Miscellaneous (Runtime)

Table of contents

DOCA Glossary 3

DOCA Crypto Acceleration 11

DOCA Services Fluent Logger 13

DOCA DPU CLI 18

DOCA Switching 24

- DPU Kernel Representors Model 30
- Virtio Acceleration through Hardware vDPA 33
- Bridge Offload 38
- Link Aggregation 40
- Controlling Host PF and VF Parameters 51
  - Configuring Uplink MTU 53
- DPDK on BlueField 53

DOCA with OpenSSL 54

BlueField Scalable Function User Guide 59

DOCA TLS Offload Guide 75

DOCA Troubleshooting 101

- NVIDIA BlueField Reset and Reboot Procedures 101

DOCA Virtual Functions User Guide 113

BlueField SR-IOV 122

fTPM over OP-TEE 124

This section contains the following pages:

- DOCA Glossary
- DOCA Crypto Acceleration
- DOCA Services Fluent Logger
- DOCA DPU CLI
- DOCA Switching
- DOCA with OpenSSL
- BlueField Scalable Function User Guide
- DOCA TLS Offload Guide
- DOCA Troubleshooting
- DOCA Virtual Functions User Guide
- BlueField SR-IOV
- fTPM over OP-TEE

DOCA Glossary

Term	Description
ACS	Access control services
ASAP^2	Accelerated Switching and Packet Processing
ASN	Autonomous system number
ATF	Arm-trusted firmware
b	Lower-case b is used to indicate size in bits or multiples of bits (e.g., 1Kb = 1024 bits)
B	Upper-case B is used to indicate size in bytes or multiples of bytes (e.g., 1KB = 1024 bytes, 1MB = 1048576 bytes)
BAR	Base address register
BDF address	Bus, device, function address. This is the device's PCIe bus address to uniquely identify the specific device.
BFB	BlueField bootstream
BGP	Border gateway protocol
BMC	Board management controller
BUF	Buffer
BSP	BlueField support package
CBS	Committed burst size
CIR	Committed information rate
CMDQ	Command queue
CPDS	Control pipe dynamic size
CQE	Completion queue events
CTX	Context
DEK	Data encryption key
DMA	Direct memory access
DN	Data network

Term	Description
DOCA	DPU SDK
DPA	Data path accelerator; a n auxiliary processor designed to accelerate data-path operations
DPCP	Direct packet control plane
DPDK	Data plane development kit
DPI	Deep packet inspection
DPIF	Datapath offload interface
DPU	Data processing unit, the third pillar of the data center with CPU and GPU. BlueField is available as a DPU and as a SuperNIC.
DW	Dword
EBS	Excess burst size
ECE	Enhanced connection establishment
ECMP	Equal-cost multi-path
ECPF	Embedded CPU physical function
EIR	Excess information rate
EM	Exact match
eMMC	Embedded multi-media card
ESP	EFI system partition
ESP	Encapsulating security payload
EU	Execution unit. HW thread; a logical DPA processing unit.
FAR	Forwarding action rule
FLR	Function level reset
FIFO	First-in-first-out
FIPS	Federal Information Processing Standards
FPGA	Field-programmable gate arrays
FW	Firmware
GDAKIN	GPUDirect async kernel-initiated network
GDB	GNU debugger

Term	Description
gNB	Next Generation NodeB
HCA	Host-channel adapter – an IB device that terminates an IB link and executes transport functions. This may be an HCA (host CA) or a TCA (target CA).
HCA card	A network adapter card based on an InfiniBand channel adapter device
Host	When referring to "the host" this documentation is referring to the **server host**. When referring to the Arm based host, the documentation will specifically call out "Arm host". • Server host OS refers to the Host Server OS (Linux or Windows) • Arm host refers to the AARCH64 Linux OS which is running on the BlueField Arm Cores
HW	Hardware
hwmon	Hardware monitoring
IB	InfiniBand
IB Cluster/Fabric/ Subnet	A set of IB devices connected by IB cables
IB device	An integrated circuit implementing InfiniBand compliant communication
ICM	Interface configuration memory
ICV	Integrity check value
IDE	Integrated development environment
IKE	Internet key exchange
In-band	A term assigned to administration activities traversing the IB connectivity only
IPoIB	IP over InfiniBand
IR	Intermediate representation
IRQ	Interrupt request
iSER	iSCSI RDMA protocol
KPI	Key performance indicator
LFT	Unicast linear forwarding tables – a table that exists in every switch providing the port through which packets should be sent to each LID

Term	Description
Local device/node/system	The IB HCA card installed on the machine running IBDIAG tools
Local identifier	An address assigned to a port (data sink or source point) by the Subnet Manager, unique within the subnet, used for directing packets within the subnet
Local port	The InfiniBand port of the HCA through which IBDIAG tools connect to the IB fabric
LRO	Large receive offload
lsb	Least significant bit
LSB	Least significant byte
LSO	Large send offload
LTO	Link-time optimization
MFT	Mellanox firmware tools
MLNX_OFED	Mellanox OpenFabrics Enterprise Distribution
MPI	Message passing interface
msb	Most significant bit
MSB	Most significant byte
MSI-X	Message signaled interrupts extended
MSS	Maximum segment size
MSS	Memory subsystem
MST	Mellanox software tools
MTU	Maximum transmission unit
Multicast forwarding tables	A table that exists in every switch providing the list of ports to forward received multicast packet. The table is organized by MLID.
NAT	Network address translation
NIC	Network interface card – a network adapter card that plugs into the PCI Express slot and provides one or more ports to an Ethernet network

Term	Description
NIST	National Institute of Standards and Technology
NP	Notification point
NS	Namespace
NUMA	Non-uniform memory access
OOB	Out-of-band
OS	Operating system
OVS	Open vSwitch
PAT	Port address translation
PBA	Pending bit array
PBS	Peak burst size
PCIe	PCI Express; Peripheral Component Interconnect Express
PDR	Packet detection rule
PF	Physical function
PFC	Priority flow control
PE	Progress engine
PHC	Physical hardware clock
PIR	Peak information rate
PK	Platform key
PKA	Public key accelerator
POC	Proof of concept
QER	QoS enforcement rule
QoS	Quality of service
PR	Path record
PUD	Process under debug
RAN	Radio access network
RAP	Reference application
RD	Route distinguisher
RDMA	Remote direct memory access

Term	Description
RDMA CM	RDMA connection manager
RegEx	Regular expression
REQ	Request
RES	Response
RN	Request node RN-F – Fully coherent request node RN-D – IO coherent request node with DVM support RN-I – IO coherent request node
RNG	Random number generator/generation
RoCE	RDMA over converged Ethernet
RP	Reaction point
RQ	Receive queue
RShim	Random shim
RSP	Remote serial protocol
RT	Route target
RTOS	Real-time operating system
RTT	Round-trip time
RX	Receive
RXP	Regular expression processor
SA	Subnet administrator – an application (normally part of the Subnet Manager) that implements the interface for querying and manipulating subnet management data
SA	Security association
SBSA	Server base system architecture
SDK	Software development kit
SF	Sub-function or scalable function
SFC	Service function chaining (HBN)
SG	Scatter-gather
SHA	Secure hash algorithm

