри модуля ядра PR_RING, за исключением ввода фильтра из приложения пользователя (без дальнейшей обработки пакетов).

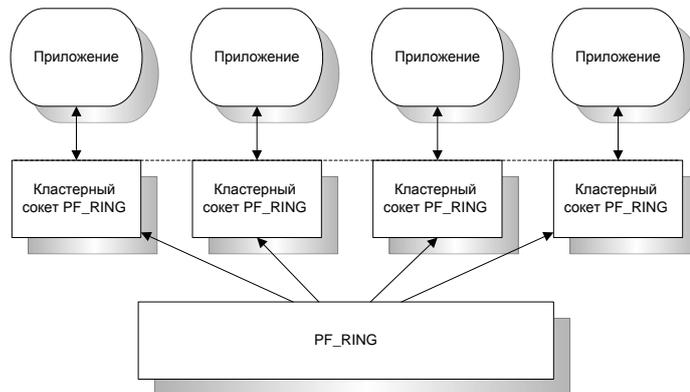
3.2. Путешествие пакета.

Перед тем как попасть в очередь кольца PF_RING, пакет проходит в нём достаточно большой путь.



3.3. Пакетная Кластеризация (Группирование).

PF_RING может увеличить производительность захвата пакетов приложениями с помощью реализации механизмов распределения и группирования. Эти механизмы позволяют приложениям, желающим разделять между собой множество пакетов для обработки, обрабатывать часть всего потока пакетов, в то время как все оставшиеся пакеты передаются другим членам группы. Это означает, что различные приложения, открывающие сокет PF_RING, могут привязать его к определённой группе с id (через `rfring_set_cluster`) для объединения сил. При этом каждый будет обрабатывать часть пакетов.



Все пакеты поделены между сокетами кластера указанные в политике кластера, которая может указать на конкретный поток (т.е. все пакеты, соответствующие полям данных `<proto, ip src/dst, port src/dst>`) что является по умолчанию, либо по принципу равномерного распределения пакетов на все приложения. Это означает, что если вы выбрали распределение по потокам, все пакеты принадлежащие потоку (т.е. определённый пятью полями) пойдут только приложению, работающему с этим потоком, хотя если указать общее правило, то все приложения будут получать поделенное поровну

количество пакетов, но нет гарантии, что пакеты, принадлежащие вышеупомянутой очереди (поток), будут получены единственным приложением. Так, с одной стороны, распределение по потокам позволяет вам сохранять логику приложения, так как в этом случае приложение получит подмножество из всех пакетов, но этот трафик будет последовательным. С другой стороны, если у Вас этот поток занимает большую часть трафика, тогда приложение, которое будет обрабатывать такой поток, будет перегружено пакетами и таким образом трафик будет не сбалансирован.

4. Семейство драйверов PF_Ring

Как уже отмечалось ранее, PF_RING может работать как напрямую со стандартными драйверами сетевых карт, так и напрямую со специальными драйверами. Модулю ядра PF_RING всё равно, но функциональность и скоростные показатели зависят от используемых драйверов.

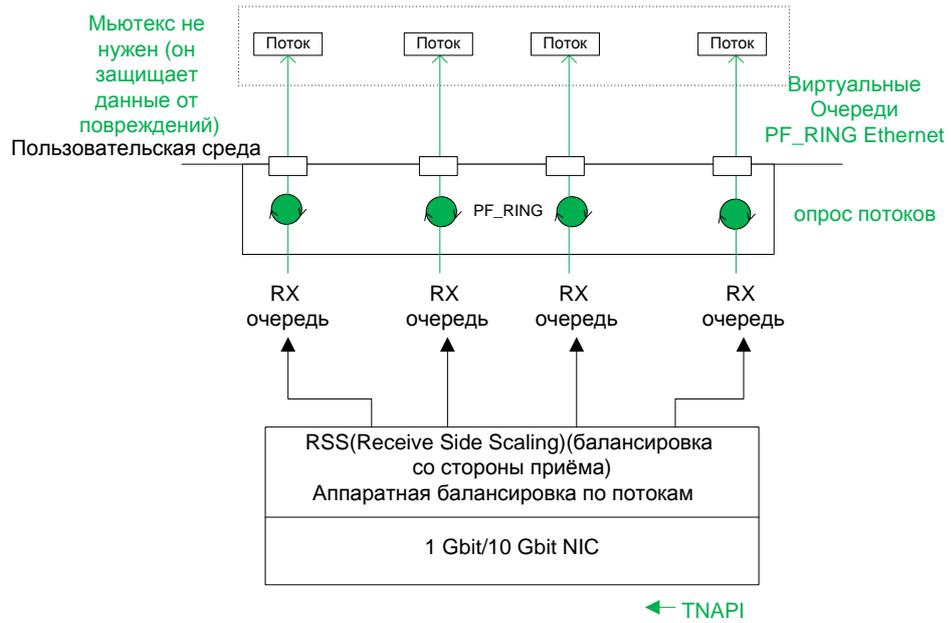
4.1. PF_RING-ориентированные драйвера.

Эти драйвера (находящиеся в PF_RING/driver/PF_RING-aware) разработаны для улучшенного захвата пакетов посредством проброса пакетов непосредственно в PF_RING без прохождения через стандартный механизм диспетчеризации пакетов Linux. С этими драйверами вы можете использовать `transparent_mode` (прозрачный режим) со значениями 1 или 2 (детали см. ниже в настоящем документе).

В дополнение к PF_RING-ориентированным драйверам, для некоторых конкретных адаптеров, можно использовать другие типы драйверов, которые ещё больше увеличат возможности захвата пакетов.

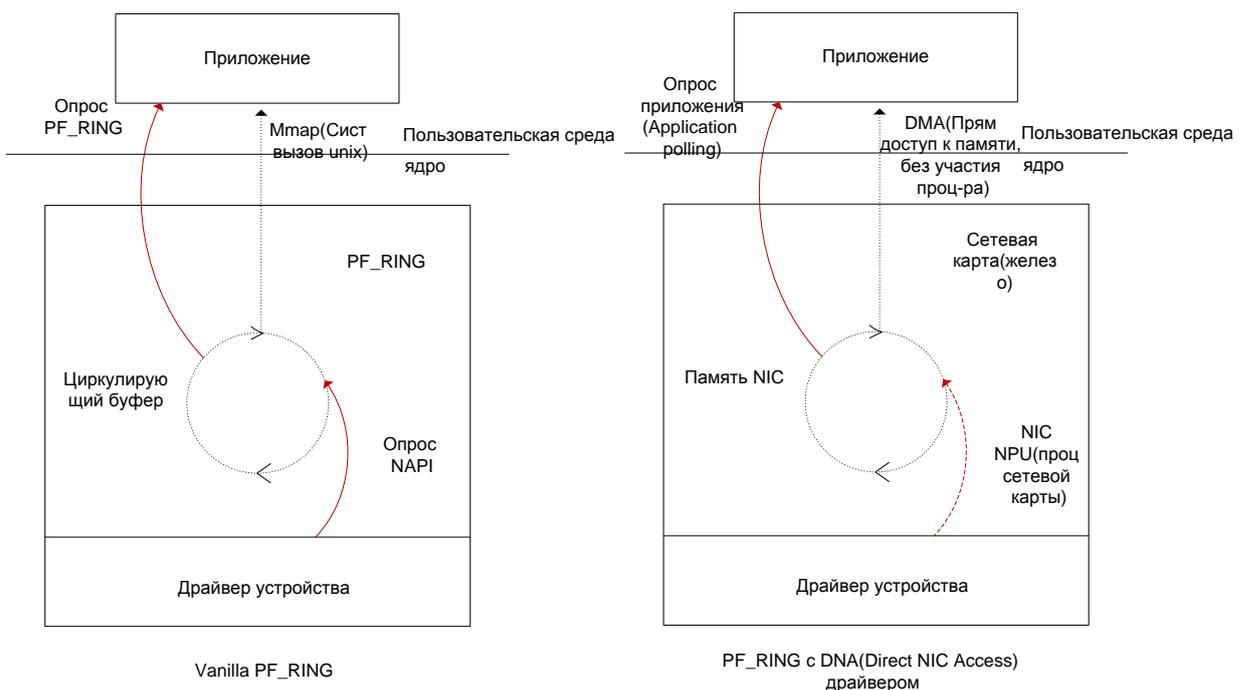
4.2. TNAPI

Первое семейство драйверов названо TNAPI (Threaded NAPI(поточные NAPI)), которое позволяет пакетам попасть в PF_RING более эффективно за счёт потоков ядра, инициированных непосредственно драйвером TNAPI. Драйвера TNAPI разработаны для улучшенного захвата пакетов, и поэтому они не могут быть использованы для передачи пакетов, так как канал передачи (TX) отключен.



4.3. DNA

Для тех пользователей, кому нужна максимальная скорость захвата пакетов с 0% загрузкой CPU для копирования пакетов на хост (т.е. механизм опроса (какой из колец свободен) NAPI не используется), это возможно используя различные типы драйверов, названные DNA ((Direct NIC access) прямой доступ к NIC), которые позволяют читать пакеты с сетевого интерфейса посредством одновременного обхода как ядра Linux, так и модуля PF_RING используя режим zero-copy (т.е. копирование без участия ядра и PF_RING - «напрямую»).



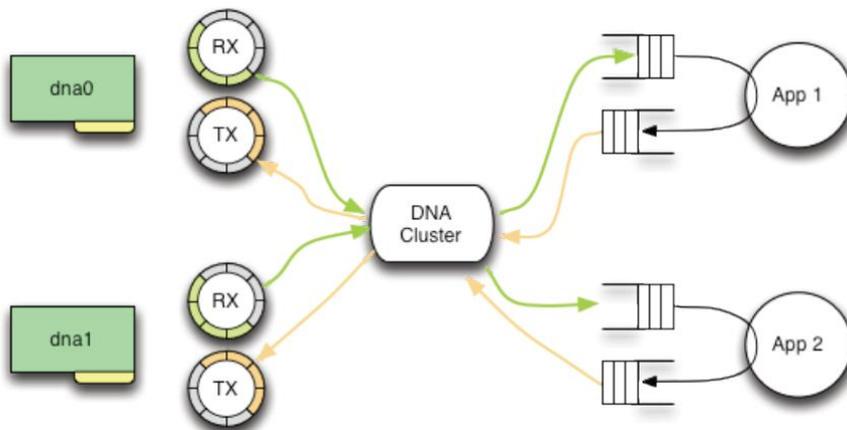
В DNA поддерживаются обе операции RX и TX. Так как ядро не задействовано, следующие функции PF_RING отсутствуют:

- Пакетная фильтрация в ядре (BPF и PF_RING фильтрация).
- Польза от плагинов PF_RING в ядре.

