Virtio-IPSec Accelerator

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Reason** |
| 04/22/2015 | 1 | Freescale Semiconductor | Initial version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[Table of Contents 3](#_Toc417469377)

[**1** Introduction 4](#_Toc417469378)

[**2** References 4](#_Toc417469379)

[**3** Scope 4](#_Toc417469380)

[**4** Virtio IPsec Device 5](#_Toc417469381)

[**5** Virtio IPsec Device Definition 6](#_Toc417469382)

[5.1 Bus Details 6](#_Toc417469383)

[5.1.1 Virtio Device ID 7](#_Toc417469384)

[5.1.2 PCI Device Discovery 7](#_Toc417469385)

[5.1.3 Feature Bits 7](#_Toc417469386)

[5.1.4 Virtio IPSec Device Configuration Registers 8](#_Toc417469387)

[5.1.5 Virtqueues 9](#_Toc417469388)

[5.2 Device Emulation 10](#_Toc417469389)

[5.3 Invocation of device in QEMU 11](#_Toc417469390)

[**6** API Messages between Virtio IPsec driver (Guest driver) and Vhost-User 11](#_Toc417469391)

[6.1 Feature Bits 11](#_Toc417469392)

[6.2 Information in queues 11](#_Toc417469393)

[6.3 Command Message Format 11](#_Toc417469394)

[6.4 Control Queue 13](#_Toc417469395)

[6.4.1 GET\_CAPABILITIES 13](#_Toc417469396)

[6.4.2 SET\_GUEST\_ENDIAN 14](#_Toc417469397)

[6.4.3 ADD OUT\_SA 14](#_Toc417469398)

[6.4.4 ADD\_IN\_SA 18](#_Toc417469399)

[6.4.5 UPDATE SA 18](#_Toc417469400)

[6.4.6 DELETE SA 19](#_Toc417469401)

[6.4.7 READ OUT\_SA 19](#_Toc417469402)

[6.4.8 READ\_FIRST\_N\_OUT\_SA 19](#_Toc417469403)

[6.4.9 READ\_NEXT\_N\_OUT\_SA 20](#_Toc417469404)

[6.4.10 READ IN\_SA 20](#_Toc417469405)

[6.4.11 READ\_FIRST\_N\_IN\_SA 20](#_Toc417469406)

[6.5 Notification Queue 21](#_Toc417469407)

[6.5.1 VIRTIO\_IPSEC\_NOTFIY\_LIFETIME\_KB\_EXPIRY 21](#_Toc417469408)

[6.5.2 VIRTIO\_IPSEC\_NOTIFY\_BEFORE\_SEQNUM\_OVERFLOW 22](#_Toc417469409)

[6.5.3 VIRTIO\_IPSEC\_NOTIFY\_SEQNUM\_PERIODIC 22](#_Toc417469410)

[6.6 Data Queue 22](#_Toc417469411)

[**7** Live Migration States 23](#_Toc417469412)

[**8** Libvirt Integration 23](#_Toc417469413)

[**9** IPsec Packet Processing – Look Aside Accelerator Packet Flow 24](#_Toc417469414)

# Introduction

A virtio-IPSec driver and device are introduced in the Virtio Framework as per the Virtio Standards, so that the VNF can use the Virtio IPsec driver to access the IPsec Accelerator functionality. Using the standard Virtio IPSec Driver the VNF can access any underlying vendor specific IPsec Accelerator.

# References

Virtio Specifications

<http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf>

<http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf>

Virtio-net, Vhost-net, Vhost-user implementations in Linux 3.19, Qemu 2.2.0

# Scope

This document identifies a Virtio IPsec Accelerator which will perform IPsec Record Layer Acceleration. (e.g. Freescale SEC engine). In this case, the Guest VM can push IPsec SAs into the hardware accelerator. Subsequently when buffers are submitted, the accelerator can perform Outbound SA processing (clear packet to encrypted packet) or Inbound SA processing (encrypted packet to clear packet) as required. This belongs to Look Aside class of accelerators as, the Guest VM submits packets to the accelerator and receives the processed packet from the accelerator before sending the packet out.