Term	Description
SL	Service level
SM	• Subnet Manager – One of several entities involved in the configuration and control of the IB fabric • Master SM – The SM which is authoritative and has the reference configuration information for the subnet • Standby SM – An SM which is currently quiescent and not in the role of a master SM, by the agency of the master SM
SMF	Session management function
SN	Sequence number
SNAP	Storage-defined network-accelerated processing
SNAT	Source NAT
SPDK	Storage performance development kit
SPI	Security parameters index
SQ	Send queue
SR-IOV	Single-root IO virtualization
SRP	SCSI RDMA protocol
SuperNIC	a configuration of a DPU that is specific for E-W networking. BlueField has a SuperNIC configuration
SVI	Switch virtual interface
SW	Software
Sync event	Synchronization event
TAI	International Atomic Time
TIR	Transport interface receive
TIS	Transport interface send
TLS	Transport layer security
TSO	TCP segmentation offload
TX	Transmit
uDAPL	User direct access programming library
UDS	Unix domain socket

Term	Description
UE	User equipment
UEFI	Unified extensible firmware interface
ULP	Upper layer protocol
UPF	User-plane function
URR	Usage reporting rule
UTC	Coordinated Universal Time
vDPA	Virtual data path acceleration
VF	Virtual function
VFE	Virtio full emulation
vHBA	Virtual SCSI host bus adapter
VL	Virtual lane
VM	Virtual machine
VMA	NVIDIA® Messaging Accelerator
VNI	• Virtual network identifier • VXLAN network identifier
VPI	Virtual protocol interconnect – a n NVIDIA technology that allows NVIDIA channel adapter devices (ConnectX®) to simultaneously connect to an InfiniBand subnet and a 10GigE subnet (each subnet connects to one of the adapter ports)
VRF	Virtual routing and forwarding
VTEP	VXLAN tunnel endpoint
WAN	Wide area network
WorkQ or workq	Work queue
WQE	Work queue elements
WR	Write
XLIO	NVIDIA® Accelerated IO

DOCA Crypto Acceleration

NVIDIA® BlueField® DPU incorporates several Public Key Acceleration (PKA) engines to offload the processor of the Arm host, providing high-performance computation of PK algorithms. BlueField's PKA is useful for a wide range of security applications. It can assist with SSL acceleration, or a secure high-performance PK signature generator/checker and certificate related operations.

BlueField's PKA software libraries implement a simple, complete framework for crypto public key infrastructure (PKI) acceleration. It provides direct access to hardware resources from the user space, and makes available a number of arithmetic operations—some basic (e.g., addition and multiplication), and some complex (e.g., modular exponentiation and modular inversion)—and high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptography, and the Federal Digital Signature Algorithm (DSA as documented in FIPS-186) public-private key systems.

Some of the use cases for the BlueField PKA involve integrating OpenSSL software applications with BlueField's PKA hardware. The BlueField PKA dynamic engine for OpenSSL allows applications integrated with OpenSSL (e.g., StrongSwan) to accomplish a variety of security-related goals and to accelerate the cryptographic processing with the BlueField PKA hardware. OpenSSL versions ⩾1.0.0, ⩽1.1.1, and 3.0.2 are supported.

Note

With CentOS 7.6, only OpenSSL 1.1 (not 1.0) works with PKA engine and keygen. Use `openssl11` with PKA engine and keygen.

The engine supports the following operations:

- RSA
- DH
- DSA
- ECDSA

- ECDH
- Random number generation that is cryptographically secure.

Up to 4096-bit keys for RSA, DH, and DSA operations are supported. Elliptic Curve Cryptography support of (nist) prime curves for 160, 192, 224, 256, 384 and 521 bits.

For example:

To sign a file using BlueField's PKA engine:

```
$ openssl dgst -engine pka -sha256 -sign <privatekey> -out
<signature> <filename>
```

To verify the signature, execute:

```
$ openssl dgst -engine pka -sha256 -verify <publickey> -signature
<signature> <filename>
```

For further details on BlueField PKA, please refer to "PKA Driver Design and Implementation Architecture Document" and/or "PKA Programming Guide". Directions and instructions on how to integrate the BlueField PKA software libraries are provided in the README files on our PKA GitHub.

DOCA Services Fluent Logger

This guide provides instructions on how to use the logging infrastructure for DOCA services on top of NVIDIA® BlueField® DPU.

Introduction

Fluent Bit is a fast log collector that collects information from multiple sources and then forwards the data onward using Fluent.

On NVIDIA DPUs, the Fluent Bit logger can be easily configured to collect system data and the logs from the different DOCA services.

Deployment

The deployment is based on a recommended configuration template for the existing Fluent Bit container.

For information about the deployment of DOCA containers on top of the BlueField DPU, refer to NVIDIA DOCA Container Deployment Guide.

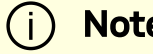
The following is an example YAML file for deploying the Fluent Bit pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: fluent-bit
spec:
  hostNetwork: true
  containers:
  - name: fluent-bit
    image: fluent/fluent-bit:latest
    imagePullPolicy: Always
    # Example resource definitions
    resources:
```

```
      requests:
        memory: "100Mi"
        cpu: "200m"
      limits:
        memory: "200Mi"
        cpu: "300m"
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: config-file
      mountPath: /fluent-bit/etc/fluent-bit.conf
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: config-file
    hostPath:
      path: /opt/mellanox/doca/services/fluent-bit.conf
      type: File
```

As explained in the "Configuration" section, Fluent Bit uses a configuration file. As such, to ensure that the example YAML file is shared from the DPU to the deployed Fluent Bit container, use the following:

```
path: /opt/mellanox/doca/services/fluent-bit.conf
```



Note

The path below is just an example for where the user can place the `fluent-bit.conf` file. The file could be placed in a different

directory on the DPU as long as the YAML file points to the updated location.

Configuration

The Fluent Bit configuration file should have the following sections:

- `[SERVICE]` – to define the service specifications
- `[INPU]` – to define folders to collect logs from (there could be multiple inputs)
- `[OUTPUT]` – IP and port to stream the data to

Example configuration file:

```
[SERVICE]
    Flush         2
    Log_Level     info
    Daemon        off
    Parsers_File  parsers.conf
    HTTP_Server   On
    HTTP_Listen   0.0.0.0
    HTTP_Port     2020

[INPUT]
    Name              tail
    Tag               kube.*
    Path              /var/log/containers/*.log
   Parser           docker
   Mem_Buf_Limit     5MB
   Skip_Long_Lines   On
   Refresh_Interval  10

[INPUT]
   Name            tail
   Tag             sys.*
```

```
    Path        /var/log/doca/*/*.log
    Mem_Buf_Limit     5MB
    Skip_Long_Lines   On
    Refresh_Interval  10