5. Библиотека Libzero для DNA.

Большинству приложений необходим расширенный подход к обработке пакетов. Начиная с версии PF_RING 5.4.0 была представлена библиотека под названием libzero, которая работает напрямую с низкоуровневым DNA-интерфейсом и осуществляет обработку пакетов по принципу zero-сору (не задействуя дополнительных элементов ОС). Libzero состоит из двух основных компонентов: DNA кластер и DNA проталкиватель.

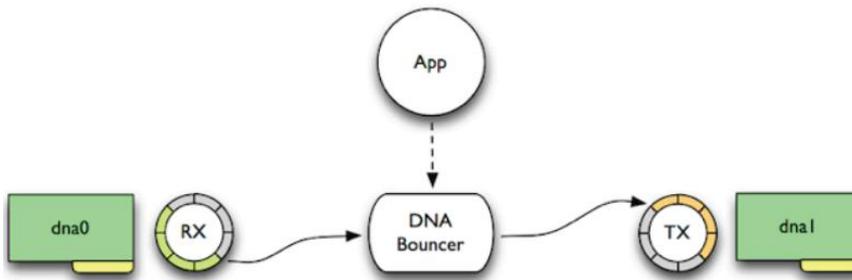
5.1. DNA кластер



DNA кластер осуществляет кластеризацию пакетов, так что все приложения, взаимосвязанные с одним и тем же кластером, смогут видеть приходящие пакеты, используя функцию гибкого распределения, и передавать все эти пакеты по принципу zero-сору. В сущности это нестандартная реализация RSS, которая позволяет распределять пакеты по очередям внутри сетевого адаптера. Кластеру можно присвоить различные управляющие функции фильтрации, раздачи и дублирования пакетов во множество потоков и приложений.

5.2. DNA проталкиватель

DNA проталкиватель коммутирует пакеты между двумя сетевыми интерфейсами по принципу zero-сору, предоставляя пользователю возможность задать функцию, которая может принимать решение, к каждому пакету, вне зависимости от того был ли полученный пакет перенаправлен или нет.



Продвижение данных осуществляется в одном направлении, но если вы хотите организовать мост, то нужно будет установить два проталкивателя (один на каждое направление).

6. Установка PF_RING

После загрузки PF_RING у вас будут следующие компоненты:

- SDK для PF_RING.
- Усовершенствованная версия библиотеки librsap, которая прозрачно работает с PF_RING-ом (если установлен), если не установлен, то библиотека работает в стандартном режиме.
- Модуль ядра PF_RING.
- PF_RING-ориентированные драйвера для разных чипсетов различных фирм-производителей.

Чтобы загрузить PF_RING с помощью SVN посмотрите здесь: <http://www.ntop.org/get-started/download/>.

Исходные коды PF_RING расположены в следующих местах:

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

Вы можете скомпилировать всё дерево командой `make` (как `normal`, `non-root`, `user`) из корневой директории.

6.1. Установка модуля ядра Linux.

Чтобы скомпилировать модуль ядра PF_RING вам необходимо иметь установленные заголовки ядра (или исходники ядра) Linux.

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

Обратите внимание, что:

- установка модуля ядра (с помощью `make install`) требует прав суперпользователя (`root`).
- В некоторых дистрибутивах Linux ядро поддерживает установку/сборку пакетов.
- В PF_RING 4.x вам БОЛЬШЕ НЕ НУЖНО делать заплату (патчить) ядра Linux как в предыдущих версиях PF_RING.

7. Запуск PF_RING

Перед использованием любого приложения PF_RING модуль ядра `pf_ring` должен быть загружен (суперпользователем):

```
# 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]
```

Примечание:

- `transparent_mode=0` (по умолчанию). Пакеты приняты через стандартный интерфейс Linux. Любой драйвер может использовать этот режим.
- `transparent_mode=1` (`vanilla` и PF_RING-ориентированные драйвера) Пакеты копируются ф-ей `memcpu()` в PF_RING, а также в стандартный тракт Linux.
- `transparent_mode=2` (только PF_RING-ориентированные драйвера) Пакеты копируются ф-ей `memcpu()` в PF_RING, исключая стандартный тракт Linux (т.е. `tcpdump` ничего не увидит).

Важно: НЕ ИСПОЛЬЗУЙТЕ `transparent_mode 1` и `2` с драйвером `vanilla` так как возможно не будут захватываться пакеты

Чем выше значение `transparent_mode`, тем быстрее будет захват пакетов.

Другие параметры:

- `min_num_slots` минимальное количество слотов кольца (по умолчанию - 4096).

PF_RING User's Guide v.5.5.3

- `enable_tx_capture` Установите значение «1» для захвата исходящих пакетов, установите значение «0», чтобы отключить захват исходящих пакетов (по умолчанию - RX+TX).
- `enable_ip_defrag` Установите значение «1» для включения IP дефрагментации, дефрагментация только rx трафика.
- `quick_mode` Установите значение «1» для запуска полной скорости, но устанавливается только на один сокет интерфейса.

7.1. Проверка конфигурации элемента PF_RING.

Когда PF_RING активизировалась, создается новая директория `/proc/net/pf_ring`.

```
# 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 позволяет пользователям устанавливать расширения (плагины) для обработки своего трафика. Эти плагины также регистрируются в дереве `pf_ring /proc`, их можно посмотреть в файле `plugins_info`.

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

7.2. Установка Libpfring и Libpcap

`libpfring` (пользовательский интерфейс библиотеки PF_RING) и `libpcap` поставляются в виде исходников. Их можно собрать следующим образом:

```
$ cd <PF_RING PATH>/userland/lib  
$ ./configure
```

```
$ make
$ sudo make install
$ cd ../libpcap
$ ./configure
$ make
```

Обратите внимание, что библиотека реентерабельна, поэтому его необходимо связать с вашим PF_RING-ориентированным приложением как и по аналогии с библиотекой – lpthread.

Важно

Устаревшие статически связанные основанные на pcap приложения необходимо пересобрать заново с новой поддерживающей PF_RING библиотекой libpcap.a, чтобы получить преимущества PF_RING. Не надейтесь использовать PF_RING без пересборки вашего существующего приложения.

Если вы обладаете новым PF_RING, вы можете начать с некоторых примеров. Каталог “userland/examples” богат готовыми для использования приложениями PF_RING:

```
$ 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
```

Например, pfcount позволяет вывести некоторую статистику принятых пакетов:

```
# ./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]
=====
```

Другой пример это `pfsend`, который позволяет вам отправлять пакеты (сгенерированные пакеты, или из файла `.pcap`) с заданной скоростью:

```
# ./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. Дополнительные модули PF_RING

В версии 4.7, PF_RING имеет новую модульную архитектуру, позволяющую использовать дополнительные компоненты отличные от стандартного модуля ядра PF_RING. Эти компоненты собраны в соответствующей библиотеке, которая инициализируется конфигурационным скриптом. В настоящее время, в набор включены следующие дополнительные модули:

- модуль DAG. Этот модуль добавляет в PF_RING поддержку карт Endace DAG (компания, специализирующаяся на записи трафика в высокопроизводительных сетях).
- модуль DNA. Этот модуль можно использовать для использования устройства в DNA режиме, если ваша карта поддерживает DNA драйвер. Пожалуйста, имейте в виду, что модуль ядра PF_RING должен быть загружен до драйвера DNA. С DNA вы можете существенно увеличить скорость захвата и передачи пакетов за счет обхода ядра, то есть приложения могут общаться с драйвером напрямую. В настоящее время доступны следующие DNA-ориентированные драйвера:
 - `e1000e`
 - `igb`
 - `ixgbe`

Драйвера являются частью дистрибутива PF_RING и их можно найти в `drivers/DNA/`. Всеми драйверами вы можете успешно измерить проводную скорость при любом размере пакета, для RX и TX. Приложением `Pfcount` вы можете протестировать RX, а TX - используя приложение `pfsend`.

Заметьте, в случае TX скорость отправки может ограничиваться производительностью RX(приема). Это потому, что если приемник не может постоянно поддерживать высокую скорость захвата (в случае наплыва пакетов возможна перегрузка буфера), то сетевая плата Ethernet (приёмника) посылает кадры Ethernet PAUSE обратно к отправителю, что замедляет отправителя (для разгрузки буфера приёмника). Если вы хотите игнорировать такие кадры и

таким образом отправлять пакеты с полной скоростью, вам нужно отключить автосогласование (autonegotiation) игнорируя их (ethtool -A dnaX autoneg off rx off tx off).

- **Модуль объединения каналов**
Этот модуль может быть использован для агрегирования нескольких интерфейсов для захвата пакетов с этих интерфейсов, которые имеют единственный общий сокет PF_RING. Например, это возможно, если открыть кольцо с именем устройства "multi:ethX;ethY;ethZ".
- **Модуль для создания RING в пользовательском пространстве**
Этот модуль позволяет приложению отправлять пакеты другому приложению используя стандартные API PF_RING через создание виртуального устройства (например usrX, где X это уникальный идентификатор RING в пользовательском пространстве). Чтобы сделать это, отправляющее приложение должно открыть кольцо, используя имя устройства "userspace:usrX" (где "userspace:" определяет модуль для создания PF_RING в пользовательском пространстве), в то время как записывающее приложение должно открыть кольцо стандартным образом, используя для этого имя устройства "usrX".
- **Модуль-пользователь ("dnacluster") библиотеки Libzero.**
Этот модуль можно использовать для подключения к DNA кластеру, позволяющий приложению отправлять и принимать пакеты, используя стандартные API PF_RING. Отправляющее приложение должно открыть кольцо, используя имя устройства "dnacluster:X@Y", где X – идентификатор кластера, Y – идентификатор потребителя, или "dnacluster:X" для авто-назначения пользовательского идентификатора.
- **Модуль-инъекция стека TCP/IP Linux.**
Этот модуль может быть использован для инъекции/захвата пакетов в/из стек TCP/IP, симулируя прибытие/отправку этих пакетов на интерфейс. Приложение должно открыть кольцо с помощью имени устройства «stack:dnaX», где dnaX интерфейс, связанный с пакетами вводимые в стек. В случае ввода пакета в стек, должна использоваться функция pfring_send(), в случае захвата исходящих пакетов должна использоваться функция pfring_recv().

8. PF_RING для разработчиков приложений

Концептуально PF_RING является простой, но мощной технологией, которая позволяет разработчикам создавать высокопроизводительные приложения мониторинга трафика и его разбора за короткое время. Это потому, что PF_RING ограждает разработчика от внутренних тонкостей ядра, которые обрабатываются библиотекой и драйвером ядра. Это позволяет разработчику существенно сэкономить время и сосредоточиться на разрабатываемом приложении, не обращая внимания на то, каким образом пакеты принимаются и передаются.