# Virtio IPsec Device

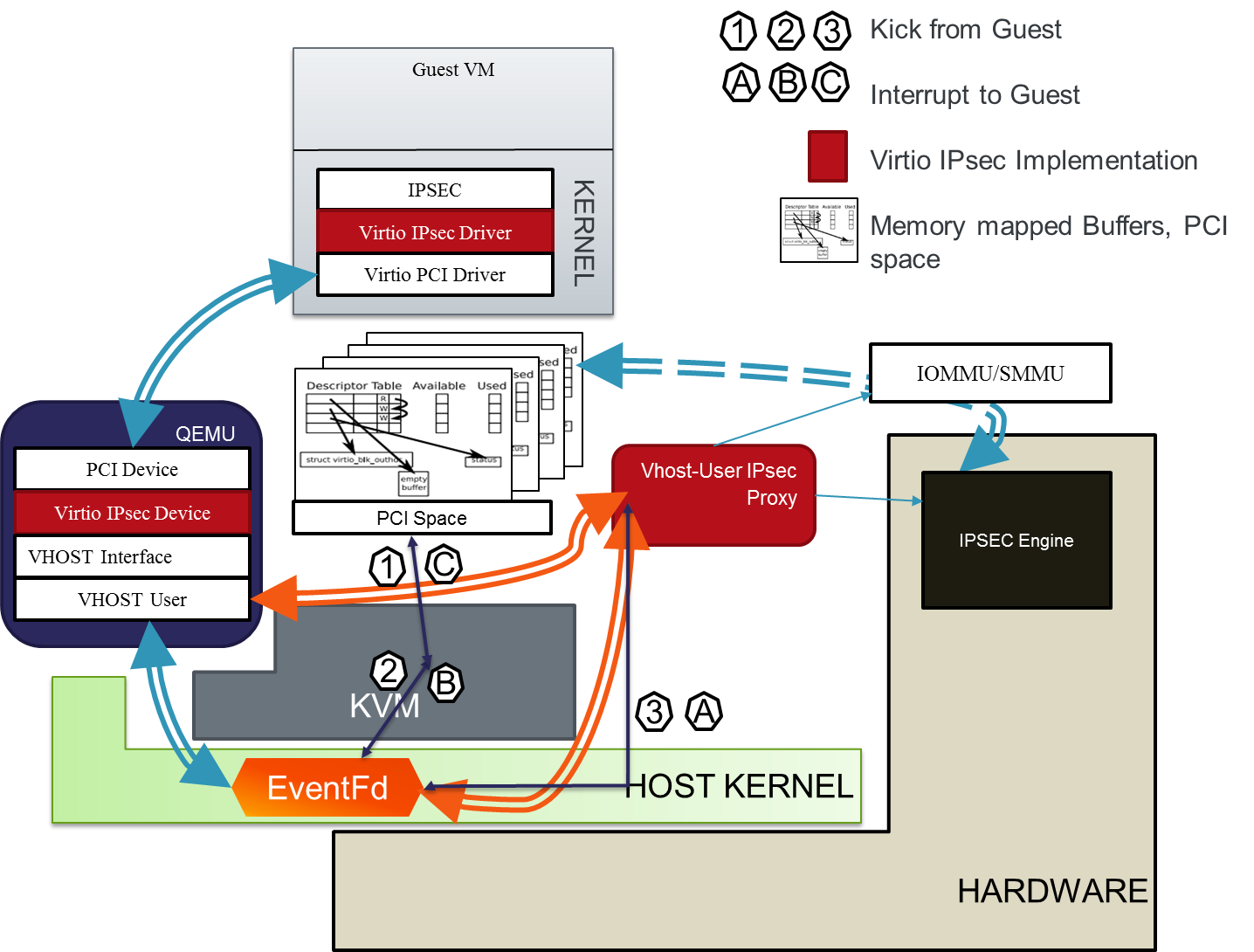


Figure 1: Virtio IPSec Device

Figure 1 shows the Virtio Device implementation in Linux. In the Guest VM, the Virtio IPsec Driver will drives the Virtio IPSec Device. The backend implementation of Virtio IPSec is a Virtio IPSec proxy, that interfaces with the IPsec Engine Hardware Accelerator.

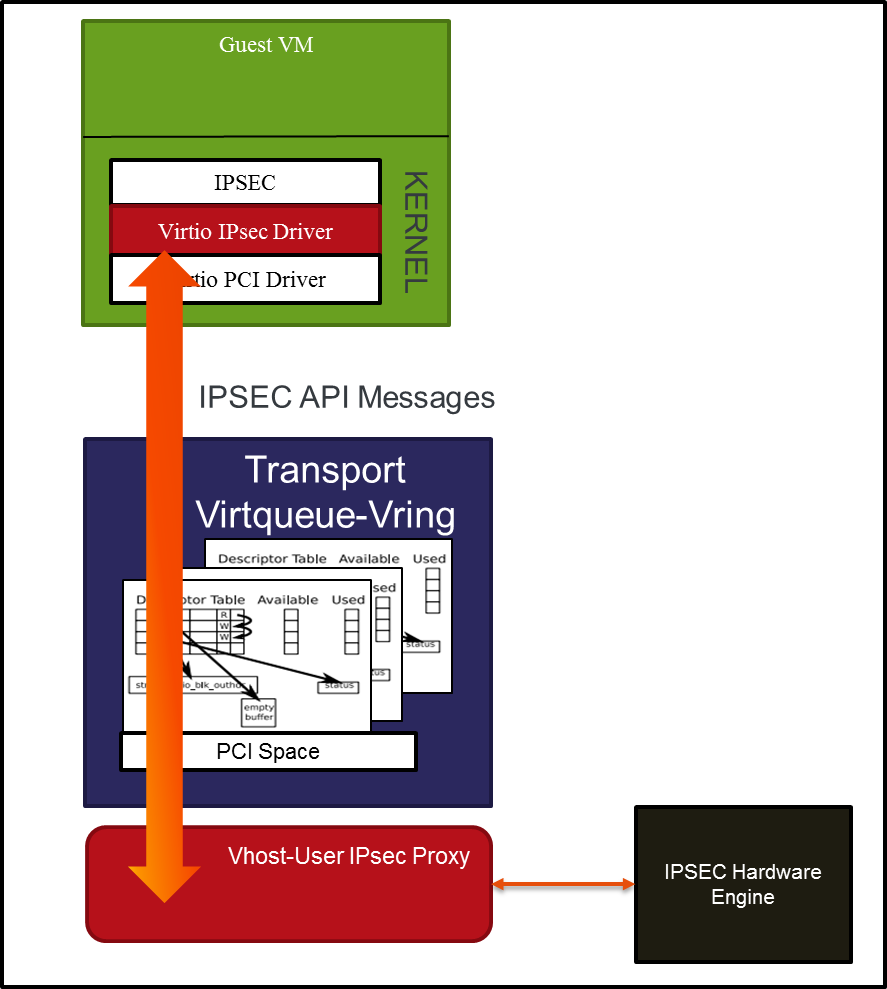


Figure 2 API Messages

As shown in Figure 2, Virtio IPsec driver, sends IPSec API messages through the Virtio/Vring Descriptors to the Vhost-User IPsec Proxy, which interfaces with the IPsec Hardware Accelerator. The API messages for Virtio IPSec is defined in this document.

# Virtio IPsec Device Definition

A default Virtio IPsec Device is expected to provide the following functionality:

1. IPv4 Support
2. Tunnel and Transport Mode
3. ESP (Encapsulating Security Protocol)
4. Checksum to be calculated for Tunnel packets

A Virtio IPSec Device may exhibit other capabilities such as AH processing etc., through the Virtio PCI Feature specifications.

## Bus Details

The bus specific method used to discover this device is Virtio Over PCI bus. The device implements the following parts as per virtio specification

* Device status Field
* Feature bits
* Device Configuration Space
* One or more Virtqueues

The virtio header in the PCI configuration space shall be in PCI and the device header shall follow the endianness as defined by Virtio Standards.