[OUTPUT]
    Name  es
    Match *
    Host 10.20.30.40
    Port 9201
    Index fluent_bit
    Type cpu_metrics
```

> **Note**
>
> The most important field to pay attention to is `Path` for the `INPUT` section. DOCA services report their logs to a unique directory under `/var/log/doca/<service_name>/*.log` per the respective DOCA service. As such, the configuration above defines the `/var/log/doca/*/*.log` input definition.

More information about the full specifications can be found in the official Fluent Bit manual.

Troubleshooting

For container-related troubleshooting, refer to the "Troubleshooting" section in the NVIDIA DOCA Container Deployment Guide.

For general troubleshooting, refer to the DOCA Troubleshooting.

When copying the above YAML file, it is possible that the container infrastructure logs give an error related to RFC 1123". These errors are usually a result of a spacing error in the file, which sometimes occur when copying the file as is from this page. To fix this

issue, make sure that only the space character (' ') is used as a spacer in the file and not other whitespace characters that might have been added during the copy operation.

DOCA DPU CLI

This guide provides quick access to a useful set of CLI commands and utilities on the NVIDIA® BlueField® DPU environment.

Introduction

This guide provides a concise guide on useful commands for DOCA deployment and configuration.

The tables in this guide provide two categories of commands:

- General commands for Linux/networking environment
- DOCA/DPU-specific commands

> **Note**
>
> For more information about these commands, such as usage instructions, flag options, arguments and so on, use the `-h` option after the command or use the manual (e.g., `man lspci`).

General Commands

Command	Description
`ifconfig`	Used to configure kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed. If no arguments are given, `ifconfig` displays the status of the currently active interfaces. If a single interface argument is given, it displays the status of the given interface only. If a single `-a` argument is given, it displays the status of all interfaces, even those that are down. Otherwise, it configures an interface.

Command	Description
`ethtool <devname>`	Used to query and control network device driver and hardware settings, particularly for wired Ethernet devices. `<devname>` is the name of the network device on which `ethtool` should operate. **Note** This command shows the speed of the network card of the DPU.
`lspci`	Displays information about PCIe buses in the system and devices connected to them. By default, it shows a brief list of devices.
`tcpdump`	Dump traffic on a network. Usage: `tcpdump -i <interface>` where `<interface>` is any port interface (physical/SF rep/VF port rep).
`ovs-vsctl`	Utility for querying and configuring `ovs-vswitchd`. The `ovs-vsctl` program supports the model of a bridge implemented by Open vSwitch in which a single bridge supports ports on multiple VLANs.
`mount 10.0.0.10:/vol/myshare/ myshare/`	Used for mounting a work directory on the DPU. **Note** Must be used after creating a new directory named `myshare` under root (i.e., `mkdir /myshare`)
`scp`	Secure copy (remote file copy program). Useful for copying files from BlueField to the host and vice versa.
`iperf`	Used for server-client connection. Useful to check if the network connection achieves the speed of the network card on the DPU (line rate).

DPU/DOCA Commands

Command	Description
`ibdev2netdev`	Displays available `mlnx` interfaces
`mst`	Used to start MST service, to stop it, and for other operations with NVIDIA devices like reset and enabling remote access
`cat /etc/mlnx-release`	Displays the full BlueField image (bfb) version
`cat /etc/os-release`	Displays the details of the underlying OS installed on BlueField
`ibv_devinfo`	Displays the current InfiniBand connected devices and relevant information. Useful for checking current firmware version.
`ipmitool power cycle`	Power cycle **Note** Prior to performing a power cycle, make sure to perform a graceful shutdown.

`echo '1024' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages`
`sudo mkdir /mnt/huge`
`sudo mount -t hugetlbfs -o pagesize=2M nodev /mnt/huge`
	DPDK setup. Allocates hugepages for DPDK environment abstraction layer (EAL).

`mlxdevm tool`	`mlxdevm tool` is found under `/opt/mellanox/iproute2/sbin/`. With this tool it is possible to create an SF and set its state to active, configure a HW address and set it to trusted, deploy the created SF and print info about it.

Command	Description
`/opt/mellanox/iproute2/sbin/mlxdevm port add pci/<pci_address> flavour pcisf pfnum <correspondig_physical_function_number> sfnum <unique_sf_number>`	Creates an SF in the flavor of the given PF with the given unique SF number. Example: `` /opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03`:00.0 flavour pcisf pfnum 0 sfnum 4 ``
`/opt/mellanox/iproute2/sbin/mlxdevm port show`	Displays information about the available SFs
`/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/<sf_index> hw_addr <HW_address> trust on state active`	Configures SF capabilities such as setting the HW address, making it "trusted", and setting its state to active. `<sf_index>` the SF. To obtain this index, you may run `mlxdevm port show`. Example: `/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/0000:03:00.0/229377 hw_addr 02:25:f2:8d:a2:4c trust on state active`
`$ echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind` `$ echo mlx5_core.sf.<next_serial> > /sys/bus/auxiliary/drivers/mlx5_core.sf/bind`	These two commands deploy the created SF. The first command unbinds the SF from the default driver, while the second command binds the SF to the actual driver. The deployment phase should be done after the capabilities of the SF are configured. The SF is identified by `<next_serial>` which can be obtained by running the command below.

Command	Description