Эта глава содержит:

- API PF_RING.
- Расширения библиотеки `libpcap` для поддержки старых приложений.

8.1. API PF_RING

Внутренняя структура данных PF_RING должна быть скрыта от пользователя, он может манипулировать пакетами и устройствами только с помощью доступных API определенных в файле `pfring.h` входящий в состав PF_RING.

8.2. Возвращаемые коды

По традиции, библиотека возвращает отрицательные значения при ошибках и исключениях. Неотрицательные коды означают успех. В случае если возвращаемое значение другое, тогда оно будет в описании соответствующей функции.

8.3. Согласование имен устройств PF_RING

Имена устройств в PF_RING такие же, как и в `libpcap` и `ifconfig`. Так `eth0` и `eth5` соответствуют именам, которые вы можете использовать в PF_RING. Вы можете определить виртуальные устройства названные "как угодно", что позволяет PF_RING-у захватывать пакеты со всех возможных сетевых устройств.

Как ранее объяснялось, с PF_RING вы можете использовать как драйвера идущие с вашим Linux дистрибутивом (которые не специализируются на PF_RING), так и некоторые PF_RING-ориентированные драйвера (вы можете их найти в директории `PF_RING drivers/`) которые продвигают пакеты PF_RING эффективнее, чем `vanilla` драйвера. Если вы владеете современной, с несколькими очередями, сетевой картой работающей с PF_RING-ориентированным драйвером (напр. адаптер Intel 10 Gbit), то PF_RING позволит вам захватывать пакеты со всего устройства (т.е. при захвате пакетов PF_RING не обращает внимания на очередь RX на которой пакет был получен, к примеру `ethX`) или из конкретно указанной очереди (напр. `ethX@Y`). Предположим, имеется адаптер с количеством очередей `Z`, очередь с `id Y`, она должна быть в диапазоне `0..Z-1`. В случае если вы укажете очередь, которая не попадает в диапазон, то пакеты перехватываться не будут.

Как указывалось в предыдущей главе, PF_RING 4.7 имеет модульную структуру. Чтобы мы смогли использовать какой-нибудь модуль библиотеки, нужно присоединить имя модуля к имени устройства, разделённые двоеточием (например `dna:dnaX@Y` для модуля `dna`, `dag:dagX@Y` для модуля `dag`, `"multi:ethA@X;ethB@Y;ethC@Z"` для модуля при агрегировании каналов, `"dnacluster:A@X"` для модуля-пользователя кластера).

8.4. PF_RING: инициализация сокета

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

Этот вызов используется для инициализации сокета PF_RING, на выходе функции получаем дескриптор структуры типа `pfring`, который может быть использован в последующих вызовах. Обратите внимание, что:

- Вы можете использовать физическое (напр. ethX) и виртуальное (напр. tapX) устройства, RX-очереди (напр. ethX@Y), и дополнительные модули (напр. dna:dnaX@Y, dag:dagX:Y, "multi:ethA@X;ethB@Y;ethC@Z", "dnacluster:A@X").
- Вам нужны права суперпользователя для открытия устройства.

Входные параметры:

device_name

Символическое имя PF_RING-ориентированного устройства, которое мы открываем (напр. eth0).

caplen

максимальная длина захватываемого пакета (также известна как snaplen).

flags

флаги позволяют использовать некоторые опции, которые должны быть указаны в компактном формате используя битовый образ (bitmaps):

- PF_RING_REENTRANT
Устройство открывается в совместно-используемом режиме. Это осуществляется путем семафоров, в результате производительность ухудшается. Используйте совместно-используемый режим только для многопоточных приложений.
- PF_RING_LONG_HEADER
Если не устанавливать (этот флаг), PF_RING не заполнит поле extended_hdr структуры pfring_pkthdr. Если установить, поле extended_hdr правильно заполнится. В случае если вам не нужна расширенная информация, установите значение 0 для ускорения операции.
- PF_RING_PROMISC
Устройство открыто в прослушивающем режиме.
- PF_RING_DNA_SYMMETRIC_RSS
Устанавливает аппаратную функцию RSS в симметричный режим (оба направления (RX и TX) одного и того же потока идут в одной аппаратной очереди). Поддерживается только в DNA драйверах. Эта опция так же доступна в PF_RING – ориентированной библиотеке libpcap через переменную среды PCAP_PF_RING_DNA_RSS.
- PF_RING_TIMESTAMP
Принуждает PF_RING указать временную метку на прием пакетов (обычно это не указывается когда используется zero-copy, для оптимизации производительности).
- PF_RING_HW_TIMESTAMP
Включает аппаратные временные метки, если возможно.
- PF_RING_DNA_FIXED_RSS_Q_0
Задаёт аппаратную RSS для отправки всего трафика на 0 очередь. Можно задать другие очереди, используя аппаратные фильтры (для карт DNA с поддержкой аппаратного фильтра).
- PF_RING_STRIP_HW_TIMESTAMP
Снять аппаратную временную метку у пакета.

- PF_RING_DO_NOT_PARSE
Отключить разбор пакета кроме того, когда используется 1-сору.
- PF_RING_DO_NOT_TIMESTAMP
Отключить временную метку пакета, кроме того, когда используется 1-сору.

Возвращаемое значение:

В случае успеха возвращается дескриптор, в противном случае NULL.

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

Эта функция подобна `pfring_open()` за исключением случая с картой с несколькими RX-очередями, вместо открытия одного кольца для всего устройства, открываются отдельные кольца (одно на каждую RX-очередь)

Входные параметры:

`device_name`

Символическое имя PF_RING- ориентированного устройства, которое мы открываем (напр. `eth0`). Нет какого-то определяющего очередь имени, это просто основное имя устройства.

`caplen`

максимальная длина захватываемого пакета (также известна как `snaplen`).

Flags

Смотрите `pfring_open()` для подробностей

Ring

Указатель на массив колец, который будет содержать указатели на открытые кольца.

Возвращаемое значение:

Последний индекс массива колец, который содержит указатель на открытое кольцо.

8.5. PF_RING: Отключение устройства

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

Эта функция используется для закрытия устройства PF_RING, которое ранее было открыто. Заметьте, что вы должны постоянно закрывать устройство до завершения приложения. Если неуверенны, вы можете закрыть устройство через обработчик сигналов.

Входные параметры:

Ring

Дескриптор PF_RING, который мы собираемся закрыть.

8.6. PF_RING: Чтение входящих пакетов

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

Эта функция возвращает вновь поступивший пакет.

Входные параметры:

Ring

Дескриптор PF_RING, с которого мы будем проводить регистрацию.

Buffer

Отведенная вызывающим абонентом область памяти, где сохраняется вновь поступивший пакет. Обратите внимание, что этот параметр указатель на указатель, чтобы реализовать механизм zero-copy (значение buffer_len должно быть 0).

buffer_len

Длина области памяти. Заметьте, что входящий пакет выбрасывается, если он слишком длинный для заданной области памяти. Когда можно реализовать оптимизацию через zero-copy установите значение 0.

hdr

Область памяти, куда заголовок пакета будет скопирован.

wait_for_incoming_packet

Если 0, мы просто проверяем доступность пакета, в противном случае функция блокируется пока пакет доступен. Эта возможность также доступна через переменную окружения PCAP_PF_RING_ACTIVE_POLL PF_RING-ориентированной библиотеки libpcap.

Возвращаемое значение:

0 в случае если ни один пакет не получили (без блокирования), 1 в случае успеха, -1 в случае ошибки.

```
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)
```

Тоже что и pfring_recv(), только с дополнительными параметрами для улучшенного разбора пакета.

Входные параметры не присутствующие в pfring_recv():

Level

Уровень заголовка, где заканчивать разбор пакета.

Add_timestamp

Добавляем временную метку.

Add_hash

Вычисленный hash двух ip адресов (источник-назначение).

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

Эта функция обрабатывает пакеты либо до вызова pfring_breakloop() либо до ошибки.

Входные параметры:

Ring

Дескриптор PF_RING.

Looper

Обратный вызов, вызываемый для каждого принятого пакета. Параметры, передаваемые этим методом: указатель на структуру pfring_pkthdr, указатель на память пакета, и указатель на user_bytes.

User_bytes

Указатель на пользовательские данные, которые переданы функцией обратного вызова.

wait_for_packet

если активен 0, ожидаем, чтоб проверить доступность пакетов.

Возвращаемое значение:

Неотрицательное число, если функция pfring_breakloop() вызвана. В случае ошибки - отрицательное число.

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

Эта функция возвращает время прибытия входящего пакета, когда это возможно.

Входные параметры:

Ring

Дескриптор PF_RING, над которым мы выполняем проверку.

Ts

Структура, где должно быть записано время.

Возвращаемое значение:

В случае успеха - 0, в противном случае - отрицательное число.

```
int pfring_next_pkt_raw_timestamp(pfring *ring, u_int64_t *timestamp_ns)
```

Эта функция возвращает исходную временную метку следующего входящего пакета, когда это возможно. Это возможно только с адаптерами, поддерживающими гх аппаратную временную метку.

Входные параметры:

ring

Дескриптор PF_RING над которым мы выполняем проверку.

timestamp_ns

Где временная метка будет записана.

Возвращаемое значение:

В случае успеха - 0, в противном случае - отрицательное число.

8.7. PF_RING: Кластерное кольцо

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

Эта функция позволяет кольцу добавиться в кластер, который будет доступен в адресном пространстве. В двух словах, когда 2 или более сокета кластеризованы (сгруппированы), то они делят входящие пакеты, которые распределяются на потоки. Этот приём полезно использовать в многоядерных системах для распределения пакетов в одном и том же адресном пространстве посредством многопоточности.

Кластеризация также доступна в PF_RING-ориентированной библиотеке libpcap посредством переменной окружения PCAP_PF_RING_CLUSTER_ID (на все потоки по умолчанию, или через переменную окружения PCAP_PF_RING_USE_CLUSTER_PER_FLOW на конкретный поток).

Входящие параметры:

Ring

Дескриптор PF_RING добавляемый в кластер

clustered

идентификатор кластера, с которым кольцо будет связано.

the_type

тип кластера (поток определяется по 2-м, 4-м или 5-и полям, для tcp только по 5-и и 6-и или на все потоки).

Возвращаемое значение:

В случае успеха 0, в противном случае отрицательное число.

```
int pfring_remove_from_cluster(pfring *ring);
```

Эта функция позволяет кольцу удалиться из кластера.

Входящие параметры:

Ring

Дескриптор PF_RING кластера.

clusterId

идентификатор кластера, с которым кольцо связано.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное значение.

