Virtual Switch on BlueField

Table of contents

Note

For general information on OVS offload using ASAP² direct, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP² Direct.

Note

ASAP² is only supported in Embedded (DPU) mode.

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) support ASAP² technology. It utilizes the representors mentioned in the previous section. The BlueField software package includes OVS installation which already supports ASAP². The virtual switch running on the Arm cores allows us to pass all the traffic to and from the host functions through the Arm cores while performing all the operations supported by OVS. ASAP² allows us to offload the datapath by programming the NIC embedded switch and avoiding the need to pass every packet through the Arm cores. The control plane remains the same as working with standard OVS.

OVS bridges are created by default upon first boot of the BlueField after BFB installation.

If manual configuration of the default settings for the OVS bridge is desired, run:

```
systemctl start openvswitch-switch.service
ovs-vsctl add-port ovsbr1 p0
ovs-vsctl add-port ovsbr1 pf0hpf
ovs-vsctl add-port ovsbr2 p1
ovs-vsctl add-port ovsbr2 pf1hpf
```

To verify successful bridging:

```
$ ovs-vsctl show
9f635bd1-a9fd-4f30-9bdc-b3fa21f8940a
    Bridge ovsbr2
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port p1
            Interface p1
        Port pf1sf0
            Interface en3f1pf1sf0
        Port pf1hpf
            Interface pf1hpf
    Bridge ovsbr1
        Port pf0hpf
            Interface pf0hpf
        Port p0
            Interface p0
        Port ovsbr1
            Interface ovsbr1
                type: internal
        Port pf0sf0
            Interface en3f0pf0sf0
    ovs_version: "2.14.1"
```

The host is now connected to the network.

> **Note**
>
> TC-offload is not supported for IPv6 fragment packets. To make IPv6 fragment packets pass through OVS, the MTU of a specific port must be set to equal to or larger than the fragmented packet size. IPv4 fragment packets can be TC-offloaded as their packet size is not checked by OVS.

Verifying Host Connection on Linux

When BlueField is connected to another BlueField on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to p3p1 on the other host, run:

```
$ ifconfig p3p1 192.168.200.1/24 up
```

2. On the host to which BlueField is connected, run:

```
$ ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the BlueField pinging the host:

```
$ ping 192.168.200.1
```

Verifying Connection from Host to BlueField

There are two SFs configured on the BlueFIeld-2 device, enp3s0f0s0 and enp3s0f1s0, and their representors are part of the built-in bridge. These interfaces will get IP addresses from the DHCP server if it is present. Otherwise it is possible to configure IP address from the host. It is possible to access BlueField via the SF netdev interfaces.

For example:

1. Verify the default OVS configuration. Run:

```
# ovs-vsctl show
5668f9a6-6b93-49cf-a72a-14fd64b4c82b
   Bridge ovsbr1
      Port pf0hpf
         Interface pf0hpf
      Port ovsbr1
         Interface ovsbr1
            type: internal
```

```
        Port p0
            Interface p0
        Port en3f0pf0sf0
            Interface en3f0pf0sf0
    Bridge ovsbr2
        Port en3f1pf1sf0
            Interface en3f1pf1sf0
        Port ovsbr2
            Interface ovsbr2
                type: internal
        Port pf1hpf
            Interface pf1hpf
        Port p1
            Interface p1
    ovs_version: "2.14.1"
```

2. Verify whether the SF netdev received an IP address from the DHCP server. If not, assign a static IP. Run:

```
# ifconfig enp3s0f0s0
enp3s0f0s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.200.125  netmask 255.255.255.0  broadcast 192.168.200.255
        inet6 fe80::8e:bcff:fe36:19bc  prefixlen 64  scopeid 0x20<link>
        ether 02:8e:bc:36:19:bc  txqueuelen 1000  (Ethernet)
        RX packets 3730  bytes 1217558 (1.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 2220 (2.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. Verify the connection of the configured IP address. Run:

```
# ping 192.168.200.25 -c 5
PING 192.168.200.25 (192.168.200.25) 56(84) bytes of data.
64 bytes from 192.168.200.25: icmp_seq=1 ttl=64 time=0.228 ms
64 bytes from 192.168.200.25: icmp_seq=2 ttl=64 time=0.175 ms
64 bytes from 192.168.200.25: icmp_seq=3 ttl=64 time=0.232 ms
64 bytes from 192.168.200.25: icmp_seq=4 ttl=64 time=0.174 ms
64 bytes from 192.168.200.25: icmp_seq=5 ttl=64 time=0.168 ms
```

```
--- 192.168.200.25 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.168/0.195/0.232/0.031 ms
```