The Virtqueue information (Vring descriptor, available ring, used ring) shall follow the endianness as defined by the Virtio Standards.

The IPsec commands, responses and buffers shall be in Guest Endian Order. The Guest shall inform the endinaness to the Virtio IPsec device using a command.

### Virtio Device ID

The device ID used shall be 20

*Current specification shows allocated device id’s till 18. For experimental drivers, advise in the spec to choose 20 and work backwards*

### PCI Device Discovery

The vendor ID and device ID used shall be as follows -

PCI Vendor ID – 0x1AF4

PCI Device ID – (0x1040 + 20) = 0x1054

### Feature Bits

An advanced Virtio IPsec device may provide features and functions in addition to a default virtio IPSec device. In such a case the Virtio IPsec device shall exhibit its features using the Virtio PCI features bits

VIRTIO\_IPSEC\_F\_SG\_BUFFERS (1) Packet(s) to be IPsec Encrypted/Decrypted can span multiple buffers and hence multiple descriptors; Only the first buffer needs to have the virtio\_ipsec\_hdr; Rest can have the data

VIRTIO\_IPSEC\_F\_AH (2) Device supports AH

VIRTIO\_IPSEC\_F\_WESP (3) /\* Device supports WESP \*/

VIRTIO\_IPSEC\_F\_SA\_BUNDLES (4) /\* Device supports SA bundle \*/

VIRTIO\_IPSEC\_F\_UDP\_ENCAPSULATION (5) /\* UDP Encapsulation for NAT Traversal \*/

VIRTIO\_IPSEC\_F\_TFC (6) Device supports Traffic Flow Confidentiality

VIRTIO\_IPSEC\_F\_ESN (7) Device supports Extended Sequence number

VIRITO\_IPSEC\_F\_ECN (8) Device supports Explicit Congestion Notification

VIRTIO\_IPSEC\_F\_DF (9) DF bit support

VIRTIO\_IPSEC\_F\_ANTI\_REPLAY\_CHECK (10) /\* Device supports Anti replay check \*/

VIRTIO\_IPSEC\_IPV6\_SUPPORT (11) /\* Is Support IPv6 \*/

VIRTIO\_IPSEC\_F\_SOFT\_LIFETIME\_BYTES\_NOTIFY (12) /\* Device notifies when soft life time is about to expire, so that Guest can initiate new SA negotiation \*/

VIRTIO\_IPSEC\_F\_SEQNUM\_OVERFLOW\_NOTIFY 13 /\* Device notifies when sequence number is about to overflow, so that Guest can initiate new SA negotiation \*/

VIRTIO\_IPSEC\_F\_SEQNUM\_PERIODIC\_NOTIFY 14 /\* Periodic update of Sequence number from device to guest \*/

The PCI feature bits part of Virtio Standards will be supported.

VIRTIO\_RING\_F\_INDIRECT\_DESC 28 /\* Support for indirect descriptor table; Refer to Appendix-B, virtio standards \*/

VIRTIO\_RING\_F\_EVENT\_IDX 29 /\* Usage of avail\_event and used\_event fields; Refer to Appendix-B, virtio-standards \*/

### Virtio IPSec Device Configuration Registers

1. Two 32 bit registers will be used for the same
   1. Register 1 – device queue features;  (Read by guest, write by device)
   2. Register 2 – guest queue features; (Write by guest, read by device)
2. Bit mapping for Register 1 and Register 2
   1. Bits 0-  Bit 15
      1. Should indicate maximum number of encap/decap queue pairs.
      2. Maximum possible value = 4k-2 (1 for notification and other for control)
      3. Device will initialize value.
   2. Bits 16- Bit 19
      1. Should indicate maximum number of traffic priority queue pairs.
      2. Device will initialize value
      3. Guest can read and write back into Register 2
   3. Bits 20 – Bit 25 (Max value 64)
      1. Should indicate number of queue pairs to achieve device side scaling
      2. e.g. if 4 hardware threads of AIOP assigned to one Virtual IPsec instance, then this number would read 4]; e.g. if the accel proxy has 4 threads, this number would read 4.
      3. Device will initialize value; Guest modifications in Register 2 will be ignored.
   4. Bits 26 – Bit 31 (Max value 64)
      1. Should indicate maximum number of queue pairs to achieve guest side scaling.
      2. For example if Guest has 4 VCPUs assigned, then the value here would be 4].
      3. Device will not initialize a value in Register 1, Guest writes into register 2
      4. Guest will indicate the value
3. Operation
   1. Qemu side
      1. Initializes the values in Register 1
   2. Guest side (upon probe)
      1. Reads the values in Register 1
      2. Writes back its selection values in Register 2
   3. Guest writes into Register 2 will be ignored or cause a device reset if written into after device is operational.
4. Number of Queues
   1. LCM of Total VM queues (Guest Side Scaling, Input+Output, DSCP priorities), Device side scaling
   2. E.g.: If there are 4 VCPUs assigned to the VM, 3 DSCP priorities, and an input-out pair, the total VM queues = 4 \* 3 \*2 = 24; If there are 2 Proxy threads assigned, the total number of Vrings would be LCM of 24, and 2, which is 24.
5. Implicit agreement between Guest driver/device on the ordering of Queues
   1. Queue 0 – Control Queue
   2. Queue 1 – Optional Notification Queue
   3. The remaining shall be data queues [DQs]
      1. For the following parameters, Guest Side Scaling (GS)=x, DSCP Priority (P) = y and Device Side Scaling (DS) = z,
      2. Total queues tq = LCM ((x\*y\*2), z)
      3. Device side handling
         1. Each proxy thread (without AIOP) or each core (AIOP) would handle tq/z queues
      4. Ordering
         1. For 4 CPUs, and 3 priorities, assuming the ordering is as follows:
            1. VCPU1 will own queues 1-3 for decapsulation
            2. VCPU1 will own queues 4-6 for encapsulation
            3. VCPU2 will own queues 7-9 for decapsulation
            4. VCPU2 will own queues 10-12 for encapsulation
            5. VCPU2 will own queues 13-15 for decapsulation
            6. VCPU2 will own queues 16-18 for encapsulation
            7. VCPU4 will own queues 19-21 for decapsulation
            8. VCPU4 will own queues 22-24 for decapsulation
      5. From device perspective
         1. If the device side scaling is 2, then for the above example, Device core 1 will handle ½ the queues (12) and Device core 2 will handle the remaining half.