8.8. PF_RING: Зеркалирование пакетов

Зеркалирование пакетов - это способность пробрасывать пакеты в ядро, без их отправки в пользовательское пространство и обратно. Вы можете определить зеркалирование пакетов внутри правил фильтрации.

```
typedef struct {  
...  
char reflector_device_name[REFLECTOR_NAME_LEN];  
...  
} filtering_rule;
```

В `reflector_device_name` укажите имя вашего устройства (напр. `eth0`), на которое пакеты соответствующие фильтру будут зеркалироваться. Убедитесь, что ваше отражающее устройство имеет отличное имя от имени устройства, которое захватывает пакеты, в противном случае у вас образуется петля пакетов.

8.9. PF_RING: Выборка пакетов

```
int pfring_set_sampling_rate(pfring *ring, u_int32_t rate)
```

Выборка пакетов реализуется непосредственно в ядре. Заметьте, что это решение намного эффективнее, чем реализация отбора в пользовательском пространстве. Отбираются только те пакеты, которые проходят через все фильтры (если они есть).

Входящие параметры:

Ring

Дескриптор PF_RING, по которому применяется выборка.

Rate

Частота выборки. Скорость X означает, что 1 пакет из X перенаправляется. Это значит, что при скорости 1 происходит отключение выборки.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное число.

8.10. PF_RING: Фильтрация пакетов

PF_RING позволяет фильтровать пакеты двумя способами: конкретным (она же хэш-фильтрация) фильтром или неполным фильтром. Конкретную фильтрацию используют, когда необходимо отследить конкретное соединение по 6-ти полям `<vlan Id, protocol, source IP, source port, destination IP, destination port>`. На неполную фильтрацию переходят всякий раз, когда фильтр может иметь только некоторые из этих полей (например, все совпадающие по UDP признаку пакеты, не смотря на их адресаты). Если значение какого-нибудь поля 0, то оно не примет участия в фильтрации.

8.10.1. PF_RING: Неполная фильтрация

```
int pfring_add_filtering_rule(pfring *ring, filtering_rule* rule_to_add)
```

Добавляет правило фильтрации существующему кольцу. Каждое правило имеет уникальный id только в пределах своего кольца (т.е. 2 кольца могут иметь правила с одинаковыми id).

Входные параметры:

Ring

Дескриптор PF_RING, в котором будет действовать добавленное правило.

Rule_to_add

Правило добавляется, как описано в последней главе этого документа.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_remove_filtering_rule(pfring *ring, u_int16_t rule_id)
```

Удаляет ранее добавленное правило фильтрации.

Входные параметры:

Ring

Дескриптор PF_RING, из которого правило будет удалено.

Rule_id

Это id ранее добавленного правила, которое будет удалено.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное число (напр. если правило не существует).

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

Считывание статистики по правилу фильтрации.

Входные параметры:

Ring

Дескриптор PF_RING, из которого будут считываться статистику.

Rule_id

Id правила, которое определяет правило для считывания статистики.

Stats

Буфер, выделенный для пользователя, который будет содержать статистику правила. Пожалуйста, убедитесь, что буфер достаточно большой для

статистики. Такой буфер будет содержать количество принятых и отброшенных пакетов.

Stats_len

Размер буфера в байтах.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное значение (напр. правило не существует)

```
int pfring_purge_idle_rules(pfring *ring, u_int16_t inactivity_sec);
```

Удаляет правила фильтрации неактивные в течение определенного количества секунд.

Входные параметры:

Ring

Дескриптор PF_RING, из которого правила будут удалены.

inactivity_sec

Предел бездействия.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное значение.

8.10.2. PF_RING: Конкретная фильтрация

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

Добавляет или удаляет правило фильтрации.

Входные параметры:

Ring

Дескриптор PF_RING, в котором правило будет добавлено/удалено.

Rule_to_add

Правило, которое будет добавлено/удалено, как описано в последней главе этого документа. Все параметры правила фильтрации должны быть заданы (без масок).

Add_rule

Если установить положительное значение, то правило добавится, если 0 правило удалится.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение (напр. удаляемое правило не существует).

Все параметры правила должны быть заданы в правиле фильтрации (полностью).

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

Считает статистику по правилу фильтрации.

Входные параметры:

Ring

Дескриптор PF_RING, с которого будет считываться статистика.

Rule

Правило, по которому считается статистика. Это должно быть то же правило, которое было ранее добавлено.

Stats

Буфер, выделенный для пользователя, который будет содержать статистику правила. Пожалуйста, будьте уверены, что буфер достаточно большой для статистики. Такой буфер будет содержать количество принятых и отброшенных пакетов.

Stats_len

Размер буфера в байтах.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение (напр. правило не существует).

```
int pfring_purge_idle_hash_rules(pfring *ring, u_int16_t inactivity_sec);
```

Удаляет правила фильтрации неактивные в течение определенного количества секунд.

Входные параметры:

Ring

Дескриптор PF_RING, из которого правило будет удалено.

inactivity_sec

Предел бездействия.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

8.10.3. PF_RING: BPF фильтрация

Как и в версии 5.1, возможно указать BPF фильтры через PF_RING API. Для этого необходимо подключить (по умолчанию включено) BPF поддержку на время компиляции и соединить PF_RING-реализуемые приложения с библиотекой libpcap (возможно отключить поддержку BPF с помощью команд "cd userland/lib/; ./configure --disablebpf; make" чтобы избежать подключения к библиотеки libpcap).

```
int pfring_set_bpf_filter(pfring *ring, char* filter_buffer)
```

Задаёт BPF фильтр существующему кольцу.

Входящие параметры:

Ring

Дескриптор PF_RING, к которому будет добавлен фильтр.

Filter_buffer

Задаём фильтр.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_remove_bpf_filter(pfring *ring)
```

Удаляет BPF фильтр.

Входные параметры:

Ring

Дескриптор PF_RING.

Выходное значение:

0 - в случае успеха, в противном случае - отрицательное число.

8.11. PF_RING: Фильтрация пакетов в NIC

Некоторые, имеющие несколько очередей, современные сетевые адаптеры предоставляют возможности «управления пакетами». С помощью них можно указать сетевой карте NIC отправлять отобранные пакеты определённой RX очереди. Если у указанной очереди id превышает максимум queueid, то пакет отбрасывается подобно работе аппаратного фаервола.

Примечание: Фильтрации пакетов в ядре не поддерживается DNA.

```
int pfring_add_hw_rule(pfring *ring, hw_filtering_rule *rule)
```

Задаёт правила фильтрации в NIC. Заметьте, добавляем не PF_RING фильтр, а именно NIC фильтр.

Входные параметры:

Ring

Дескриптор PF_RING, в который будет добавлено правило.

Rule

Правило фильтрации, которое устанавливается в NIC, как показано в последней главе данного документа. Все параметры правила должны быть описаны, и если указано значение 0, тогда эти параметры не примут участия в фильтрации.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение (напр. добавленное правило имело неверный формат или NIC, которая работает с этим кольцом, не поддерживает аппаратной фильтрации).

```
int pfring_remove_hw_rule(pfring *ring, hw_filtering_rule *rule)
```

Удаляет указанное правило фильтрации из NIC.

Входные параметры:

Ring

Дескриптор PF_RING, из которого правило будет удалено.

Rule

Правило фильтрации, которое будет удалено из NIC.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное число.

```
int pfring_set_filtering_mode(pfring *ring, filtering_mode mode)
```

Задаёт режим фильтрации (только программный, только аппаратный, аппаратный и программный) для того, чтобы прозрачно добавлять/удалять аппаратные правила с помощью того же API функционала, используемого для программных правил (неполных или конкретных).

Входные параметры:

Ring

Дескриптор PF_RING, для которого будет задан режим.

Mode

Режим фильтрации.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное число.

8.12. PF_RING: Политика фильтрации

```
int pfring_toggle_filtering_policy(pfring *ring, u_int8_t rules_default_accept_policy)
```

Устанавливает дефолтную (по умолчанию) политику фильтрации. Это значит, что если нет правила удовлетворяющего входящему пакету, то дефолтная политика решает, отбросить пакет или направить в пользовательское пространство. Обратите внимание, что каждое фильтрующее правило действует в пределах кольца, так что кольца могут иметь различные правила или политики по умолчанию.

Входные параметры:

Ring

Дескриптор PF_RING, в котором происходит переключение дефолтной политики.

rules_default_accept_policy

Если указать положительное значение, то дефолтная политика будет принята (т.е. передача пакетов в пространство пользователя), при отрицательном значении происходит отбрасывание пакетов, не удовлетворяющих фильтру.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

8.13. PF_RING: Передача пакетов

В зависимости от используемого драйвера, передача пакета может различаться:

- Vanilla и PF_RING ориентированные драйвера: PF_RING не ускоряет передачу, сравнивая с используемыми стандартными средствами передачи Linux. Не ожидайте преимущества в скорости, когда используете PF_RING в этом режиме.
- TNAPI: передача пакетов не поддерживается.
- DNA: передача пакетов поддерживается.

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

Несмотря на то, что PF_RING оптимизирован для приёма(RX), он также позволяет отправлять пакеты(TX). Эта функция позволяет отправлять сырой пакет (т.е. пакет передается по линии такой, каким был сформирован). Этот пакет должен быть полностью сформирован (вплоть до MAC адреса) и должен быть передан как есть, без каких-либо дополнительных обработок.

Входные параметры:

Ring

Дескриптор PF_RING, с которого пакет будет отправлен.

Pkt

Буфер, содержащий отправляемый пакет.

Pkt_len

Длина буфера pkt.

flush_packet

1=незамедлительная отправка очереди. Если указать 0, вы уменьшите нагрузку на CPU, но пакеты встанут в (общую) очередь и будут передаваться с большей задержкой.

Возвращаемое значение:

В случае успеха количество переданных байтов, в противном случае отрицательное число.

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

Схожа с pfring_send(), с возможностью определять индекс исходящего сетевого интерфейса.

Входные параметры не представленные в pfring_send():

If_index

Индекс интерфейса соответствующий сетевому устройству (с которого отправится пакет).

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

Эта функция позволяет отправить пакет, и вернуть точное время (ns), в которое он был отправлен в линию связи. Заметьте, это возможно только когда адаптер поддерживает передачу аппаратных временных меток при отправке (tx), и они могут повлиять на производительность.

Входные параметры, не представленные в pfring_send():

Ts

Структура, куда временная метка отправки пакета будет записана.

```
int pfring_send_last_rx_packet(pfring *ring, int tx_interface_id)
```

Отправляет последний принятый пакет на указанное устройство. Эта оптимизация, работающая только в стандартном PF_RING.

Входные параметры:

Ring

Дескриптор PF_RING, через который пакет принимается.

tx_interface_id

id интерфейса отправляющего пакет.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

8.14. PF_RING: Прочие функции

```
int pfring_enable_ring(pfring *ring)
```

Когда кольцо создано, оно не включено (т.е. входящие пакеты отбрасываются) пока эта функция не вызвана.