Verifying Host Connection on Windows

Set IP address on the Windows side for the RShim or Physical network adapter, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> New-NetIPAddress -InterfaceAlias "Ethernet 16" -IPAddress "192.168.100.1" -PrefixLength 22
```

To get the interface name, please run the following command in Command Prompt:

```
PS C:\Users\Administrator> Get-NetAdapter
```

Output should give us the interface name that matches the description (e.g. NVIDIA BlueField Management Network Adapter).

```
Ethernet 2          NVIDIA ConnectX-4 Lx Ethernet Adapter      6 Not Present  24-8A-07-0D-E8-1D
Ethernet 6          NVIDIA ConnectX-4 Lx Ethernet Ad...#2     23 Not Present  24-8A-07-0D-E8-1C
Ethernet 16         NVIDIA BlueField Management Netw...#2     15 Up           CA-FE-01-CA-FE-02
```

Once IP address is set, Have one ping the other.

```
C:\Windows\system32>ping 192.168.100.2


Pinging 192.168.100.2 with 32 bytes of data:
Reply from 192.168.100.2: bytes=32 time=148ms TTL=64
Reply from 192.168.100.2: bytes=32 time=152ms TTL=64
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

```
Reply from 192.168.100.2: bytes=32 time=158ms TTL=64
```

Enabling OVS HW Offloading

OVS HW offloading is set by default by the /sbin/mlnx_bf_configure script upon first boot after installation.

1. Enable TC offload on the relevant interfaces. Run:

   ```
   $ ethtool -K <PF> hw-tc-offload on
   ```

2. Enable the HW offload: run the following commands (after enabling the HW offload):

   ```
   $ ovs-vsctl set Open_vSwitch . Other_config:hw-offload=true
   ```

3. Restarting OVS is required for the configuration to apply:

   - For Ubuntu:

     ```
     $ systemctl restart openvswitch-switch
     ```

   - For CentOS/RHEL:

     ```
     $ systemctl restart openvswitch
     ```

To show OVS configuration:

```
$ ovs-dpctl show
system@ovs-system:
  lookups: hit:0 missed:0 lost:0
  flows: 0
```

```
masks: hit:0 total:0 hit/pkt:0.00
port 0: ovs-system (internal)
port 1: armbr1 (internal)
port 2: p0
port 3: pf0hpf
port 4: pf0vf0
port 5: pf0vf1
port 6: pf0vf2
```

At this point OVS would automatically try to offload all the rules.

To see all the rules that are added to the OVS datapath:

```
$ ovs-appctl dpctl/dump-flows
```

To see the rules that are offloaded to the HW:

```
$ ovs-appctl dpctl/dump-flows type=offloaded
```

Enabling OVS-DPDK Hardware Offload

1. Remove previously configured OVS bridges. Run:

   ```
   ovs-vsctl del-br <bridge-name>
   ```

   Issue the command ovs-vsctl show to see already configured OVS bridges.

2. Enable the Open vSwitch service. Run:

   ```
   systemctl start openvswitch
   ```

3. Configure huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

4. Configure DPDK socket memory and limit. Run:

```
# ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-limit=2048
# ovs-vsctl set Open_vSwitch . other_config:dpdk-socket-mem=2048
```

5. Enable hardware offload (disabled by default). Run:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-offload=true
```

6. Configure the DPDK whitelist. Run:

```
ovs-vsctl set Open_vSwitch . other_config:dpdk-extra="-a 0000:03:00.0,representor=
[0,65535],dv_flow_en=1,dv_xmeta_en=1,sys_mem_en=1"
```

7. Create OVS-DPDK bridge. Run:

```
ovs-vsctl add-br br0-ovs -- set Bridge br0-ovs datapath_type=netdev -- br-set-external-id br0-ovs
bridge-id br0-ovs -- set bridge br0-ovs fail-mode=standalone
```

8. Add PF to OVS. Run:

```
ovs-vsctl add-port br0-ovs p0 -- set Interface p0  type=dpdk options:dpdk-devargs=0000:03:00.0
```

9. Add representor to OVS. Run:

```
ovs-vsctl add-port br0-ovs pf0vf0 -- set Interface pf0vf0 type=dpdk options:dpdk-devargs=0000:03:00.0,representor=[0]
ovs-vsctl add-port br0-ovs pf0hpf -- set Interface pf0hpf type=dpdk options:dpdk-devargs=0000:03:00.0,representor=[65535]
```

10. Restart the Open vSwitch service. This step is required for HW offload changes to take effect.
    - For CentOS, run:

      ```
      systemctl restart openvswitch
      ```

    - For Debian/Ubuntu, run:

      ```
      systemctl restart openvswitch-switch
      ```

For a reference setup configuration for BlueField-2 devices, refer to the article "Configuring OVS-DPDK Offload with BlueField-2".

Configuring DPDK and Running TestPMD

1. Configure hugepages. Run:

   ```
   echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

2. Run testpmd.
   - For Ubuntu/Debian:

     ```
     env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib/aarch64-linux-gnu /opt/mellanox/dpdk/bin/dpdk-testpmd -a 03:00.0,representor=[0,65535] --socket-
     ```

```
mem=1024 -- --total-num-mbufs=131000 -i
```

- For CentOS:

```
env LD_LIBRARY_PATH=/opt/mellanox/dpdk/lib64/ /opt/mellanox/dpdk/bin/dpdk-testpmd
-a 03:00.0,representor=[0,65535] --socket-mem=1024 -- --total-num-mbufs=131000 -i
```

For a detailed procedure with port display, refer to the article "Configuring DPDK and Running testpmd on BlueField-2".

Flow Statistics and Aging

The aging timeout of OVS is given in milliseconds and can be configured by running the following command:

```
$ ovs-vsctl set Open_vSwitch . other_config:max-idle=30000
```

Connection Tracking Offload

This feature enables tracking connections and storing information about the state of these connections. When used with OVS, BlueField can offload connection tracking, so that traffic of established connections bypasses the kernel and goes directly to hardware.

Both source NAT (SNAT) and destination NAT (DNAT) are supported with connection tracking offload.

Configuring Connection Tracking Offload

This section provides an example of configuring OVS to offload all IP connections of host PF0.

1. Enable OVS HW offloading.

2. Create OVS connection tracking bridge. Run:

```
$ ovs-vsctl add-br ctBr
```

3. Add p0 and pf0hpf to the bridge. Run:

```
$ ovs-vsctl add-port ctBr p0
$ ovs-vsctl add-port ctBr pf0hpf
```

4. Configure ARP packets to behave normally. Packets which do not comply are routed to table1. Run:

```
$ ovs-ofctl add-flow ctBr "table=0,arp,action=normal"
$ ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)"
```

5. Configure RoCEv2 packets to behave normally. RoCEv2 packets follow UDP port 4791 and a different source port in each direction of the connection. RoCE traffic is not supported by CT. In order to run RoCE from the host add the following line before ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1)":

```
$ ovs-ofctl add-flow ctBr table=0,udp,tp_dst=4791,action=normal
```

   This rule allows RoCEv2 UDP packets to skip connection tracking rules.

6. Configure the new established flows to be admitted to the connection tracking bridge and to then behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+new,action=ct(commit),normal"
```

7. Set already established flows to behave normally. Run:

```
$ ovs-ofctl add-flow ctBr "table=1,priority=1,ip,ct_state=+trk+est,action=normal"
```

Connection Tracking With NAT

This section provides an example of configuring OVS to offload all IP connections of host PF0, and performing source network address translation (SNAT). The server host sends traffic via source IP from 2.2.2.1 to 1.1.1.2 on another host. Arm performs SNAT and changes the source IP to 1.1.1.16. Note that static ARP or route table must be configured to find that route.

1. Configure untracked IP packets to do nat. Run:

   ```
   ovs-ofctl add-flow ctBr "table=0,ip,ct_state=-trk,action=ct(table=1,nat)"
   ```

2. Configure new established flows to do SNAT, and change source IP to 1.1.1.16. Run:

   ```
   ovs-ofctl add-flow ctBr
   "table=1,in_port=pf0hpf,ip,ct_state=+trk+new,action=ct(commit,nat(src=1.1.1.16)), p0"
   ```

3. Configure already established flows act normal. Run:

   ```
   ovs-ofctl add-flow ctBr "table=1,ip,ct_state=+trk+est,action=normal"
   ```

   Conntrack shows the connection with SNAT applied. Run `conntrack -L` for Ubuntu 22.04 kernel or `cat /proc/net/nf_conntrack` for older kernel versions. Example output:

   ```
   ipv4     2 tcp      6 src=2.2.2.1 dst=1.1.1.2 sport=34541 dport=5001 src=1.1.1.2 dst=1.1.1.16
   sport=5001 dport=34541 [OFFLOAD] mark=0 zone=1 use=3
   ```

Querying Connection Tracking Offload Status

Start traffic on PF0 from the server host (e.g., iperf) with an external network. Note that only established connections can be offloaded. TCP should have already finished the handshake, UDP should have gotten the reply.

> **Note**
>
> ICMP is not currently supported.

To check if specific connections are offloaded from Arm, run `conntrack -L` for Ubuntu 22.04 kernel or `cat /proc/net/nf_conntrack` for older kernel versions.

The following is example output of offloaded TCP connection:

```
ipv4     2 tcp      6 src=1.1.1.2 dst=1.1.1.3 sport=51888 dport=5001 src=1.1.1.3 dst=1.1.1.2 sport=5001 dport=51888 [HW_OFFLOAD] mark=0 zone=0 use=3
```

Performance Tune Based on Traffic Pattern

Offloaded flows (including connection tracking) are added to virtual switch FDB flow tables. FDB tables have a set of flow groups. Each flow group saves the same traffic pattern flows. For example, for connection tracking offloaded flow, TCP and UDP are different traffic patterns which end up in two different flow groups.

A flow group has a limited size to save flow entries. By default, the driver has 4 big FDB flow groups. Each of these big flow groups can save at most 4000000/(4+1)=800k different 5-tuple flow entries. For scenarios with more than 4 traffic patterns, the driver provides a module parameter (`num_of_groups`) to allow customization and performance tune.

Note

The size of each big flow groups can be calculated according to formula: size = 4000000/(num_of_groups+1)

To change the number of big FDB flow groups, run:

```
$ echo <num_of_groups> > /sys/module/mlx5_core/parameters/num_of_groups
```

The change takes effect immediately if there is no flow inside the FDB table (no traffic running and all offloaded flows are aged out), and it can be dynamically changed without reloading the driver.

If there are residual offloaded flows when changing this parameter, then the new configuration only takes effect after all flows age out.

Connection Tracking Aging

Aside from the aging of OVS, connection tracking offload has its own aging mechanism with a default aging time of 30 seconds.

Maximum Tracked Connections

Note

The maximum number for tracked offloaded connections is limited to 1M by default.

The OS has a default setting of maximum tracked connections which may be configured by running:

```
$ /sbin/sysctl -w net.netfilter.nf_conntrack_max=1000000
```

This changes the maximum tracked connections (both offloaded and non-offloaded) setting to 1 million.

The following option specifies the limit on the number of offloaded connections. For example:

```
# devlink dev param set pci/${pci_dev} name ct_max_offloaded_conns value $max cmode runtime
```

This value is set to 1 million by default from BlueFiled. Users may choose a different number by using the `devlink` command.

> **Note**
>
> Make sure `net.netfilter.nf_conntrack_tcp_be_liberal=1` when using connection tracking.

Offloading VLANs

OVS enables VF traffic to be tagged by the virtual switch.

For BlueField, the OVS can add VLAN tag (VLAN push) to all the packets sent by a network interface running on the host (either PF or VF) and strip the VLAN tag (VLAN pop) from the traffic going from the wire to that interface. Here we operate in Virtual Switch Tagging (VST) mode. This means that the host/VM interface is unaware of the VLAN tagging. Those rules can also be offloaded to the HW embedded switch.

To configure OVS to push/pop VLAN you need to add the tag=$TAG section for the OVS command line that adds the representor ports. So if you want to tag all the traffic of VF0 with VLAN ID 52, you should use the following command when adding its representor to the bridge:

```
$ ovs-vsctl add-port armbr1 pf0vf0 tag=52
```

Note

If the virtual port is already connected to the bridge prior to configuring VLAN, you would need to remove it first:

```
$ ovs-vsctl del-port pf0vf0
```

In this scenario all the traffic being sent by VF 0 will have the same VLAN tag. We could set a VLAN tag by flow when using the TC interface, this is explained in section "Using TC Interface to Configure Offload Rules".

VXLAN Tunneling Offload

VXLAN tunnels are created on the Arm side and attached to the OVS. VXLAN decapsulation/encapsulation behavior is similar to normal VXLAN behavior, including over `hw_offload=true`.

To allow VXLAN encapsulation, the uplink representor (`p0`) should have an MTU value at least 50 bytes greater than that of the host PF/VF. Please refer to "Configuring Uplink MTU" for more information.

Configuring VXLAN Tunnel

1. Consider `p0` to be the local VXLAN tunnel interface (or VTEP).

> **Note**
>
> To be consistent with the examples below, it is assumed that p0 is configured with a 1.1.1.1 IPv4 address.

2. Remove p0 from any OVS bridge.

3. Build a VXLAN tunnel over OVS arm-ovs. Run:

```
ovs-vsctl add-br arm-ovs -- add-port arm-ovs vxlan11 -- set interface vxlan11 type=vxlan
options:local_ip=1.1.1.1 options:remote_ip=1.1.1.2 options:key=100
options:dst_port=4789
```

4. Connect any host representor (e.g., pf0hpf) for which VXLAN is desired to the same arm-ovs bridge.

5. Configure the MTU of the VTEP (p0) used by VXLAN to at least 50 bytes larger than the host representor's MTU.

At this point, the host is unaware of any VXLAN operations done by the BlueField's OVS. If the remote end of the VXLAN tunnel is properly set, any network traffic traversing arm-ovs undergoes VXLAN encap/decap.

Querying OVS VXLAN hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
in_port(2),eth(src=ae:fd:f3:31:7e:7b,dst=a2:fb:09:85:84:48),eth_type(0x0800), packets:1, bytes:98,
used:0.900s, actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,tp_dst=4789,flags(key))),3
tunnel(tun_id=0x64,src=1.1.1.2,dst=1.1.1.1,tp_dst=4789,flags(+key)),in_port(3),eth(src=a2:fb:09:85:84:48,ds
packets:75, bytes:7350, used:0.900s, actions:2
```

Note

For the host PF, in order for VXLAN to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (pf0hpf in the example above) of the VXLAN tunnel must be smaller than the MTU of the tunnel interfaces (p0) to account for the size of the VXLAN headers. For example, you can set the MTU of P0 to 2000.

GRE Tunneling Offload

GRE tunnels are created on the Arm side and attached to the OVS. GRE decapsulation/encapsulation behavior is similar to normal GRE behavior, including over hw_offload=true.

To allow GRE encapsulation, the uplink representor (p0) should have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

Configuring GRE Tunnel

1. Consider p0 to be the local GRE tunnel interface. p0 should not be attached to any OVS bridge.

Note

To be consistent with the examples below, it is assumed that p0 is configured with a 1.1.1.1 IPv4 address and that the remote end of the tunnel is 1.1.1.2.

2. Create an OVS bridge, br0, with a GRE tunnel interface, gre0. Run:

```
ovs-vsctl add-port br0 gre0 -- set interface gre0 type=gre options:local_ip=1.1.1.1
options:remote_ip=1.1.1.2 options:key=100
```

3. Add pf0hpf to br0.

```
ovs-vsctl add-port br0 pf0hpf
```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GRE processing inside the BlueField OS.

Querying OVS GRE hw_offload Rules

Run the following:

```
ovs-appctl dpctl/dump-flows type=offloaded
recirc_id(0),in_port(3),eth(src=50:6b:4b:2f:0b:74,dst=de:d0:a3:63:0b:30),eth_type(0x0800),ipv4(frag=no),
packets:878, bytes:122802, used:0.440s,
actions:set(tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,ttl=64,flags(key))),2
tunnel(tun_id=0x64,src=1.1.1.1,dst=1.1.1.2,flags(+key)),recirc_id(0),in_port(2),eth(src=de:d0:a3:63:0b:30,dst
packets:995, bytes:97510, used:0.440s, actions:3
```

Note

For the host PF, in order for GRE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (pf0hpf in the example above) of the GRE tunnel must be smaller than the MTU of the tunnel interfaces (p0) to account for the size of the GRE headers. For example, you can set the MTU of P0 to 2000.

GENEVE Tunneling Offload

GENEVE tunnels are created on the Arm side and attached to the OVS. GENEVE decapsulation/encapsulation behavior is similar to normal GENEVE behavior, including over hw_offload=true.

To allow GENEVE encapsulation, the uplink representor (p0) must have an MTU value at least 50 bytes greater than that of the host PF/VF.