`ls /sys/bus/auxiliary/devices/mlx5_core.sf.*`	Displays additional information about the created SFs and their "next serial numbers". For example, if `mlx5_core.sf.2` exists in the output of the command, then running `cat /sys/bus/auxiliary/devices/mlx5_core.sf.2/sfnum` would output the sfnum related to `mlx5_core.sf.2`.
`/opt/mellanox/iproute2/sbin/mlxdevm port function set pci/<pci_address>/<sf_index> state inactive` `/opt/mellanox/iproute2/sbin/mlxdevm port del pci/<pci_address>/<sf_index>`	These two commands must be executed to delete a given SF. First, users must set the state of the SF to inactive, and only then should it be deleted.
`/opt/mellanox/iproute2/sbin/mlxdevm port help`	Displays additional information about operations that can be used on created SF ports
`crictl pods`	Displays currently active K8S pods, and their IDs (it might take up to 20-30 seconds for the pod to start)
`crictl ps`	Displays currently active containers and their IDs
`crictl ps -a`	Displays all containers, including containers that recently finished their execution
`crictl logs <container-id>`	Examines the logs of a given container
`crictl exec -it <container-id> /bin/bash`	Attaches a shell to a running container
`journalctl -u kubelet`	Examines the Kubelet logs. Useful when a pod/container fails to spawn.

Command	Description
`crictl stopp <pod-id>`	Stops a running K8S pod
`crictl stop <container-id>`	Stops a running container
`crictl rmi <image-id>`	Removes a container image from the local K8S registry

DOCA Switching

NVIDIA® BlueField® and NVIDIA® ConnectX® platforms provide robust support for diverse applications through hardware-based offloads, offering unparalleled scalability, performance, and efficiency.

This section lists the extensive switching capabilities enabled by DOCA libraries and services on these platforms. It includes detailed configurations of Open Virtual Switch (OVS) such as the setup of representors, virtualization options, and optional bridge configurations. The subsections guide users through the steps to effectively implement these software components.

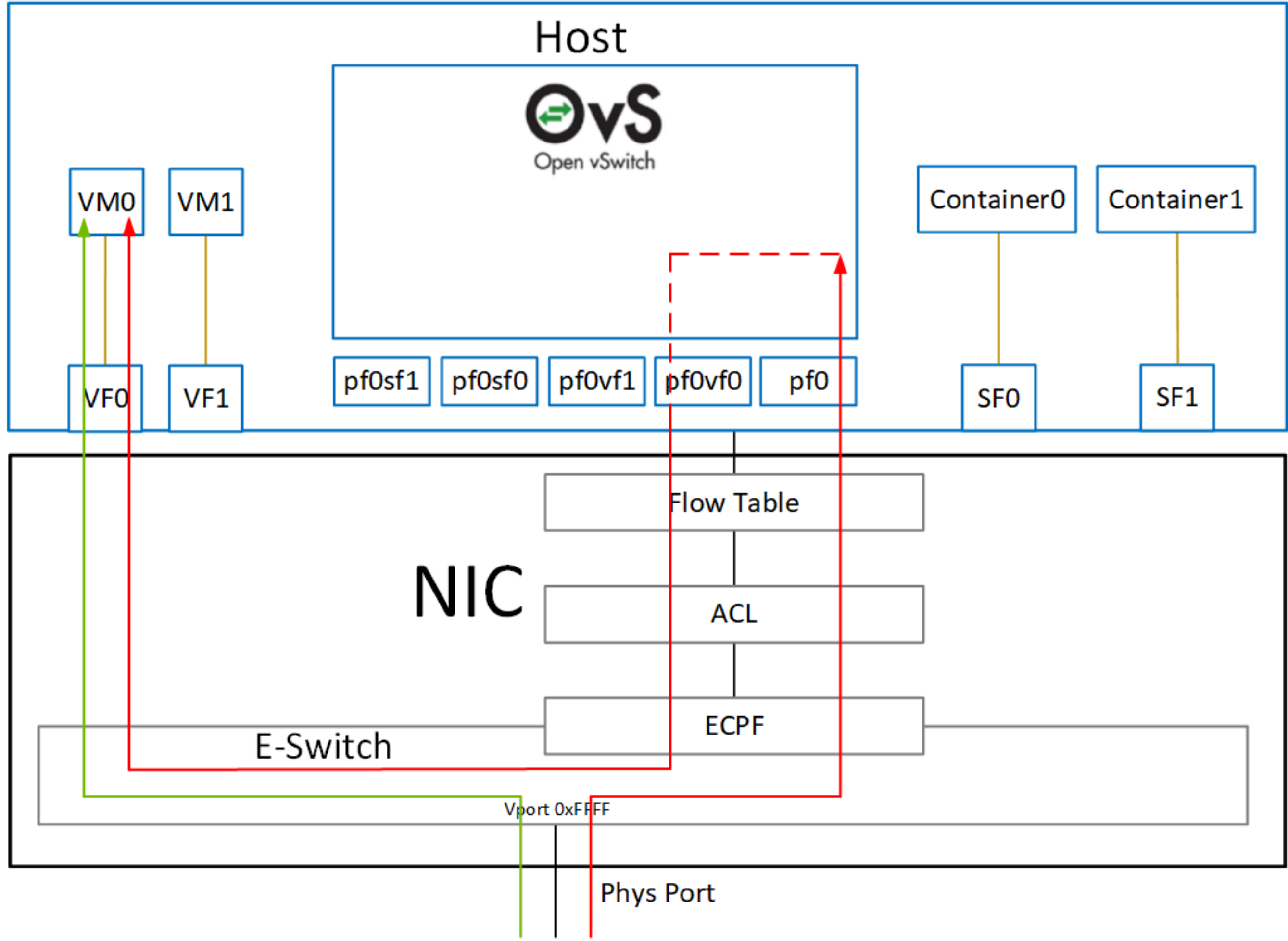
Switch Device Mode

Switchdev mode allows the physical function (PF) to operate as a virtual switch, granting software a means to control associated virtual function (VF) and scalable function (SF) traffic. In this mode, software can intercept incoming/outgoing VF/SF traffic and configure steering rules that offload traffic.

Once this mode is configured, the PF is considered an E-Switch, and additional ports are allocated for each associated VF and SF. The additional ports are referred to as port representors.

ConnectX and BlueField NIC Mode

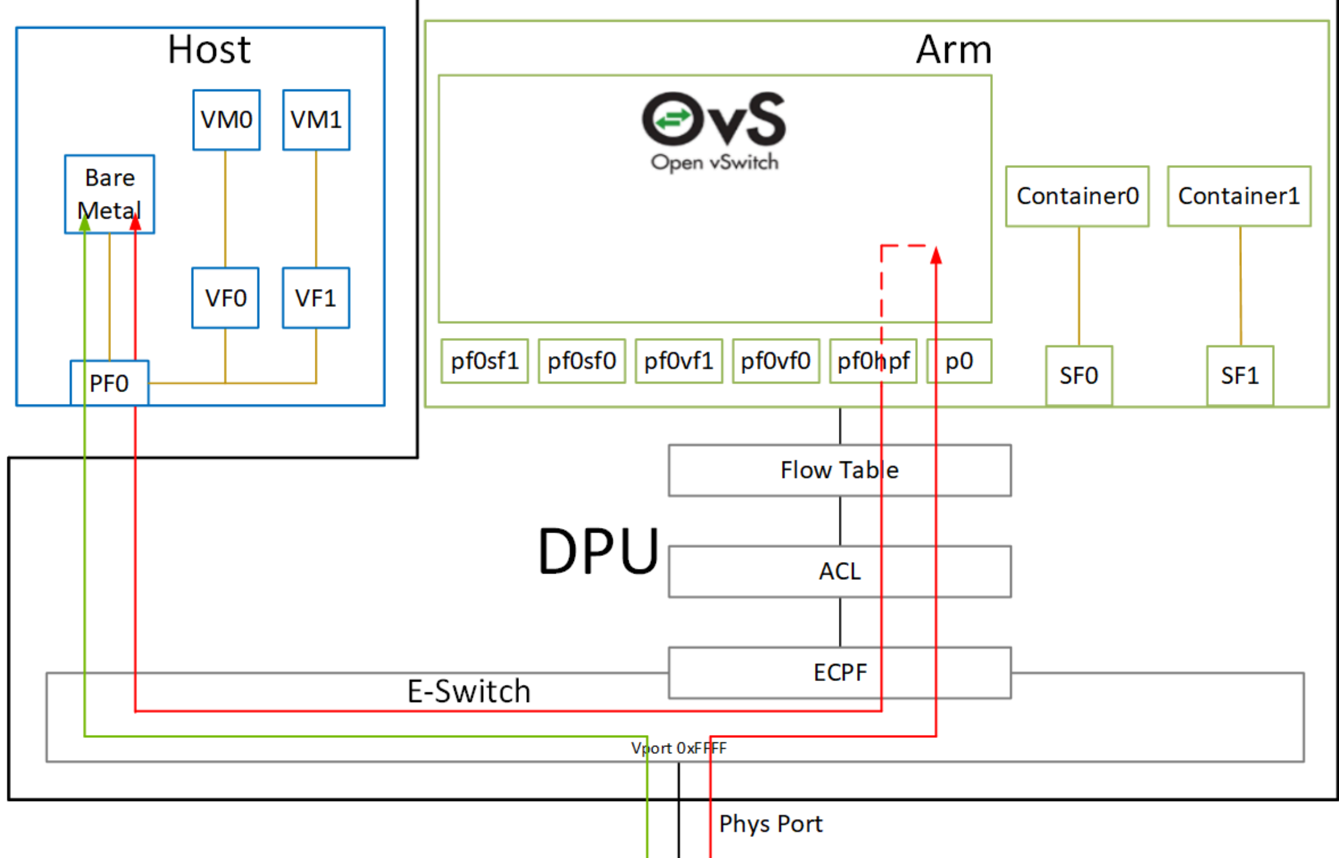
Switchdev mode may be configured for each PF as described in the "Configuring Switchdev Mode" section.



BlueField DPU Mode

When BlueField operates in DPU mode, switchdev mode is configured by default for all PFs on the Arm. That configuration must not be changed. In this mode, the host cannot configure PFs to operate in switchdev mode. Instead, host PFs operate in legacy mode only.

The following diagram shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, the diagram shows a single port model (duplicated for the second port).



The red arrow demonstrates packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch.

E-Switch Port

Once switchdev mode is configured, the PF acts as an e-switch and manages all switch ports. The e-switch can be used to control VF and SF traffic. Software can then use the e-switch through the netdev and RDMA core interfaces.

Netdev Interface

The e-switch has a netdev interface (e.g., `pf0`), allowing it to be used with common Linux networking tools (e.g., `ifconfig`) and OVS. While the interface looks like a regular network port, in reality it is considered the upstream representor and it cannot host an IP server. That is, configuring an IP address to this interface is not useful (e.g., it cannot be pinged).

The netdev interface can be used mainly by OVS and the Linux bridge.

To overcome this limitation, it is possible to create SFs. For more details, see BlueField Scalable Function User Guide.

RDMA Core Interface

The e-switch has an RDMA device instance (e.g., `mlx5_0`) which grants software access to the e-switch capabilities for offloading steering rules and accessing various RDMA capabilities.

The instance behaves like an RDMA core device with the following limitations:

- It cannot host an RDMA connection and cannot be used to connect to the remote RDMA target (e.g., cannot do RDMA send)
- It cannot have an RDMA GID

The RDMA device can mainly be used to configure the e-switch.

To overcome this limitation, it is possible to create SFs. For more details, see BlueField Scalable Function User Guide.

Representor Port

For each network function (i.e., VF/SF) a corresponding representor port is created. The representor port represents the actual port. The network function can then be used to access the network by a VM/container, while the representor can be used to manage switching by virtual switch software (e.g., bare metal host).

Netdev Interface

Each representor port would have a netdev interface (e.g., `pf0vf0`), allowing it to be used with common Linux networking tools (e.g., `ifconfig`) as well as OVS.

While the interface looks like a regular network port, in reality it is a VF/SF representor and it cannot host an IP server. That is, configuring an IP address to this interface is not useful (e.g., it cannot be pinged).

The representor netdev can be used to refer to a specific network function when configuring steering rules using OVS or Linux bridge.

RDMA Core Interface

The representor does not have a corresponding RDMA core instance (e.g., `mlx5_0`). Instead, it is managed by the RDMA core instance of the e-switch manager. Each representor is an RDMA port of the e-switch RDMA device.

Configuring Switchdev Mode

Note

For BlueField in DPU mode, there is no need to follow these steps as the PFs are already configured to switchdev mode by default.

1. Unbind all VFs:

```
# echo 0000:3d:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
# echo 0000:3d:00.3 > /sys/bus/pci/drivers/mlx5_core/unbind
```

Note

VMs with attached VFs must be powered off to be able to unbind the VFs.

2. Change the e-switch mode from legacy to switchdev on the PF device:

```
# devlink dev eswitch set pci/0000:3d:00.0 mode switchdev
```

This creates the VF/SF representor ports in the host OS.

Note

Before changing the mode, make sure that all VFs are unbound.

Info

To return to legacy mode, run:

```
# devlink dev eswitch set pci/0000:3d:00.0 mode
legacy
```

This removes the VF/SF representor ports.

On OSes or kernels that do not support devlink, moving to switchdev mode can be done using sysfs:

```
# echo switchdev > /sys/class/net/pf0/compat/devlink/mode
```

3. At this stage, VF representors have been created. To map a representor to its VF, make sure to obtain the representor's `switchid` and `portname` by running:

```
# ip -d link show eth0
41: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq
state UP mode DEFAULT group default qlen 1000
    link/ether ba:e6:21:37:bc:d4 brd ff:ff:ff:ff:ff:ff
promiscuity 0 addrgenmode eui64 numtxqueues 10 numrxqueues 10
gso_max_size 65536 gso_max_segs 65535 portname pf0vf0 switchid
f4ab580003a1420c
```

Where:

- `switchid` – used to map representor to device, both device PFs have the same `switchid`
- `portname` – used to map representor to PF and VF. Value returned is `pf<X>vf<Y>`, where `X` is the PF number and `Y` is the number of VF.

4. Bind the VFs:

```
echo 0000:3d:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
echo 0000:3d:00.3 > /sys/bus/pci/drivers/mlx5_core/bind
```

DPU Kernel Representors Model

Note

This model is only applicable when the NVIDIA® BlueField® networking platform (DPUs or SuperNIC) is operating in DPU mode.

BlueField uses netdev representors to map each one of the host side physical and virtual functions.

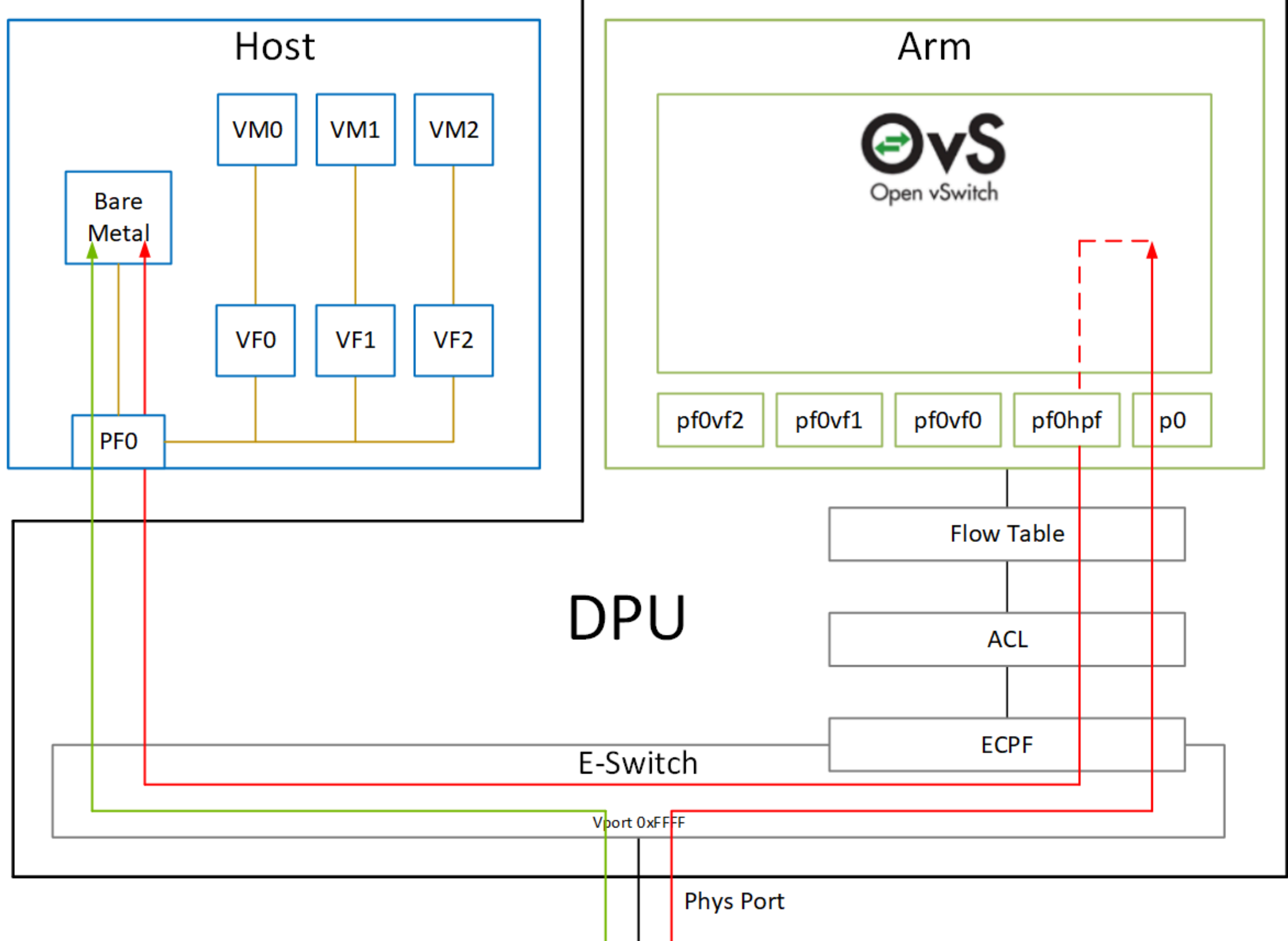
1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the host side.
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

When operating in DPU mode, we see 2 representors for each one of the BlueField's network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: `p<port_number>`
- PF representors: `pf<port_number>hpf`
- VF representors: `pf<port_number>vf<function_number>`

The following diagram shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, a single port model (duplicated for the second port) is shown.



The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

Note

The MTU of host functions (PF/VF) must be smaller than the MTUs of both the uplink and corresponding PF/VF representor. For example, if the host PF MTU is set to 9000, both uplink and PF representor must be set to above 9000.

Virtio Acceleration through Hardware vDPA

Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.

To install QEMU:

1. Clone the sources:

```
git clone https://git.qemu.org/git/qemu.git
cd qemu
git checkout v2.12
```

2. Build QEMU:

```
mkdir bin
cd bin
../configure --target-list=x86_64-softmmu --enable-kvm
make -j24
```

To install DPDK:

1. Clone the sources:

```
git clone git://dpdk.org/dpdk
cd dpdk
git checkout v20.11
```

2. Install dependencies (if needed):

```
yum install cmake gcc libnl3-devel libudev-devel make
pkgconfig valgrind-devel pandoc libibverbs libmlx5 libmnl-
devel -y
```

3. Configure DPDK:

```
export RTE_SDK=$PWD
make config T=x86_64-native-linuxapp-gcc
cd build
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_PMD=\)n/\1y/g' .config
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=\)n/\1y/g' .config
```

4. Build DPDK:

```
make -j
```

5. Build the vDPA application:

```
cd $RTE_SDK/examples/vdpa/
make -j
```

Hardware vDPA Configuration

To configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
```

```
echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-
1048576kB/nr_hugepages
echo <more> > /sys/devices/system/node/node1/hugepages/hugepages-
1048576kB/nr_hugepages
```

To configure a vDPA VirtIO interface in an existing VM's xml file (using `libvirt`):

1. Open the VM's configuration XML for editing:

   ```
   virsh edit <domain name>
   ```

2. Perform the following:

   1. Change the top line to:

      ```
      <domain type='kvm'
      xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
      ```

   2. Assign a memory amount and use 1GB page size for huge pages (size must be the same as that used for the vDPA application), so that the memory configuration looks as follows.

      ```
      <memory unit='KiB'>4194304</memory>
      <currentMemory unit='KiB'>4194304</currentMemory>
      <memoryBacking>
        <hugepages>
          <page size='1048576' unit='KiB'/>
        </hugepages>
      </memoryBacking>
      ```

3. Assign an amount of CPUs for the VM CPU configuration, so that the `vcpu` and `cputune` configuration looks as follows:

```
<vcpu placement='static'>5</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='14'/>
  <vcpupin vcpu='1' cpuset='16'/>
  <vcpupin vcpu='2' cpuset='18'/>
  <vcpupin vcpu='3' cpuset='20'/>
  <vcpupin vcpu='4' cpuset='22'/>
</cputune>
```

4. Set the memory access for the CPUs to be shared, so that the `cpu` configuration looks as follows:

```
<cpu mode='custom' match='exact' check='partial'>
  <model fallback='allow'>Skylake-Server-IBRS</model>
  <numa>
    <cell id='0' cpus='0-4' memory='8388608' unit='KiB'
memAccess='shared'/>
  </numa>
</cpu>
```

5. Set the emulator in use to be the one built in step 2, so that the emulator configuration looks as follows:

```
<emulator><path to qemu executable></emulator>
```

6. Add a virtio interface using QEMU command line argument entries, so that the new interface snippet looks as follows:

```
<qemu:commandline>
  <qemu:arg value='-chardev'/>
  <qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
  <qemu:arg value='-netdev'/>
  <qemu:arg value='vhost-user,chardev=charnet1,queues=16,id=hostnet1'/>
  <qemu:arg value='-device'/>
  <qemu:arg value='virtio-net-
pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:d3
   page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'/>
</qemu:commandline>
```

Note

In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCIe slot of the virtio device can be configured.

Running Hardware vDPA

Note

Hardware vDPA supports switchdev mode only.

1. Create the ASAP[2] environment:
   1. Create the VFs.
   2. Enter switchdev mode.
   3. Set up OVS.

2. Run the vDPA application:

```
cd $RTE_SDK/examples/vdpa/build
./vdpa -w <VF PCI BDF>,class=vdpa --log-level=pmd,info -- -i
```

3. Create a vDPA port via the vDPA application CLI:

```
create /tmp/sock-virtio0 <PCI DEVICE BDF>
```

Note

The vhostuser socket file path must be the one used when configuring the VM.

4. Start the VM:

```
virsh start <domain name>
```

For further information on the vDPA application, visit the Vdpa Sample Application DPDK documentation.

Bridge Offload

Note

Bridge offload is supported switchdev mode only.

Note

Bridge offload is supported from kernel version 5.15 onward.

A Linux bridge is an in-kernel software network switch (based on and implementing a subset of IEEE 802.1D standard) used to connect Ethernet segments together in a protocol-independent manner. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides the ability to offload bridge dataplane unicast packet forwarding and VLAN management to hardware.

Basic Configuration

1. Initialize the ASAP^2 environment:
    1. Create the VFs.
    2. Enter switchdev mode.
2. Create a bridge and add mlx5 representors to bridge:

```
ip link add name bridge0 type bridge
ip link set enp8s0f0_0 master bridge0
```

Configuring VLAN

1. Enable VLAN filtering on the bridge:

```
ip link set bridge0 type bridge vlan_filtering 1
```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

```
bridge vlan add dev enp8s0f0_0 vid 2
```

3. Configure port VLAN tagging (access mode). In this configuration, VLAN header is pushed/popped upon reception/transmission on port.

```
bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
```

VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in section "SR-IOV VF LAG".

To add a bonding net device to bridge:

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, refer to man 8 bridge.

Link Aggregation

Network bonding enables combining two or more network interfaces into a single interface. It increases the network throughput, bandwidth and provides redundancy if one of the interfaces fails.

NVIDIA ® BlueField ® networking platforms (DPUs or SuperNICs) have an option to configure network bonding on the Arm side in a manner transparent to the host. Under such configuration, the host would only see a single PF.