Входные параметры:

Ring

Дескриптор PF_RING, который нужно включить.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение (напр. кольцо не может быть включено).

```
int pfring_disable_ring(pfring *ring)
```

Отключить кольцо.

Входные параметры:

Ring
Дескриптор PF_RING для отключения.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_stats(pfring *ring, pfring_stat *stats)
```

Чтение статистики кольца (принятые и отброшенные пакеты).

Входные параметры:

Ring
Дескриптор PF_RING, с которого будет читаться статистика.
Stats
Буфер пространства пользователя, в который будет сохранена статистика (количество принятых и отброшенных пакетов).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_application_stats(pfring *ring, char *stats)
```

Задать статистику пользовательского приложения.

Входные параметры:

Ring
Дескриптор PF_RING.
Stats
Статистика приложения.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное значение.

```
char* pfring_get_appl_stats_file_name(pfring *ring, char *path, u_int path_len)
```

Вернуть имя файла, где можно прочесть статистику по приложению.

Входные параметры:

Ring
Дескриптор PF_RING.
path

Пользовательский буфер, в который будет записана статистика из файла.

Path_len

Длина path.

Возвращаемое значение:

В случае успеха - имя файла, в противном случае - NULL.

```
int pfring_version(pfring *ring, u_int32_t *version)
```

Читает версию кольца. Заметьте, если версия кольца 3.7, то будет возвращено значение 0x030700.

Входные параметры:

Ring

Дескриптор PF_RING для считывания версии.

Version

Буфер пространства пользователя, в который будет скопирована версия ядра.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_direction(pfring *ring, packet_direction direction)
```

Указывает PF_RING принимать во внимание только те пакеты, которые соответствуют указанному направлению. Если приложение не вызывает эту функцию, тогда возвращаются все пакеты (независимо от направления, TX или RX).

Входные параметры:

Ring

Дескриптор PF_RING.

Direction

Направления пакетов (RX, TX или RX и TX).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_socket_mode(pfring *ring, socket_mode mode)
```

Сообщает PF_RING-у, что приложению необходимо отправить и/или принять пакеты в/из сокета.

Входные параметры:

Ring

Дескриптор PF_RING.

Mode

Режим сокета (отправка, прием или отправка и приём).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_poll(pfring *ring, u_int wait_duration)
```

Выполняет пассивное ожидание на PF_RING сожете, схожая со стандартной ф-ей poll(), отвечающая за синхронизацию структур данных.

Входные параметры:

Ring
Сокет PF_RING для прослушивания.
Wait_duration
Время прослушивания в миллисекундах.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_poll_watermark(pfring *ring, u_int16_t watermark)
```

Всякий раз, когда приложение пользователя вынуждено ожидать прибытия входящих пакетов, оно может задать PF_RING-у не возвращать функцию poll(), по крайней мере, пока пакеты "watermark" не вернулись. Низкое значение watermark, такое как 1, уменьшает время ожидания poll(), но вероятно, увеличивается количество вызовов ф-ции poll(). Высокое значение watermark (оно не может превышать 50% от размера кольца, в противном случае модуль ядра PF_RING будет превышать своё значение), вместо уменьшения количества вызовов ф-ции poll() немного увеличится время ожидания пакета. Значение по умолчанию watermark (т.е. если приложения пользователя не изменяют значение watermark через эту функцию) – 128.

Входные параметры:

Ring
Дескриптор PF_RING.

Watermark
Прослушивание пакета watermark.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_tx_watermark(pfring *ring, u_int16_t watermark)
```

Задаёт число пакетов, которое должно быть поставлено в очередь перед её отправкой по линии связи.

Входные параметры:

Ring
Дескриптор PF_RING.
Watermark
Tx watermark.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_poll_duration(pfring *ring, u_int duration)
```

Задаёт время ожидания poll, когда используется пассивное ожидание.

Входные параметры:

- Ring
Дескриптор PF_RING, который нужно включить.
- Duration
Время ожидания poll в миллисекундах.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_set_application_name(pfring *ring, char *name)
```

Сообщает PF_RING-у имя приложения (обычно задается через argv[0]), которое использует это кольцо. Эта информация используется для того, чтобы идентифицировать приложение, при доступе к файлам директории PF_RING каталога /proc файловой системы. Это также возможно с помощью PF-RING- ориентированной библиотеки libpcap через переменную окружения PCAP_PF_RING_APPNAME.

Пример:

```
> cat /proc/net/pf_ring/16614-eth0.0
Bound Device : eth0
Slot Version : 13 [4.7.1]
Active : 1
Sampling Rate : 1
Appl. Name : pfcoun
IP Defragment : No
....
```

Входные параметры:

- Ring
Дескриптор PF_RING.
- Name
Имя приложения, использующее это кольцо.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

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

Возвращает MAC-адрес устройства, связанного с сокетом.

Входные параметры:

- Ring
Дескриптор PF_RING-а, который будет делать запрос.
- Mac_address

Массив, в который будет скопирован MAC-адрес.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int pfring_get_bound_device_id(pfring *ring, int* device_id)
```

Возвращает id устройства связанного с сокетом.

Входные параметры:

Ring

Дескриптор PF_RING-а, который будет делать запрос.

device_id

Куда будет скопирован идентификатор устройства.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
u_int8_t pfring_get_num_rx_channels(pfring *ring)
```

Возвращает количество RX каналов (известные как RX очереди) Ethernet интерфейса, который связан с данным кольцом.

Входные параметры:

Ring

Дескриптор PF_RING-а, который будет делать запрос.

Возвращаемое значение:

Количество каналов RX, или 1 (по умолчанию), в случае если информация отсутствует.

```
int pfring_get_selectable_fd(pfring *ring)
```

Возвращает файловый дескриптор, связанный с указанным кольцом. Это число может быть использовано в таких функциях как, `poll()` и `select()` для пассивного ожидания входящих пакетов.

Входные параметры:

Ring

Дескриптор PF_RING-а, который будет делать запрос.

Возвращаемое значение:

Это число может быть использовано как ссылка на кольцо, и в функциях, которые требуют указывать файловый дескриптор.

```
int pfring_enable_rss_rehash(pfring *ring)
```

Сообщает PF_RING сделать rehash входящих пакетов, используя двунаправленную хэш-функцию. Это также возможно с помощью библиотеки librcar с поддержкой PF-RING через переменную окружения PCAP_PF_RING_RSS_REHASH.

Входные параметры:

Ring

Дескриптор PF_RING-а, который будет делать запрос.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

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

Читает время с аппаратных часов сетевого устройства, если адаптер поддерживает аппаратную пометку временем.

Входные параметры:

Ring

Дескриптор PF_RING-а.

Ts

Структура, куда будет записано время.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

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

Задаёт время на аппаратных часах сетевого устройства, если адаптер поддерживает аппаратную пометку временем.

Входные параметры:

Ring

Дескриптор PF_RING-а.

Ts

Время, которое будет задано.

Возвращаемое значение:

0 в случае успеха, в противном случае отрицательное значение.

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

Настройка времени на аппаратных часах на заданное смещение, если адаптер поддерживает пометку временем.

Входные параметры:

Ring

Дескриптор PF_RING-а.

Offset

Время смещения.

Sign

Знак смещения (отстают или спешат).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

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

Включает аппаратные метки для tx и rx, если адаптер их поддерживает.

Входные параметры:

Ring

Дескриптор PF_RING-а.

device_name

Имя устройства, на котором будут включена пометка временем.

Enable_rx

Флаг, который включает пометку временем rx пакетов.

Enable_tx

Флаг, который включает пометку временем tx пакетов.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
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)
```

Разбор пакета.

Разборщик пакета, который ожидает запись hdr или нулями или допустимыми данными текущего пакета, что позволяет избежать вторичного разбора заголовков того же пакета. Это обеспечивается с помощью контроля полей l3_offset и l4_offset, указывающие на разобранные уровни L2 и L3 соответственно, при наличии на них значений отличных от нуля.

Входные параметры:

Pkt

Буфер для пакета.

Hdr

Заполненный заголовок.

Level

Уровень заголовка, на котором разбор заканчивается.

`add_timestamp`

Добавляем пометку временем.

`Add_hash`

Вычислить hash на основе двух IP.

Возвращаемое значение:

В случае успеха - неотрицательное число, указывающее самый верхний уровень в заголовке, в противном случае - отрицательное число.

8.15. Интерфейс PF_RING в C++

Интерфейс C++ (смотрите PF_RING/userland/c++/) аналогичен интерфейсу C. Больших изменений не произошло и все имена функций схожи с функциями в C. К примеру:

- C: `int pfring_stats(pfring *ring, pfring_stat *stats);`
- C++: `inline int get_stats(pfring_stat *stats);`

9. Библиотека `libzero` для DNA

Эта библиотека реализует механизм zero-сору для взаимодействия процессов, так что она может быть использована в многопоточных и многопроцессных приложениях. Как описывалось вначале, она предоставляет 2 главных компонентов: кластер DNA и проталкиватель DNA.

9.1. Кластер DNA

Кластер DNA реализует пакетную кластеризацию, благодаря чему все приложения, относящиеся к одному кластеру, могут видеть все входящие пакеты, реализуя механизм zero-сору используя функцию распределения, определенную пользователем. Так же приложения могут отправлять пакеты через zero-сору. Каждое приложение читает/отправляет пакеты из/на «управляемый» сокет.

Основной(ое) поток/приложение ответственный(ое) за распределение входящих пакетов для подчинённых потоков/приложений использует определённую пользователем функцию распределения (по умолчанию это хэш-функция двух IP). Функция так же может выступать в качестве разветвителя, («тройника») предоставляющая тот же пакет нескольким управляемым (подчинённым) потокам/приложениям, не замедляя потребителя, который влияет на производительность последующих функций. Кластер позволяет приложению обрабатывать пакеты «с пропусками» (т.е. обрабатывать пакеты не по порядку), переходя к следующему входящему пакету, даже если предыдущий пакет еще не обработан.

9.1.1. Основные API (The Master API)

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

Создает новый дескриптор кластера DNA. Созданный кластер ещё не взаимосвязан с кольцом.

Входные параметры:

Cluster_id
Идентификатор кластера.

Num_apps
Количество подчинённых приложений/потоков.

Flags

Маска для включения дополнительных функций:

- DNA_CLUSTER_DIRECT_FORWARDING: включает «направленную пересылку» для передачи входящих пакетов на интерфейс для отправки, в соответствии с возвращаемыми значениями распределяющей функции.
- DNA_CLUSTER_NO_ADDITIONAL_BUFFERS: отключает область памяти «pfring_pkt_buff» (смотри раздел «управляемые API»), тем самым уменьшая используемую память.
- DNA_CLUSTER_HUGEPAGES: включает область памяти через hugepages (механизм выделения памяти в Linux).