Please refer to "Configuring Uplink MTU" for more information.

Configuring GENEVE Tunnel

1. Consider p0 to be the local GENEVE tunnel interface. p0 should not be attached to any OVS bridge.

2. Create an OVS bridge, br0, with a GENEVE tunnel interface, gnv0. Run:

```
ovs-vsctl add-port br0 gnv0 -- set interface gnv0 type=geneve options:local_ip=1.1.1.1
```

```
options:remote_ip=1.1.1.2 options:key=100
```

3. Add pf0hpf to br0.

```
ovs-vsctl add-port br0 pf0hpf
```

4. At this point, any network traffic sent or received by the host's PF0 undergoes GENEVE processing inside the BlueField OS.

Options are supported for GENEVE. For example, you may add option 0xea55 to tunnel metadata, run:

```
ovs-ofctl add-tlv-map geneve_br "{class=0xffff,type=0x0,len=4}->tun_metadata0"
ovs-ofctl add-flow geneve_br ip,actions="set_field:0xea55->tun_metadata0",normal
```

Note

For the host PF, in order for GENEVE to work properly with the default 1500 MTU, follow these steps.

1. Disable host PF as the port owner from Arm (see section "Zero-trust Mode"). Run:

```
$ mlxprivhost -d /dev/mst/mt41682_pciconf0 --disable_port_owner r
```

2. The MTU of the end points (pf0hpf in the example above) of the GENEVE tunnel must be smaller than the MTU of the tunnel interfaces (p0) to account for the size of the GENEVE headers. For example, you can set the MTU of P0 to 2000.

Using TC Interface to Configure Offload Rules

Offloading rules can also be added directly, and not just through OVS, using the tc utility. To enable TC ingress on all the representors (i.e., uplink, PF, and VF).

```
$ tc qdisc add dev p0 ingress
$ tc qdisc add dev pf0hpf ingress
$ tc qdisc add dev pf0vf0 ingress
```

L2 Rules Example

The rule below drops all packets matching the given source and destination MAC addresses.

```
$ tc filter add dev pf0hpf protocol ip parent ffff: \
                        flower \
                                skip_sw \
                                dst_mac e4:11:22:11:4a:51 \
                                src_mac e4:11:22:11:4a:50 \
                         action drop
```

VLAN Rules Example

The following rules push VLAN ID 100 to packets sent from VF0 to the wire (and forward it through the uplink representor) and strip the VLAN when sending the packet to the VF.

```
$ tc filter add dev pf0vf0 protocol 802.1Q parent ffff: \
                            flower \
                                    skip_sw \
                                    dst_mac e4:11:22:11:4a:51 \
                                    src_mac e4:11:22:11:4a:50 \
                            action vlan push id 100 \
                            action mirred egress redirect dev p0
```

```
$ tc filter add dev p0 protocol 802.1Q parent ffff: \
                        flower \
                                skip_sw \
                                dst_mac e4:11:22:11:4a:51 \
                                src_mac e4:11:22:11:4a:50 \
                                vlan_ethtype 0x800 \
                                vlan_id 100 \
                                vlan_prio 0 \
                       action vlan pop \
                       action mirred egress redirect dev pf0vf0
```

VXLAN Encap/Decap Example

```
$ tc filter add dev pf0vf0 protocol 0x806 parent ffff: \
                        flower \
                                skip_sw \
                                dst_mac e4:11:22:11:4a:51 \
                                src_mac e4:11:22:11:4a:50 \
                       action tunnel_key set \
                       src_ip 20.1.12.1 \
                       dst_ip 20.1.11.1 \
                       id 100 \
                       action mirred egress redirect dev vxlan100

$ tc filter add dev vxlan100 protocol 0x806 parent ffff: \
                        flower \
                                skip_sw \
                                dst_mac e4:11:22:11:4a:51 \
                                src_mac e4:11:22:11:4a:50 \
                                enc_src_ip 20.1.11.1 \
                                enc_dst_ip 20.1.12.1 \
                                enc_key_id 100 \
                                enc_dst_port 4789 \
                       action tunnel_key unset \
                       action mirred egress redirect dev pf0vf0
```

VirtIO Acceleration Through Hardware vDPA

For configuration procedure, please refer to the MLNX_OFED documentation under OVS Offload Using ASAP² Direct > VirtIO Acceleration through Hardware vDPA.