### Virtqueues

The following queues shall be supported

1. Control Queue
   1. Each Virtio IPsec device shall support one control queue, through which the Guest/driver shall send commands to setup – add, modify, update SAs and other functionality required for a IPsec Look Aside Accelerator
2. Notification Queue
   1. A virtio IPSec device shall support one notification queue if any of the feature bits VIRTIO\_IPSEC\_F\_SEQNUM\_OVERFLOW\_NOTIFY, VIRTIO\_IPSEC\_F\_SEQNUM\_PERIODIC\_NOTIFY or VIRTIO\_IPSEC\_F\_SOFT\_LIFETIME\_BYTES\_NOTIFY is negotiated between the device and driver.
   2. If any of the above feature bits are negotiated, the Virtio IPSec device shall use the notification queue to notify the guest driver for any such notifications, when enabled using the Control Queue commands.

* Multiple data Encapsulation, Decapsulation Queues will be supported.
  + For a given VM, the number of data Virtqueues calculation would depend on
    - Number of VCPUs assigned to the VM
    - Number of AIOP/iNIC/Proxy threads available to perform the SEC operation (across VNFs or VMs)
    - DSCP/TOS – traffic classes that need to be supported.
    - For example, if there are 4 VCPUs assigned to the Guest, 2 threads in Proxy (device backend) to perform SEC function and 3 Queues to support DSCP/TOS classes, the total number of virtqueues would be LCM of ((4\*3\*2), 2) = 24
    - Arrangement of queues is explained as part of the Device Features [Section 5.1.4].
* Default number of queues will be 4 - 1 Command Queue, 2 Data queues (one for IPsec Encapsulation and encryption, the other for IPSec decapsulation and Decryption) and 1 for optional special notifications from device

## Device Emulation

The virtio-IPsec device shall use a Vhost-User as a backend. The initialization sequence of the virtio IPsec device would be similar to a vhost-user/virtio-net. To summarize the steps here:

1. The device initialization and setup occurs as described in “Device Initialization” in virtio Spec.
2. Vrings are determined at initialization. The virtio-ipsec driver in Guest VM allocates the vrings and writes the Guest Physical Address corresponding to this page via iowrite to VIRTIO\_PCI\_QUEUE\_PFN.
3. Once “DRIVER\_OK” status bit is set in the status register by the driver, the device is live. At this point, vhost-user backend would be hooked with this device.

## Invocation of device in QEMU

Specifying the following in the Qemu command line will add the device:

-device virtio-ipsec-pci,queues=<n>

queues is an optional parameter. By default, 4 queues shall be created.

The full invocation may look like :

# Qemu-kvm ….\

-chardev socket,path=/path/to/socket,id=chr0 \

-object <ipsec-dev>, type=vhost-user, chardev-chr0,id=ipsec0 \

-device virtio-ipsec-pci,ipsec-dev=ipsec0

-object memory-backend-file share=on

Multiple instances of the device virtio-ipsec-pci shall be supported in the QEMU invocation.

# API Messages between Virtio IPsec driver (Guest driver) and Vhost-User

## Feature Bits

Covered in Section 7.2.3

## Information in queues

The information sent as structures in the queue (Control, Notification or Data) shall be packed, using C attribute \_\_packed\_\_.

## Command Message Format

/\*

\* IPSec Control virtqueue data structures

\*

\* The control virtqueue expects a header in the first sg entry

\* and an result/status response in the last entry. Data for the

\* command goes in between.

\* Note: The ctrl\_hdr, ctrl\_result and the actual command can be sent as a single buffer as well

\*/

struct virtio\_ipsec\_ctrl\_hdr {

u8 class; /\* class of command \*/

u8 cmd; /\* actual command \*/

};

Struct virtio\_ipsec\_ctrl\_result {

u8 result; /\* VIRTIO\_IPSEC\_OK or VIRTIO\_IPSEC\_ERR \*/

u8 result\_data; /\* error information if any \*/

};

/\* Defines for the result field \*/

enum virtio\_ipsec\_result\_value

{

VIRTIO\_IPSEC\_OK = 0, /\* Result is Ok \*/

VIRTIO\_IPSEC\_ERR /\* Result is an error \*/

};

Following messages will be sent on the Command Queue:

enum virtio\_ipsec\_ctrl\_command\_class

{

VIRTIO\_IPSEC\_CTRL\_GENERIC = 1, /\* Generic Commands such as Get/Set Capabilities, Set Endianness etc. \*/

VIRTIO\_IPSEC\_CTRL\_SA, /\* Class of commands to add/modify/delete SA \*/

VIRTIO\_IPSEC\_CTRL\_GET\_RAND\_DATA, /\* Class of commands to get random data \*/

VIRTIO\_IPSEC\_CTRL\_ADVANCED /\* Any vendor specific or advanced commands \*/

};

/\* Generic commands \*/

enum virtio\_ipsec\_ctrl\_command\_class\_generic

{

VIRTIO\_IPSEC\_CTRL\_GET\_CAPABILITIES=1, /\* Underlying algorithm support \*/

VIRTIO\_IPSEC\_CTRL\_SET\_CAPABILITIES=2, /\* Nothing defined here as of now \*/

VIRTIO\_IPSEC\_CTRL\_SET\_GUEST\_ENDIAN=3 /\* Set the Guest endian mode to device \*/

};