Возвращаемое значение:

Дескриптор (заголовок) кластера.

```
int dna_cluster_low_level_settings(pfring_dna_cluster *handle, u_int32_t slave_rx_queue_len,  
u_int32_t slave_tx_queue_len, u_int32_t slave_additional_buffers);
```

Изменяет стандартные значения количества слотов у очередей rx/tx и дополнения к буферу (только для профессионалов).

Входные параметры:

Handle
Дескриптор кластера DNA.

slave_rx_queue_len
количество слотов в подчиненной rx очереди.

slave_tx_queue_len
количество слотов в подчиненной tx очереди.

slave_additional_buffers
Максимальное значение расширенного буфера, которым он ограничен (смотрите раздел «управляемые API»).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int dna_cluster_register_ring(pfring_dna_cluster *handle, pfring *ring)
```

Добавляет сокет PF_RING к кластеру DNA.

Входные параметры:

Handle
Дескриптор кластера DNA.

Ring
Дескриптор PF_RING.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

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

Привязывает главные потоки RX и TX к ядру с id. В кластерах, потоки используются для приёма (RX) и отправки (TX) пакетов кластера. Эта функция используется для привязки указанного потока к ядру.

Входные параметры:

Handle
Дескриптор кластера DNA.

Rx_core_id
Id ядра для потока RX.

Tx_core_id
Id ядра для потока TX.

```
int dna_cluster_set_mode(pfring_dna_cluster *handle, socket_mode mode)
```

Задаёт режим кластеру: принимать (только RX), отправлять (только TX), или и принимать и отправлять (RX и TX).

Входные параметры:

Handle
Дескриптор DNA кластера.

Mode
Режим кластера.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
void dna_cluster_set_distribution_function(pfring_dna_cluster *handle,  
pfring_dna_cluster_distribution_func func)
```

Задаёт функцию распределения пакетов (стандартная функция это hash функция двух ip). Этот вызов позволяет разработчикам задавать свою функцию.

Входные параметры:

Handle
Указатель на кластер DNA.

Func
Функция распределения.

```
void dna_cluster_set_wait_mode(pfiring_dna_cluster *handle, u_int32_t active_wait)
```

Задаёт входящему пакету режим ожидания: пассивное (опрос) или активное ожидание.

Входные параметры:

Handle
Дескриптор DNA кластера.

active_wait
Булево значение: 0 – пассивный режим, 1 – активный режим.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int dna_cluster_enable(pfiring_dna_cluster *handle)
```

Включает кластер.

Входные параметры:

Handle
Дескриптор DNA кластера.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int dna_cluster_disable(pfiring_dna_cluster *handle)
```

Отключает кластер.

Входные параметры:

Handle
Дескриптор DNA кластера.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
int dna_cluster_stats(pfiring_dna_cluster *handle, u_int64_t *tot_rx_packets, u_int64_t *tot_tx_packets, u_int64_t *tot_rx_processed)
```

Возвращает статистику кластера.

Входные параметры:

Handle
Дескриптор DNA кластера.

tot_rx_packets
Общее количество принятых пакетов

tot_tx_packets
Общее количество отправленных пакетов.

tot_rx_processed
Общее количество пакетов обработанных подчиненными (процессами, потоками).

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
void dna_cluster_destroy(pfring_dna_cluster *handle)
```

Обрушение кластера, и закрытие связи с сокетом PF_RING.

Входные параметры:

handle
Дескриптор DNA кластера.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

9.1.2. Подчинённые API

DNA кластер ведёт (управляет) поток/приложение используя расширенный набор PF_RING API, предоставляющий обратную совместимость со всеми существующими приложениями. Далее в этом документе вы увидите различия между двумя наборами.

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

Возвращает дескриптор буфера пакета. Память выделяется на PF_RING внутри ядра и управляется с помощью PF_RING (т.е. нет free() на эту память) используя вызовы pfring_XXX_XXX.

Входные параметры

Ring
Дескриптор PF_RING.

Возвращаемое значение:

Возвращает дескриптор буфера.

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

Освобождает заголовок буфера пакета, (предварительно) выделенного функцией `pfring_alloc_pkt_buff`.

Входные параметры:

Ring
Дескриптор PF_RING

Pkt_handle
Дескриптор буфера.

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

Принимает загруженный пакет из буфера, обозначенный параметром `pkt_handle`, вместо возвращения нового буфера. В двух словах, возвращенный пакет помещают в тот же аргумент (прошедший контроль) функции.

Входные параметры:

Ring
Дескриптор PF_RING

Pkt_handle
Дескриптор буфера, куда будет записан входящий пакет.

Hdr
Заголовок PF_RING

wait_for_incoming_packet
если 0, то мы просто проверяем доступность пакета, в противном случае функция блокируется до того, пока пакет не будет доступен.

Выходное значение:

0 - в случае отсутствия принятого пакета (при не блокирующейся функции), 1 в случае успеха, -1 в случае ошибки.

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

Отправляет пакет, через дескриптор буфера `pkt_handle`. Примечание: Эта функция сбрасывает содержимое указателя буфера, если нужно сохранить содержимое убедитесь, что скопировали данные до её вызова.

Входные параметры:

Ring
Дескриптор PF_RING.

pkt_handle

Дескриптор буфера.

flush_packet

возможная очистка переданной очереди.

Возвращаемое значение:

В случае успеха – возвращает количество отправленных байтов, в противном случае - отрицательное значение.

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

Возвращает указатель буфера, указывающий на дескриптор буфера с пакетом.

Входные параметры:

Ring

Дескриптор PF_RING.

Pkt_handle

Дескриптор буфера.

Возвращаемое значение:

Указатель на буфер с пакетом.

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

Задаёт длину пакета. Функция нужна, если вы хотите указывать собственную длину пакета, вместо того чтобы использовать размер принятого пакета.

Входные параметры:

Ring

Дескриптор PF_RING.

Pkt_handle

Дескриптор буфера.

Len

Длина пакета.

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

Связывает дескриптор буфера (обрабатывающий пакет) с id интерфейса. Этой функцией обычно задаётся индекс исходящего интерфейса.

Входные параметры:

Ring

Дескриптор PF_RING.

Pkt_handle

Дескриптор на буфер.

if_index
id интерфейса.

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

Добавляет id интерфейса к интерфейсам ids дескриптора буфера. Функция используется, чтобы задать выходные интерфейсы (разветвитель) буферу пакетов.

Входные параметры:

Ring
Дескриптор PF_RING.

Pkt_handle
Дескриптор буфера.

If_index
Id интерфейса.

```
void pfring_register_zerocopy_tx_ring(pfring *ring, pfring *tx_ring)
```

Присоединяет сокет DNA к кластеру DNA, что позволяет приложению принимать пакеты из кластера и отправлять их через zero-copy на DNA интерфейс /очередь.

Входные параметры:

Ring
Дескриптор PF_RING (кластер DNA подчиненный/потребитель).

Tx_ring
Сокет DNA (tx), который должен быть подсоединён к кластеру.

9.2. DNA проталкиватель

DNA проталкиватель перекидывает (коммутирует) пакеты между 2-мя интерфейсами в режиме zero-copy, оставляя возможность пользователю устанавливать функцию которая может принимать решение к каждому последующему пакету, любой из которых должен быть оправлен или нет. Проталкиватель может работать в однонаправленном режиме, что означает копирование только в одном направлении (ввод в выходные кольца, если нужно двунаправленное копирование необходимо создать два проталкивателя потока/приложения), или в двунаправленном режиме.

9.2.1. API DNA проталкивателя

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

Создаёт новый дескриптор DNA проталкивателя.

Входные параметры:

ingress_ring

Сокет, откуда пакеты будут читаться.

egress_ring

Сокет через который будут отправляться пакеты.

Возвращаемое значение:

Дескриптор DNA проталкивателя.

```
int pfring_dna_bouncer_set_mode(pfring_dna_bouncer *handle, dna_bouncer_mode mode)
```

Задаёт режим работы DNA проталкивателя: односторонний (стандартный) или двусторонний.

Входные параметры:

Handle

Дескриптор DNA проталкивателя.

Mode

Режим: one_way_mode или two_way_mode.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
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)
```

Включает DNA проталкиватель.

Входные параметры:

Handle

Дескриптор DNA проталкивателя.

Func

Функция обработки пакета.

user_bytes

Указатель на пользовательские данные.

wait_for_packet

если 0 то применяется ожидание, используемое для проверки доступности пакета.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

```
void pfring_dna_bouncer_breakloop(pfring_dna_bouncer *handle)
```

Останавливает проталкиватель.

Входные параметры:

Handle

Дескриптор DNA проталкивателя.

```
void pfring_dna_bouncer_destroy(pfring_dna_bouncer *handle)
```

Уничтожает проталкиватель и закрывает связанные сокет PF_RING.

Входные параметры:

Handle
Дескриптор DNA проталкивателя.

9.3. Фрагменты кода для общих случаев.

9.3.1. DNA кластер: принять пакет и отложить его в сторону

```
/*берем дескриптор буфера с пакетом*/  
pfring_pkt_buff *pkt_handle = pfring_alloc_pkt_buff(ring);  
//проверяем его, на наличие в нём данных  
if (pkt_handle != NULL) {  
    /*помещаем принятый пакет в pkt_handle (делаем перезапись этого аргумента) */  
    rc = pfring_recv_pkt_buff(ring, pkt_handle, &hdr, wait_for_packet);  
    /*если пакет записали успешно, то*/  
    if (rc > 0) {  
        /*откладываем пакет с сторону для дальнейших действий над ним*/  
        enqueue_packet(pkt_handle);  
    }  
}  
/*берем отложенный пакет*/  
pkt_handle = dequeue_packet();  
  
/*делаем что-нибудь с этим пакетом и освобождаемся от него*/  
/*берем дескриптор этого буфера*/  
buffer = pfring_get_pkt_buff_data(ring, pkt_handle);  
/*чистим дескриптор буфера */  
pfring_release_pkt_buff(ring, pkt_handle);
```

9.3.2. DNA кластер: приём пакета и отправка его через механизм zero-copу

```
/*берем дескриптор буфера с пакетом*/  
pfring_pkt_buff *pkt_handle = pfring_alloc_pkt_buff(ring);  
//проверяем его, на наличие в нём данных  
if (pkt_handle != NULL) {  
    /*помещаем принятый пакет в pkt_handle (делаем перезапись этого аргумента) */  
    rc = pfring_recv_pkt_buff(ring, pkt_handle, &hdr, wait_for_packet);  
    /*если пакет записали успешно, то*/  
    if (rc > 0) {  
        /*если, пакет отправляется на другой интерфейс*/  
        if (forward_packet_to_another_interface) {  
            /*задаём соответствие пакета pkt_handle с интерфейсом if_index*/  
            pfring_set_pkt_buff_ifindex(ring[thread_id], pkt_handle, if_index);  
        }  
    }  
}
```

```

    } else {
        /*отражает пакет на rx интерфейс (уже заданный в pkt_handle)*/
    }

    /*отправляет пакет pkt_handle, после отправки буфер чистится */
    pfring_send_pkt_buff(ring[thread_id], pkt_handle, 0);
}
}

```

9.3.3. Кластер DNA: Замена стандартной функции распределения на пользовательскую функцию.

```

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;
    /*рассчитывание хэша двунаправленной программы */
    *hash = custom_hash_function(buffer, buffer_len);
    /*балансировка по hash*/
    slave_idx = (*hash) % num_slaves;
    *id_mask = (1 << slave_idx);
    return DNA_CLUSTER_PASS;
}
/*задаём функцию распределения пакетов вместо стандартной
hash_distribution_function*/
dna_cluster_set_distribution_function(dna_cluster_handle, hash_distribution_function);

```

9.3.4. Кластер DNA: Замена стандартной функции распределения на разветвляющую функцию.

```

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;
    /* возвращение битового отображения подчинённых id */
    *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. Кластер DNA: отправка входящего пакета сразу, без прохождения через подчинённый процесс или программу.

```
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;
/*создаём новый указатель на кластер DNA*/
dna_cluster_handle = dna_cluster_create(cluster_id, // ид кластера
                                       num_threads, //кол-во подчинённых потоков
                                       DNA_CLUSTER_DIRECT_FORWARDING); // включает
                                       «направленную пересылку»
/*задаёт режим кластеру, в данном случае – отправлять и принимать*/
int dna_cluster_set_mode(dna_cluster_handle, send_and_recv_mode);
/*задаём функцию распределения пакетов вместо стандартной
hash_distribution_function*/
dna_cluster_set_distribution_function(dna_cluster_handle,
                                     hash_distribution_function);
```

10. Пишем плагины к PF_RING

Начиная с версии 3.7, разработчики могут писать плагины, чтобы делегировать с PF_RING:

- Анализ (парсинг) полей данных пакета
- Фильтрация по содержимому пакета
- Расчёт статистики трафика в ядре

Чтобы понять идею представьте, что вам нужно разработать приложение для мониторинга VoIP трафика. В этом случае Вам для этого необходимо:

- Анализировать сигнализирующие пакеты (напр. SIP или IAX), чтобы пересылать только те пакеты, которые причастны к интересующим соединениям.
- Вычислить статистику пакетов передающих голос, через PF_RING и оповестить пользователя (приложение) только о статистике, без самих пакетов.

В этом случае разработчик может написать 2 плагина, так что PF_RING можно использовать как расширенный фильтр трафика и как способ увеличения скорости обработки пакета с помощью обхода пакетов, не касаясь ядра, когда в этом нет необходимости.

В остальной части главы объясняется, как сделать плагин и как его вызвать из пространства пользователя.


```
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)
```

Эта функция вызывается всякий раз, когда получают входящий (RX или TX) пакет. Она обычно обновляет статистику фильтра. Заметьте, если разработчик задал этот плагин как плагин фильтрации, тогда пакет:

- уже проанализирован;
- передан правилу фильтрации полезной нагрузки (если задано).

Входные параметры:

Rule

Указатель на неполное правило (если этому плагину было задано неполное правило) или NULL (если этому плагину было задано конкретное правило).

hash_rule

Указатель на конкретное правило (если этому плагину было задано конкретное правило), или NULL (если этому плагину было задано неполное правило). Имейте в виду, если правило принимает значение NULL, hash_rule не будет выполняться, и наоборот.

Hdr

Указатель на заголовок принимаемого пакета rсар. Пожалуйста, обратите внимание, что:

- пакет уже разобран;
- заголовок представляет собой расширенный rсар заголовок, содержащий разобранный заголовок пакета по полям протоколов.

Skb

sk_buff структура данных, используемая в Linux для переноса пакетов внутри ядра.

filter_plugin_id

Id плагина который делает разбор полезной нагрузки пакета (не заголовка, который уже записан внутри hdr). Если у filter_plugin_id такой же id как и id у dummy_plugin тогда пакет уже разобран этим плагином и параметр filter_rule_memory_storage указывает на память с разобранный полезной нагрузкой.

parse_memory

Указатель на структуру данных содержащую информацию разобранный полезной нагрузки пакета, которая была разобрана плагином с id параметра filter_plugin_id. Обратите внимание, что:

- только один плагин может разбирать пакет;
- память для разбора распределяется динамически (т.е. через kmalloc) с помощью plugin_filter_skb и освобождается с помощью ядра PF_RING.

Возвращаемое значение:

0 - в случае успеха, в противном случае - отрицательное значение.

10.3. Плагин PF_RING: Фильтрация входящих пакетов.

```
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)
```

Эта функция вызывается всегда перед разбором пакета (через `plugin_handle_skb`) входящего пакета (RX или TX) который нужно пропустить через фильтр. В этом случае пакет разбирается, возвращается проанализированная информация и возвращается значение указывающее, был ли пакет пропущен фильтром.

Входные параметры:

Rule

Указатель на неполное правило, которое содержит значение фильтра применимое к пакету.

Hdr

Указатель на заголовок принимаемого пакета `pcap`. Пожалуйста, обратите внимание, что:

- пакет уже разобран;
- заголовок представляет собой расширенный `pcap` заголовок, содержащий разобранный заголовок пакета по полям протоколов.

Skb

`sk_buff` структура данных, используемая в Linux для переноса пакетов внутри ядра.

Выходные параметры:

`parse_memory`

Указатель на область памяти заданная функцией, которая будет содержать информацию о проанализированной полезной нагрузке пакета.

Возвращаемое значение:

0 - если пакет не совпал с правилом фильтрации, и положительное число - если совпал.

10.4. Плагин PF_RING: Чтение статистики пакетов.

```
int 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)
```

Эта функция вызывается, когда приложение пользователя хочет узнать статистику по фильтрующему правилу.

Входные параметры:

Rule

Указатель на неполное правило (если этому плагину было задано неполное правило) или NULL (если этому плагину было задано конкретное правило).

hash_rule

Указатель на конкретное правило (если этому плагину было задано конкретное правило), или NULL (если этому плагину было задано неполное правило).

Имейте в виду, если правило NULL, hash_rule не будет, и наоборот.

stats_buffer

указатель на буфер, куда статистика будет скопирована.

stats_buffer_len

Длина в байтах буфера stats_buffer.

Возвращаемое значение:

Длина правила статистики, 0 - в случае ошибки.

10.5. Использование плагина PF_RING

Приложение, основанное на PF_RING, может воспользоваться преимуществами плагинов, когда заданы правила фильтрации. Структура данных filtering_rule используется для задания правила и указания связанного с ней плагина.

```
filtering_rule rule;
```

```
rule.rule_id = X;
```

```
....
```

```
rule.plugin_action.plugin_id = MY_PLUGIN_ID;
```

Когда plugin_action.plugin_id задан, всякий раз, когда пакет соответствует части заголовка правила, тогда плагин MY_PLUGIN_ID (если зарегистрирован) вызывается и вызываются plugin_filter_skb () и plugin_handle_skb() .

Если разработчик хочет фильтровать пакеты до вызова plugin_handle_skb(), тогда расширенные поля filtering_rule должны быть заданы. Например, реализуем плагин фильтрации SIP и сделаем так, чтобы возвращались только пакеты INVITE. Следующие строки кода показывают, как это сделать.

```
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 */
```

Как объяснялось выше, функция `pfiring_add_filtering_rule()` используется для регистрации правил фильтрации.

11. Структуры данных PF_RING

Ниже описаны некоторые релевантные (актуальные) структуры данных PF_RING.

```
typedef struct {
    u_int16_t rule_id;                /*правила обрабатываются от нижнего до высшего id*/
    rule_action_behaviour rule_action; /*что делать в случае совпадения*/
    u_int8_t balance_id, balance_pool; /*если balance_pool > 0, тогда передаём пакет выше, только если
                                        (hash(proto, sip, sport, dip, dport) % balance_pool) = balance_id */
    u_int8_t locked;                 /*не удалять*/
    u_int8_t bidirectional;          /* взаимодействие (направление) узлов (По умолчанию: моно)*/
    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;    /*внутренние поля PF_RING*/
} filtering_rule;

typedef struct {
    u_int8_t smac[ETH_ALEN], dmac[ETH_ALEN]; /*используйте «0» (нулевой MAC-адрес) для любого MAC-адреса. Это
                                                применяется к источнику и назначению */
    u_int16_t vlan_id;                  /*используйте «0» для указания любого vlan*/
    u_int8_t proto;                    /*используйте «0» для указания любого протокола*/
    ip_addr shost, dhost;              /* «0» для любого хоста. Это применяется к источнику и
                                        назначению */
    ip_addr shost_mask, dhost_mask;    /*IPv4/6 маска*/
    u_int16_t sport_low, sport_high;   /*диапазон портов от port_low до port_high */
    u_int16_t dport_low, dport_high;  /* диапазон портов от port_low до port_high */
} filtering_rule_core_fields;

typedef struct {
    char payload_pattern[32];          /*если strlen(payload_pattern) > 0, тогда полезная нагрузка пакета
                                        должна соответствовать шаблону*/
    u_int16_t filter_plugin_id;        /*если > 0 то он определяет плагин, к которому будет передавать
                                        нижеприведённую структуру данных для сравнения
                                        (сопоставления) */
    char filter_plugin_data[FILTER_PLUGIN_DATA_LEN]; /*непрозрачная структура данных, которая определяется с
                                                помощью заданного плагина и которая задаёт критерии
                                                фильтрации для проверки соответствия. Как правило, эти
                                                данные пересобирается на более полную структуру
                                                данных*/
} 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, /*автозаполняющееся конкретное правило или через
                                                plugin_add_rule() */
    forward_packet_del_rule_and_stop_rule_evaluation, /*только plugin_del_rule()*/
    reflect_packet_and_stop_rule_evaluation,
}
```

PF_RING User's Guide v.5.5.3

```
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; /* задаёт, что делать в случае совпадения */  
    filtering_rule_plugin_action plugin_action;  
    char reflector_device_name[REFLECTOR_NAME_LEN];  
    filtering_internals internals; /* внутренние поля PF_RING */  
} hash_filtering_rule;
```

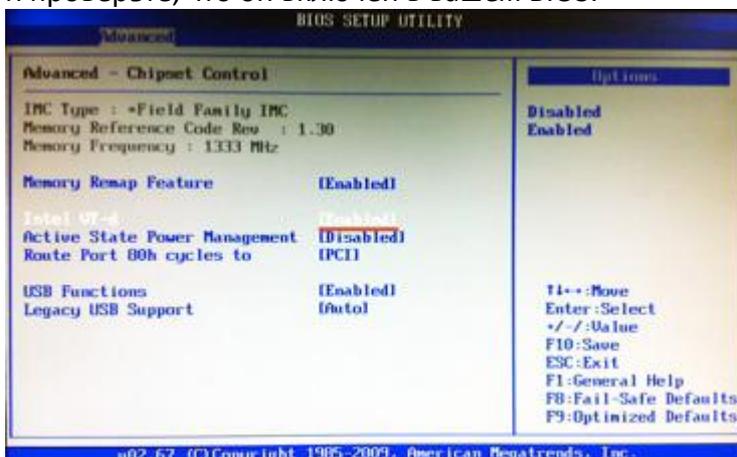
```
typedef struct {  
    u_int64_t recv, drop;  
} pfring_stat;
```

12. PF_RING DNA на виртуальной машине.

Раздел 5.4 содержит краткое изложение модуля PF_RING DNA, который позволяет вам манипулировать пакетами любого размера на 10 Gbit скоростях. Благодаря виртуальным технологиям основанным на IOMMU (Intel VT-d или AMD IOMMU), стало возможным присваивать устройство гостевой операционной системе, которая получит преимущества модуля DNA PF_RING в пределах VM (виртуальной машины). В следующих разделах показано, как сконфигурировать VMWare и KVM. Пользователи XEN могут использовать аналогичные конфигурации системы.