/\* SA Commands \*/

enum virtio\_ipsec\_ctrl\_command\_class\_sa

{

VIRTIO\_IPSEC\_CTRL\_ADD\_OUT\_SA=1, /\* Add an outbound SA – Encapsulation \*/

VIRTIO\_IPSEC\_CTRL\_DEL\_OUT\_SA, /\* Delete Outbound SA \*/

VIRTIO\_IPSEC\_CTRL\_UPDATE\_OUT\_SA, /\* Update Outbound SA \*/

VIRTIO\_IPSEC\_CTRL\_READ\_OUT\_SA, /\* Read Outbound SA \*/

VIRTIO\_IPSEC\_CTRL\_READ\_FIRST\_N\_OUT\_SAs /\* Read first N outbound SAs \*/

VIRTIO\_IPSEC\_CTRL\_READ\_NEXT\_N\_OUT\_SAs /\* Read next N Out SAs \*/

VIRTIO\_IPSEC\_CTRL\_ADD\_IN\_SA, /\* Add an inbound SA – Decapsulation \*/

VIRTIO\_IPSEC\_CTRL\_DEL\_IN\_SA, /\* Delete Inbound SA \*/

VIRTIO\_IPSEC\_CTRL\_UPDATE\_IN\_SA, /\* Update Inbound SA \*/

VIRTIO\_IPSEC\_CTRL\_READ\_IN\_SA, /\* Read Inbound SA \*/

VIRTIO\_IPSEC\_CTRL\_READ\_IN\_SA, /\* Read In SA \*/

VIRTIO\_IPSEC\_CTRL\_READ\_FIRST\_N\_IN\_SAs, /\* Read first N SAs \*/

VIRTIO\_IPSEC\_CTRL\_READ\_NEXT\_N\_IN\_SAs /\* Read Next N SAs \*/

};

/\* Random Number \*/

enum virtio\_ipsec\_ctrl\_command\_class\_rand

{

VIRTIO\_IPSEC\_GET\_RAND\_NUM=1

};

## Control Queue

### GET\_CAPABILITIES

The below would be used as bit positions in the capabilities

enum virtio\_ipsec\_hmac\_algorithms {

VIRTIO\_IPSEC\_HMAC\_NULL=0,

VIRTIO\_IPSEC\_HMAC\_MD5,

VIRTIO\_IPSEC\_HMAC\_SHA1,

VIRTIO\_IPSEC\_HMAC\_AES\_XCBC\_MAC,

VIRTIO\_IPSEC\_HMAC\_SHA256,

VIRTIO\_IPSEC\_HMAC\_SHA384,

VIRTIO\_IPSEC\_HMAC\_SHA512,

VIRTIO\_IPSEC\_HMAC\_SHA1\_160

};

enum virtio\_ipsec\_cipher\_alogithms {

VIRTIO\_IPSEC\_CIPHER\_NONE=0,

VIRTIO\_IPSEC\_CIPHER\_DES,

VIRTIO\_IPSEC\_CIPHER\_3DES,

VIRTIO\_IPSEC\_CIPHER\_ESP\_NULL,

VIRTIO\_IPSEC\_CIPHER\_AES,

VIRTIO\_IPSEC\_CIPHER\_AESCTR,

VIRTIO\_IPSEC\_CIPHER\_AES\_CCM\_ICV8,

VIRTIO\_IPSEC\_CIPHER \_AES\_CCM\_ICV12,

VIRTIO\_IPSEC\_CIPHER\_AES\_CCM\_ICV16 ,

VIRTIO\_IPSEC\_CIPHER\_AES\_GCM\_ICV8,

VIRTIO\_IPSEC\_AES\_GCM\_ICV12 ,

VIRTIO\_IPSEC\_AES\_GCM\_ICV16,

VIRTIO\_IPSEC\_NULL\_AES\_GMAC

};

struct virtio\_ipsec\_ctrl\_capabilities {

/\* Algorithm capabilities \*/

u32 hmac\_algorithms;

u32 cipher\_algorithms;

};

### SET\_GUEST\_ENDIAN

enum virtio\_ipsec\_endian

{

VIRTIO\_IPSEC\_GUEST\_LITTLE\_ENDIAN=1,

VIRTIO\_IPSEC\_GUEST\_BIG\_ENDIAN 2

};

struct virtio\_ipsec\_set\_guest\_endian {

u8 endian; /\* GUEST\_LITTLE\_ENDIAN or GUEST\_BIT\_ENDIAN \*/

};

### ADD OUT\_SA

/\* Authentication Algorithms \*/

/\* Refer to enum structure in Get Capabilities Command class \*/

/\* Encryption Algorithms \*/

/\* Refer to enum structure in Get Capabilities Command Class \*/

#define VIRTIO\_IPSEC\_NOUNCE\_LEN 16

#define VIRTIO\_IPSEC\_MAX\_CIPHER\_KEY\_SIZE 64

enum virtio\_ipsec\_qos\_dscp\_setting

{

VIRTIO\_IPSEC\_QOS\_DSCP\_COPY=0,

VIRTIO\_IPSEC\_QOS\_DSCP\_CLEAR,

VIRTIO\_IPSEC\_QOS\_DSCP\_SET

};

enum virtio\_ipsec\_df\_setting

{

VIRTIO\_IPSEC\_DF\_COPY=0

VIRTIO\_IPSEC\_DF\_CLEAR,

VIRTIO\_IPSEC\_DF\_SET

};

enum virtio\_ipsec\_transforms

{

VIRTIO\_IPSEC\_ESP=0,

VIRTIO\_IPSEC\_AH,

VIRTIO\_IPSEC\_ESP\_WITH\_AH

};

#define VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE 8 /\* in bytes \*/