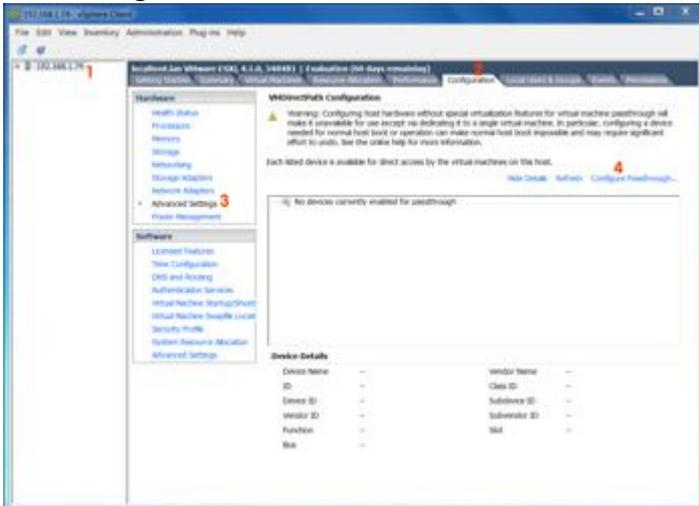
12.1. Конфигурация BIOS

Во-первых, убедитесь, что ваша материнская плата поддерживает проброс PCI устройства и проверьте, что он включен в вашем BIOS.

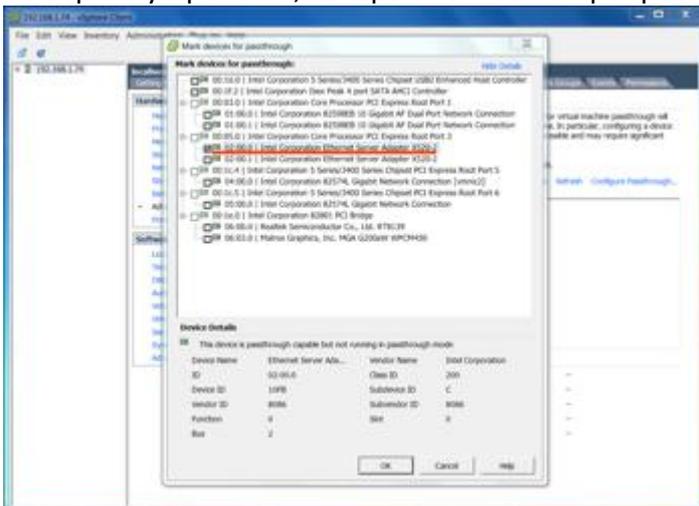


12.2. Конфигурация VMware ESX

Для настройки PCI проброса в VMWare, откройте vSphere Client и соединитесь с сервером. Выберите сервер, проследуйте по “Configuration”, “Advanced Settings”, “Configure Passthrough”.

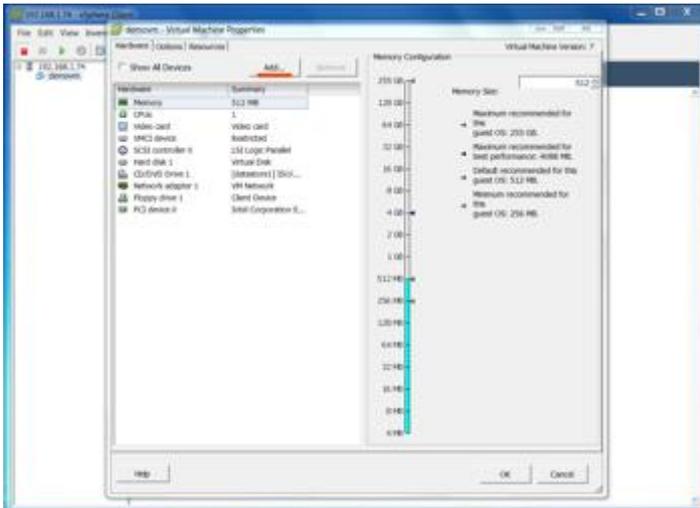


Выберите устройства, которые вы хотите пробросить (видеть) в виртуальную машину.

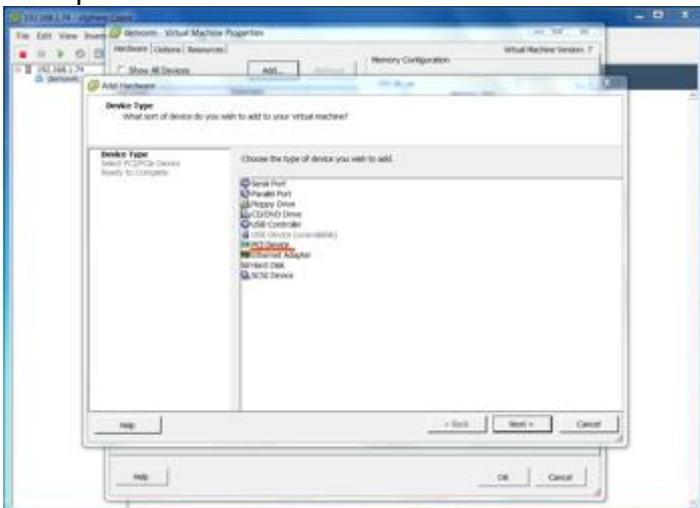


Перезагрузите сервер.

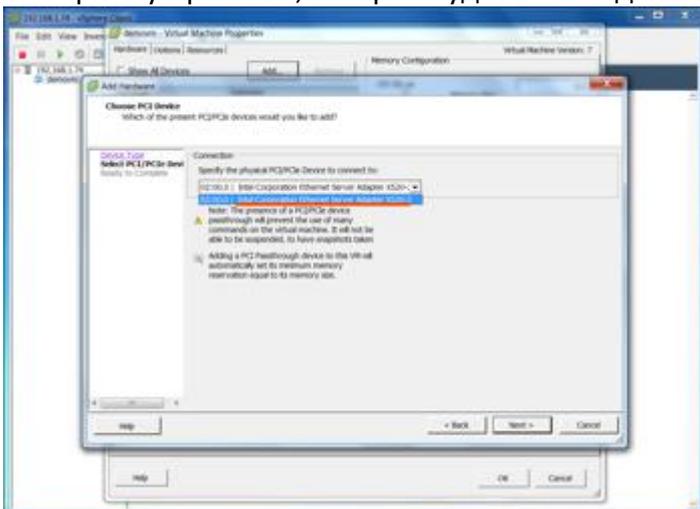
После перезагрузки, убедитесь, что V-ая М-на, где устройство PCI будет задействовано, находится в выключенном состоянии. Откройте настройки VM и нажмите “Add...” во вкладке “Hardware”.



Выберите “PCI Device”.



Выберите устройство, которое будет взаимодействовать с VM.



Загрузите виртуальную машину и установите PF_RING с DNA драйвером, как обычно.

12.3. Конфигурация KVM.

PF_RING User's Guide v.5.5.3

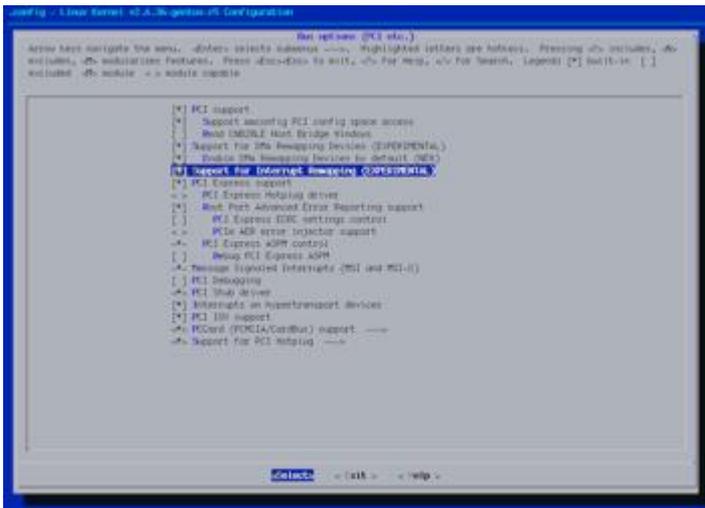
Для того чтобы сконфигурировать PCI проброс на KVM, убедитесь что в вашем ядре включены следующие опции:

Опции шины (PCI и т.д.)

- [*] Поддержка для DMA переназначения устройств.
- [*] Включить DMA переназначение устройств.
- [*] Поддержка для переназначения прерываний.
- <*> Драйвер PCI Stub (заглушка)

```
$ cd /usr/src/linux
```

```
$ make menuconfig
```



```
$ make
$ make modules_install
$ make install
( эти действия зависят от используемого дистрибутива)
```

Передайте “intel_iommu=on” как параметр ядра. Например, если вы используете grub, отредактируйте /boot/grub/menu.lst так:

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

Отвяжите устройство от драйвера ядра основной ОС, которое вы хотите назначить VM.

```
$ 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
```

Загрузите KVM и запустите VM.

```
$ modprobe kvm
$ modprobe kvm-intel
```

PF_RING User's Guide v.5.5.3

```
$ /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
```

Установите и запустите PF_RING с DNA драйвером, как и обычно.