#define VIRTIO\_IPSEC\_TUNNEL\_HDR\_IS\_IPV4 0

#define VIRTIO\_IPSEC\_TUNNEL\_HDR\_IS\_IPV6 1

#define VIRTIO\_IPSEC\_SA\_SAFLAGS\_TUNNEL\_MODE 0

#define VIRTIO\_IPSEC\_SA\_SAFLAGS\_TRANSPORT\_MODE 1

#define VIRTIO\_IPSEC\_SA\_PARAMS\_PROTO\_ESP 50

#define VIRTIO\_IPSEC\_SA\_PARAMS\_PROTO\_AH 51

#define VIRTIO\_IPSEC\_PROPOGATE\_ECN\_ON 0

#define VIRTIO\_IPSEC\_PROPOGATE\_ECN\_OFF 1

#define VIRTIO\_IPSEC\_REPLAY\_CHECK\_ON 0

#define VIRTIO\_IPSEC\_REPLAY\_CHECK\_OFF 1

#define VIRTIO\_IPSEC\_UDP\_ENCAPSULATION\_ON 0

#define VIRTIO\_IPSEC\_UDP\_ENCAPSULATION\_OFF 1

#define VIRTIO\_IPSEC\_UDP\_NAT\_TRAVERSAL\_V1 0

#define VIRTIO\_IPSEC\_UDP\_NAT\_TRAVERSAL\_V2 1

struct virtio\_ipsec\_sa\_params {

u32 ulSPI; /\* Security Parameter Index \*/

u16 /\* Flags \*/

bEncapsulationMode:1,

bIPv4OrIPv6;

bUseExtendedSeqNum:1,

bDoAntiReplayCheck:1,

bDoUDPEncapsulation:1,

bTransforms:2, /\* 00=ESP, 01=AH, 10 = ESP+AH \*/

bNotifySoftLifeKBExpiry:1, /\* Notify when soft life time expires \*/

bNotifyBeforeSeqNumOverlfow:1, /\* Notify ‘n’ packets before Seq number expires \*/

bNotifySeqNumPeriodic:1; /\* Notify Periodically every ‘n’ packets \*/

u32 antiReplayWin;

}

/\* The following structures may be used following the virtio\_ipsec\_sa\_params, based

on the bit field settings \*/

struct virtio\_ipsec\_tunnel\_hdr\_ipv4

{

u32 saddr; /\* Source Address \*/

u32 daddr; /\* Destination Address \*/

u8 bCopyDscp:1,

bHandleDf:2,

bPropogateECN:1;

u8 Dscp; /\* Value to be used for creating DSCP field in Outer IP header \*/

};

struct virtio\_ipsec\_tunnel\_hdr\_ipv6

{

u32 s\_addr[4]; /\* Source Address \*/

u32 d\_addr[4]; /\* Destination Address \*/

u8 b\_copy\_dscp:1,

b\_handle\_df:2,

b\_propogate\_ECN:1;

u8 Dscp; /\* Value to be used for creating DSCP field in Outer IP header \*/

};

/\* Structure to hold NAT Traversal information \*/

struct virtio\_ipsec\_udp\_encapsulation\_info

{

u8 ulNatTraversalMode; /\* v1 or v2 \*/

u16 d\_port; /\* Destination Port Value \*/

u16 s\_port; /\* Source Port Value \*/

};

/\* Structure to hold variable length data; Can be used for sending keys etc. \*/

struct virtio\_ipsec\_lv

{

u32 len; /\* Length of following data \*/

u8 data[0]; /\* actual data \*/

}

struct virtio\_ipsec\_esp\_info

{

u8 ulflags reserved:4,

bAuth:1,

bEncrypt:1;

u8 cipher\_algo; /\* Encryption algorithm as defined in Get Features \*/

u8 IV\_Size;

u8 block\_size;

struct virtio\_ipsec\_lv cipher\_key;

u32 counter\_initial; /\* Initial counter for counter mode algorithms \*/

u8 auth\_algo; /\* Authentication Algorithm as defined in Get Features \*/

struct virtio\_ipsec\_lv auth\_key;

u8 AHPaddingLen;

u8 ICVSize;

struct virtio\_ipsec\_lv nounce\_IV;

}

struct virtio\_ipsec\_ah\_info

{

u8 authAlgo;

struct virtio\_ipsec\_lv auth\_key;

u8 AHPaddingLen;

u8 ICVSize;

}

struct virtio\_ipsec\_create\_sa{

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* Output \*/

struct virtio\_ipsec\_sa\_params sa\_params; /\* Input \*/

/\* Followed by structures based on the flag bits in sa\_params \*/

};

The structures that follow struct virtio\_ipsec\_sa\_params depend on the flags member of the structure.

1. If bEncaspulationMode is set to
   1. VIRTIO\_IPSEC\_SA\_SAFLAGS\_TUNNEL\_MODE, the next structure would be either struct virtio\_ipsec\_tunnel\_hdr\_ipv4 or struct virtio\_ipsec\_tunnel\_hdr\_ipv6 based on the bIPv4OrIPv6 flag setting.
2. If bTransforms is set to
   1. VIRTIO\_IPSEC\_ESP, the next structure would be struct virtio\_ipsec\_esp\_info
   2. VIRTIO\_IPSEC\_AH, the next structure would be struct virtio\_ipsec\_ah\_info
   3. VIRTIO\_IPSEC\_ESP\_WITH\_AH, the next structure would be struct virtio\_ipsec\_esp\_info followed by struct virtio\_ipsec\_ah\_info
3. If bDoUDPEncapsulation is set to VIRTIO\_IPSEC\_UDP\_ENCAPSULATION\_ON
   1. The next structure to follow would be struct virtio\_ipsec\_udp\_encapsulation\_info
4. If bNotifySoftLifeKBExpiry is set
   1. The next structure to follow would be struct virtio\_ipsec\_notify\_lifetime\_kb
5. If bNotifyBeforeSeqNumOverflow is set
   1. The next structure to follow would be virtio\_ipsec\_notify\_before\_seqnum\_overflow
6. If bNotifySeqNumPeriodic is set
   1. The next structure to follow would be virtio\_ipsec\_notify\_before\_seqnum\_overflow

### ADD\_IN\_SA

struct virtio\_ipsec\_create\_in\_sa{

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on flag member of sa\_params; \*/

/\* Refer to Section 8.2.1.3, the set of structures that follow depend on the flag settings in that order \*/

};

### UPDATE SA

/\* Capability – update \*/

struct virtio\_ipsec\_update\_sa\_ipaddr\_v4 {

u16 port;

u32 addr;

}

struct virtio\_ipsec\_update\_sa\_ipaddr\_v6{

u16 port;

u32 addr;

}

#define VIRTIO\_IPSEC\_UPDATE\_SA\_LOCAL\_GW 0

#define VIRTIO\_IPSEC\_UPDATE\_SA\_PEER\_GW 1

struct virtio\_ipsec\_update\_sa {

u8 changeType; /\* LOCAL\_GW, REMOTE\_GW \*/

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

/\* The structure that follows would be either virtio\_ipsec\_update\_sa\_ipaddr\_v4 or virtio\_ipsec\_update\_sa\_ipaddr\_v6 based on whether the updated SA is an ipv4 or ipv6 SA.

};

### DELETE SA

struct virtio\_ipsec\_delete\_sa{

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

};

### READ OUT\_SA

struct virtio\_ipsec\_out\_sa\_info {

u32 low\_seq\_number;

u32 hi\_seq\_number;

};

struct virtio\_ipsec\_read\_out\_sa\_exact {

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_out\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/

};

### READ\_FIRST\_N\_OUT\_SA

struct virtio\_ipsec\_read\_out\_n\_first\_sa {

u32 num\_sas;

u32 opaque\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* Output by Accelerator; Input for next n calls \*/

Array of [

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_out\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/

]

};

### READ\_NEXT\_N\_OUT\_SA

struct virtio\_ipsec\_read\_out\_n\_next\_sa {

u32 num\_sas;

u32 opaque\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* Output by Accelerator; Input for next n calls \*/

Array of [

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_out\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/]

};

### READ IN\_SA

struct virtio\_ipsec\_in\_sa\_info {

u32 low\_seq\_number;

u32 hi\_seq\_number;

};

struct virtio\_ipsec\_read\_in\_sa\_exact {

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_in\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/

};

### READ\_FIRST\_N\_IN\_SA

struct virtio\_ipsec\_read\_in\_n\_first\_sa {

u32 num\_sas;

u32 opaque\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* Output by Accelerator; Input for next n calls \*/

Array of [

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_in\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/

]

};

#### READ\_NEXT\_N\_IN\_SA

struct virtio\_ipsec\_read\_in\_n\_next\_sa {

u32 num\_sas;

u32 opaque\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* Output by Accelerator; Input for next n calls \*/

Array of [

u32 sa\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

struct virtio\_ipsec\_in\_sa\_info info;

struct virtio\_ipsec\_sa\_params sa\_params;

/\* Followed by structures based on the flag bits in sa\_params \*/

]

};

## Notification Queue

Three optional notification messages can be sent by the device to the Guest driver.

enum virtio\_ipsec\_notify\_event

{

VIRTIO\_IPSEC\_NOTIFY\_LIFETIME\_KB\_EXPIRY=1,

VIRTIO\_IPSEC\_NOTIFY\_BEFORE\_SEQNUM\_OVERFLOW,

VIRTIO\_IPSEC\_NOTIFY\_ SEQNUM\_PERIODIC

}

### VIRTIO\_IPSEC\_NOTFIY\_LIFETIME\_KB\_EXPIRY

When this notification message is sent by device to Guest, the following structure will be queued in the Notification Queue

struct virtio\_ipsec\_notify\_lifetime\_kb\_expiry

{

struct enum virtio\_ipsec\_notify\_event; /\* Value = VIRTIO\_IPSEC\_NOTIFY\_LIFETIME\_KB\_EXPIRY \*/

u32 sa\_context\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

u32 lifetime\_in\_kb; /\* Current Lifetime in Kb \*/

};

### VIRTIO\_IPSEC\_NOTIFY\_BEFORE\_SEQNUM\_OVERFLOW

When this notification message is sent by device to guest, the following structure will be queued in the Notification Queue.

struct virtio\_ipsec\_notify\_before\_seqnum\_overflow

{

struct enum virtio\_ipsec\_notify\_event; /\* Value = VIRTIO\_IPSEC\_NOTIFY\_BEFORE\_SEQNUM\_OVERFLOW \*/

u32 sa\_context\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

u32 seqnum; /\* Current Sequence Number \*/

};

### VIRTIO\_IPSEC\_NOTIFY\_SEQNUM\_PERIODIC

Struct virtio\_ipsec\_notify\_periodic\_seqnum

{

struct enum virtio\_ipsec\_notify\_event; /\* Value = VIRTIO\_IPSEC\_NOTIFY\_SEQNUM\_PERIODIC \*/

u32 sa\_context\_handle[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE];

u32 seqnum; /\* Current sequence number \*/

};

## Data Queue

Packets are IPSec-encrypted by placing them in the EncryptQ1…EncryptQn and decrypted by placing them in the DecryptQ1..DecryptQn. In each case, the data is preceded by a header. When data is spread across a SG list, only the first buffer shall have the virtio\_ipsec\_hdr. The rest of the buffers will have only data.

struct virtio\_ipsec\_hdr {

u32 sa\_context\_handle[[VIRTIO\_IPSEC\_SA\_HANDLE\_SIZE]; /\* IPsec SA Context \*/

u32 num\_input\_buffers; /\* Number of input buffers \*/

u32 input\_data\_length; /\* Length of input data \*/

u32 num\_output\_buffers; /\* Number of output buffers \*/

u32 output\_buffer\_length; /\* Size of output buffers \*/

u32 flags; /\* for future use \*/

u32 output\_data\_length; /\* Output data length \*/

u32 result;

u32 error\_code;

};

Following the header there will be the descriptors pointing to the input buffer, followed by descriptors pointing to the output buffer.

Desc-1

Desc-2

Desc-3

Desc-4

Desc-5

struct virtio\_ipsec\_hdr

Input Buffer1

Input Buffer 2

Output Buffer 1

Output Buffer 2

Figure 3 Example descriptor chain

Figure 6 shows an example descriptor chain. The first descriptor points to the struct virtio\_ipsec\_hdr, that contains the buffer information and results. The next 2 descriptors point to input buffers. The last 2 descriptors point to the Output buffers.

Note: As the descriptor queue size supported is 64k (16 bit space), there is no need to use the Indirect Descriptors. SG descriptors shall be made use of.

# Live Migration States

TBD

# Libvirt Integration

<devices>

<ipsec model='virtio'>

<queues=6/>

<backend model='vhostuser'>

<source type='unix' path='/tmp/vhost.sock' mode='server'/>

<model type='virtio'/>

</backend>

</ipsec>

</devices>

Name of the device “ipsec”.

Optional parameter – queues – If this option is not specified, by default 3 queues will be created.

Backend – of type vhostuser

The <source> element has to be specified along with the type of char device. Currently, only type='unix' is supported, where the path (the directory path of the socket) and mode attributes are required. Both mode='server' and mode='client' are supported. vhost-user requires the virtio model type, thus the <model> element is mandatory.

# IPsec Packet Processing – Look Aside Accelerator Packet Flow

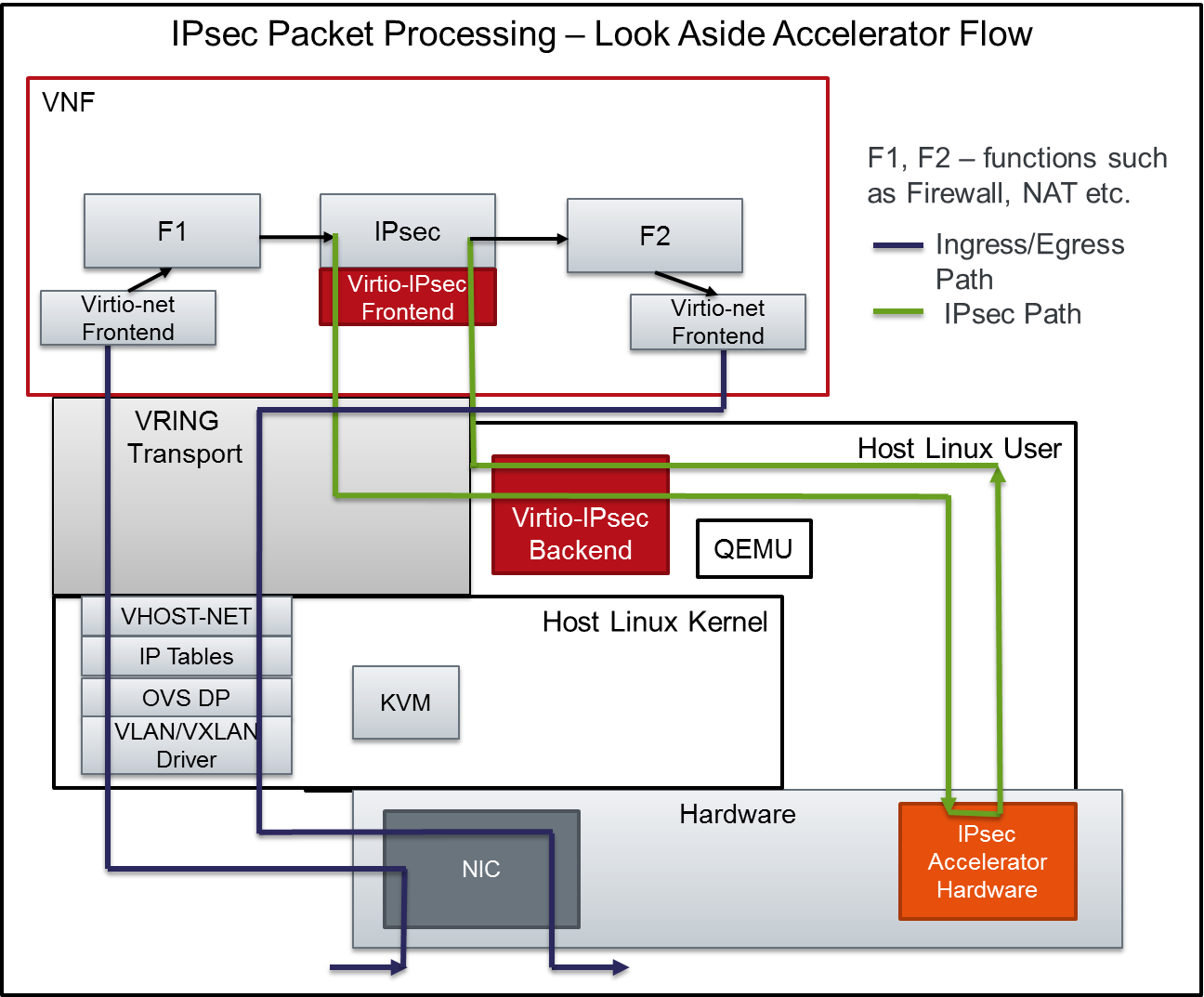


Figure 5 IPsec Packet Processing –Look Aside Accelerator Flow

Figure 4 shows the flow of packets when IPsec Look aside accelerator is used. F1, F2 stand for several packet processing functions such as Firewall, NAT etc.

Ingress Packet Flow:

* Packets processed by VXLAN/VLAN, OVS Data Path, IP Tables, Vhost-Net
* Packet announced to VNF through Virtio-Net driver
* Packets under several function processing such as Firewall etc.
* Packets arrive at the IPsec module for IPsec Packet Processing
* As packets are submitted by the IPsec Module to the Virtio-IPsec front end driver, the buffers are put in the Virtio Descriptor Vrings or Virt Qs to be transferred to the Virtio-IPsec Backend.
* The Virtio IPSec Backend is responsible for translating the packets from Virt Q Descriptor to the actual hardware accelerator in a message that the accelerator understands and vice-versa.
* The Virtio IPsec Backend is also responsible for picking up processed packets from the hardware accelerator, updating the VirtQ rings and notifying the Guest VNF.
* The processed packets under further processing functions (F2 etc.) before being sent out through the Virtio interface.