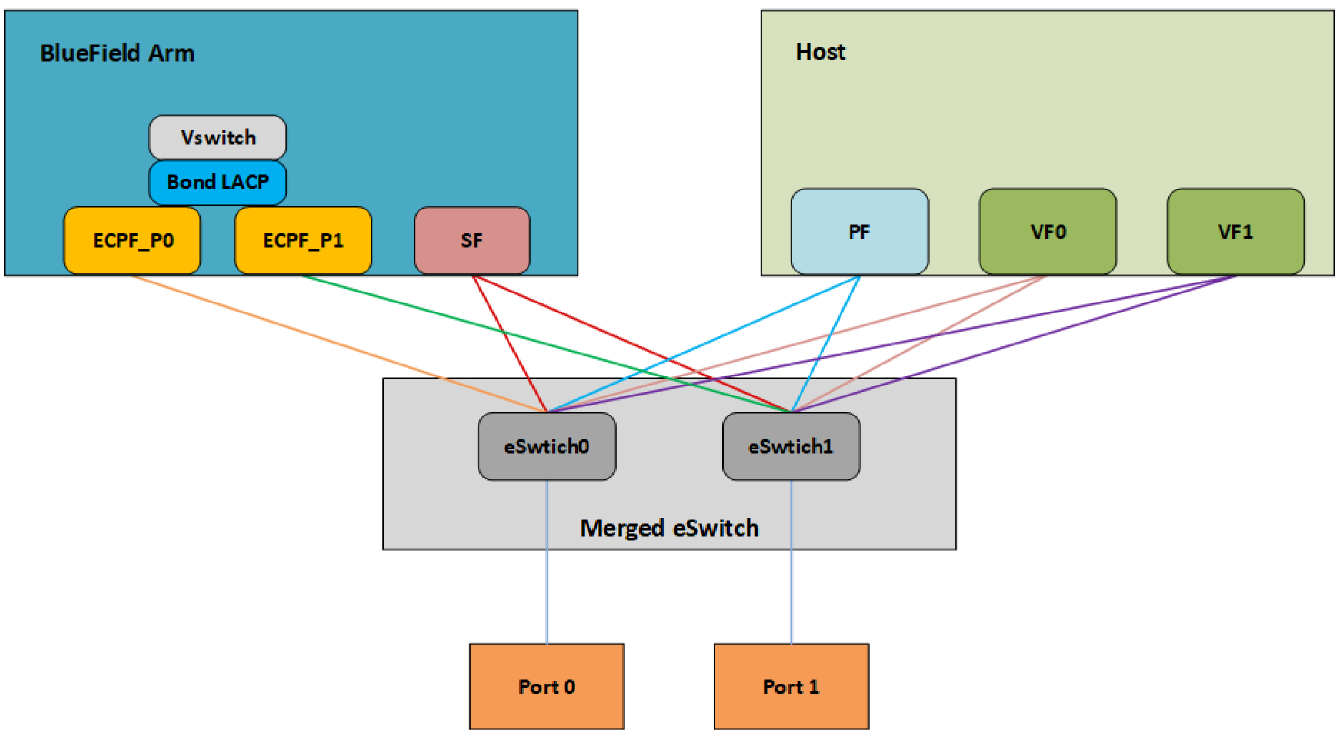
Note

This functionality is supported when BlueField is set in embedded function ownership mode for both ports.

Note

While LAG is being configured (starting with step 2 under section "LAG Configuration"), traffic cannot pass through the physical ports.

The following diagram describes this configuration:



LAG Modes

Two LAG modes are supported on BlueField:

- Queue Affinity mode
- Hash mode

Queue Affinity Mode

In this mode, packets are distributed according to the QPs.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=0
```

   Example device name: `mt41686_pciconf0`.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:

```
LAG_HASH_MODE="no"
```

3. Perform BlueField system reboot for the `mlxconfig` settings to take effect.

Hash Mode

In this mode, packets are distributed to ports according to the hash on packet headers.

Note

For this mode, prerequisite steps 3 and 4 are not required.

1. To enable this mode, run:

```
$ mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=1
```

Example device name: `mt41686_pciconf0`.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:

```
LAG_HASH_MODE="yes"
```

3. Perform BlueField system reboot for the `mlxconfig` settings to take effect.

Prerequisites

1. Set the LAG mode to work with.
2. (Optional) Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True
NUM_OF_PF=1
```

Example device name: `mt41686_pciconf0`.

Note

Perform BlueField system reboot for the `mlxconfig` settings to take effect.

3. Delete any installed Scalable Functions (SFs) on the Arm side.

4. Stop the driver on the host side. Run:

```
$ systemctl stop openibd
```

5. The uplink interfaces (`p0` and `p1`) on the Arm side must be disconnected from any OVS bridge.

LAG Configuration

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4
xmit_hash_policy layer3+4
```

> **Note**
>
> While LAG is being configured (starting with the next step), traffic cannot pass through the physical ports.

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p1 down
$ ip link set p0 master bond0
$ ip link set p1 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p1 up
$ ip link set bond0 up
```

The following is an example of LAG configuration in Ubuntu:

```
# cat  /etc/netplan/70-bf-lag.yaml

network:
  renderer: networkd
  bonds:
    bond0:
      optional: true
      dhcp4: no
      interfaces: [p0, p1]
      parameters:
        mode: 802.3ad
        mii-monitor-interval: 100
  ethernets:
    p0:
      dhcp4: no
      optional: true
    p1:
      dhcp4: no
      optional: true
  version: 2
```

As a result, only the first PF of the BlueFields would be available to the host side for networking and SR-IOV.

Warning

When in shared RQ mode (enabled by default), the uplink interfaces (`p0` and `p1`) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.

For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces `p0` and `p1` should not be added). The PF representor for the first port (`pf0hpf`) of the LAG must be added to the OVS bridge. The PF representor for the second port (`pf1hpf`) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```
ovs-vsctl add-br bf-lag
ovs-vsctl add-port bf-lag bond0
ovs-vsctl add-port bf-lag pf0hpf
```

Warning

Trying to change bonding configuration in Queue Affinity mode (including bringing the subordinated interface up/down) while the host driver is loaded would cause FW syndrome and failure of the operation. Make sure to unload the host driver before altering BlueField bonding configuration to avoid this.

Note

When performing driver reload (`openibd restart`) or reboot, it is required to remove bond configuration and to reapply the

configurations after the driver is fully up. Refer to steps 1-4 of "Removing LAG Configuration".

Removing LAG Configuration

1. If Queue Affinity mode LAG is configured (i.e., `LAG_RESOURCE_ALLOCATION=0`):
    1. Delete any installed Scalable Functions (SFs) on the Arm side.
    2. Stop driver (openibd) on the host side. Run:

    ```
    systemctl stop openibd
    ```

2. Delete the LAG OVS bridge on the Arm side. Run:

    ```
    ovs-vsctl del-br bf-lag
    ```

    This allows for later restoration of OVS configuration for non-LAG networking.

3. Stop OVS service. Run:

    ```
    systemctl stop openvswitch-switch.service
    ```

4. Run:

    ```
    ip link set bond0 down
    modprobe -rv bonding
    ```

    As a result, both of the BlueField's network interfaces would be available to the host side for networking and SR-IOV.

5. For the host to be able to use BlueField's ports, make sure to attach the ECPF and host representor in an OVS bridge on the Arm side. Refer to "Virtual Switch on BlueField" for instructions on how to perform this.

6. Revert from `HIDE_PORT2_PF`, on the Arm side. Run:

```
mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=False
NUM_OF_PF=2
```

7. Restore default LAG settings in BlueField's firmware. Run:

```
mlxconfig -d /dev/mst/<device-name> s
LAG_RESOURCE_ALLOCATION=DEVICE_DEFAULT
```

8. Delete the following line from `/etc/mellanox/mlnx-bf.conf` on the Arm side:

```
LAG_HASH_MODE=...
```

9. Perform BlueField system reboot for the `mlxconfig` settings to take effect.

LAG on Multi-host

Only LAG hash mode is supported with BlueField multi-host.

LAG Multi-host Prerequisites

1. Enable LAG hash mode.

2. Hide the second PF on the host. Run:

```
$ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True
```

```
NUM_OF_PF=1
```

3. Make sure NVME emulation is disabled:

```
$ mlxconfig -d /dev/mst/<device-name> s
NVME_EMULATION_ENABLE=0
```

Example device name: `mt41686_pciconf0`.

4. The uplink interfaces (`p0` and `p4`) on the Arm side, representing port0 and port1, must be disconnected from any OVS bridge. As a result, only the first PF of BlueField would be available to the host side for networking and SR-IOV.

LAG Configuration on Multi-host

1. Create the bond interface. Run:

```
$ ip link add bond0 type bond
$ ip link set bond0 down
$ ip link set bond0 type bond miimon 100 mode 4
xmit_hash_policy layer3+4
```

2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p4 down
$ ip link set p0 master bond0
$ ip link set p4 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p4 up
$ ip link set bond0 up
```

4. For OVS configuration, the bond interface is the one that must be added to the OVS bridge (interfaces p0 and p4 should not be added). The PF representor, `pf0hpf`, must be added to the OVS bridge with the bond interface. The rest of the uplink representors must be added to another OVS bridge along with their PF representors. Consider the following examples:

```
ovs-vsctl add-br br-lag
ovs-vsctl add-port br-lag bond0
ovs-vsctl add-port br-lag pf0hpf
ovs-vsctl add-br br1
ovs-vsctl add-port br1 p1
ovs-vsctl add-port br1 pf1hpf
ovs-vsctl add-br br2
ovs-vsctl add-port br2 p2
ovs-vsctl add-port br2 pf2hpf
ovs-vsctl add-br br3
ovs-vsctl add-port br3 p3
ovs-vsctl add-port br3 pf3hpf
```

Note

When performing driver reload (`openibd restart`) or reboot, you must remove bond configuration from NetworkManager, and to reapply the configurations after the driver is fully up.

Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".

Controlling Host PF and VF Parameters

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) allow control over some of the networking parameters of the PFs and VFs running on the host side.

Setting Host PF and VF Default MAC Address

From the Arm, users may configure the MAC address of the physical function in the host. After sending the command, users must reload the NVIDIA driver in the host to see the newly configured MAC address. The MAC address goes back to the default value in the firmware after system reboot.

Example:

```
$ echo "c4:8a:07:a5:29:59" > /sys/class/net/p0/smart_nic/pf/mac
$ echo "c4:8a:07:a5:29:61" > /sys/class/net/p0/smart_nic/vf0/mac
```

Setting Host PF and VF Link State

vPort state can be configured to Up, Down, or Follow. For example:

```
$ echo "Follow" > /sys/class/net/p0/smart_nic/pf/vport_state
```

Querying Configuration

To query the current configuration, run:

```
$ cat /sys/class/net/p0/smart_nic/pf/config
MAC         : e4:8b:01:a5:79:5e
MaxTxRate   : 0
State       : Follow
```

Zero signifies that the rate limit is unlimited.

Disabling Host Networking PFs

It is possible to not expose networking functions to the host for users interested in using storage or virtio functions only. When this feature is enabled, the host PF representors (i.e. `pf0hpf` and `pf1hpf`) will not be seen on the Arm.

- Without a PF on the host, it is not possible to enable SR-IOV, so VF representors will not be seen on the Arm either
- Without PFs on the host, there can be no SFs on it

To disable host networking PFs, run:

```
mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=0
```

To reactivate host networking PFs:

- For single-port BlueFields, run:

  ```
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=1
  ```

- For dual-port BlueFields, run:

  ```
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=2
  ```

Note

When there are no networking functions exposed on the host, the reactivation command must be run from the Arm.

Note

Perform BlueField system reboot for the `mlxconfig` settings to take effect.

Configuring Uplink MTU

To configure the port MTU while operating in DPU mode, users must restrict the external host port ownership by issuing the following command on the BlueField:

```
mlxprivhost -d /dev/mst/<pciconf0 device> r --disable_port_owner
```

Server cold reboot is required for this restriction to take effect.

Once the host is restricted, the port MTU is configured by changing the MTU of the uplink representor (`p0` or `p1`).

DPDK on BlueField

Please refer to "NVIDIA BlueField Board Support Package" in the DPDK documentation.

DOCA with OpenSSL

This guide provides instructions on using DOCA SHA for OpenSSL implementations.

Introduction

The `doca_sha_offload_engine` is an OpenSSL dynamic engine with the ability of offloading SHA calculation. It can offload the OpenSSL one-shot SHA-1, SHA-256, and SHA-512. It supports synchronous mode and asynchronous mode by leveraging the OpenSSL `async_jobs` library. For more information on the `async_jobs` library, please refer to official OpenSSL documentation.

This engine is based on the `doca_sha` library and the OpenSSL dynamic engine interface API. For more information on the OpenSSL dynamic engine, please refer to official OpenSSL documentation.

This engine can be called by an OpenSSL application through the OpenSSL high-level algorithm call interface, `EVP_Digest`. For more information on the `EVP_Digest`, please refer to official OpenSSL documentation.

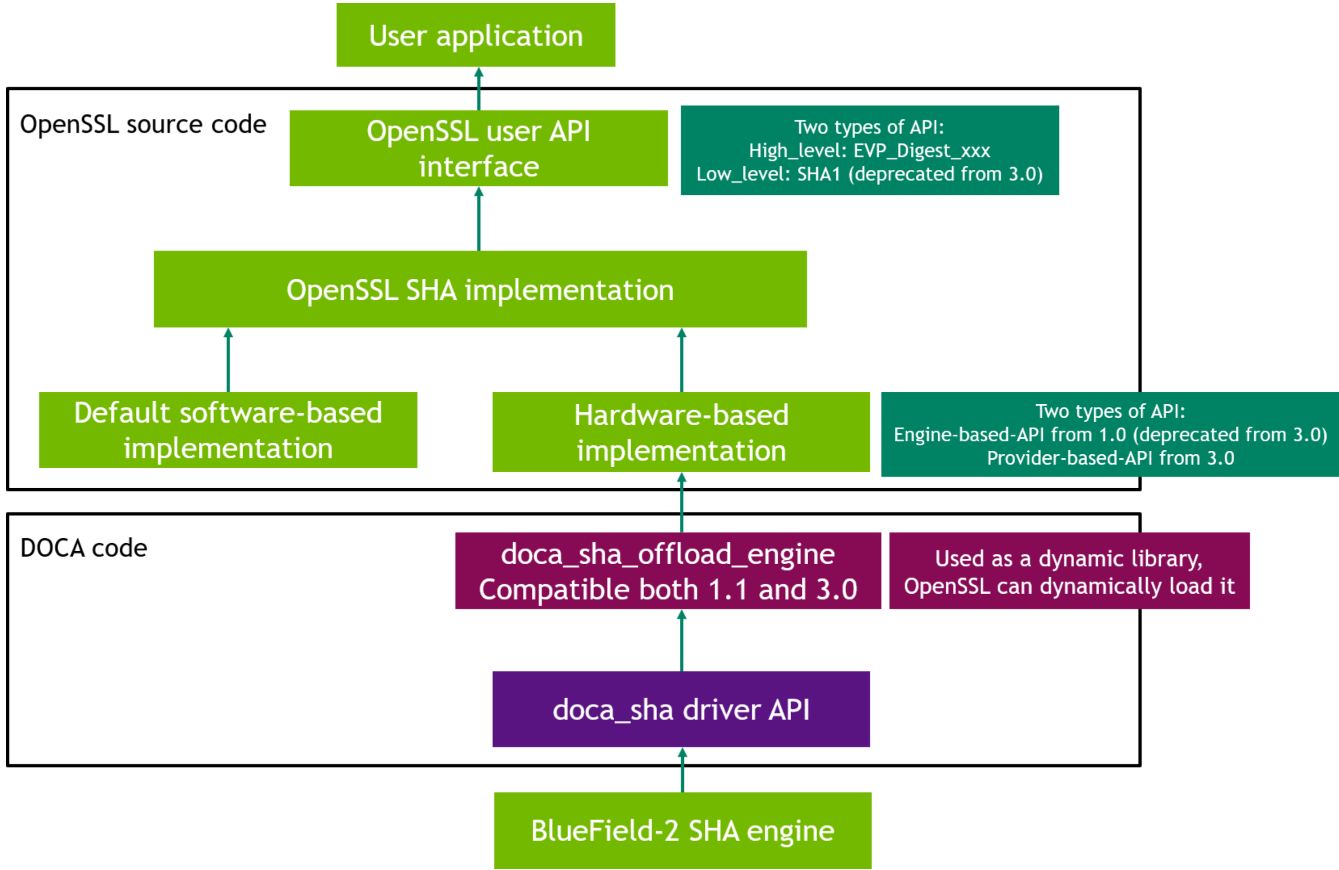
Prerequisites

- Hardware-based `doca_sha` engine which can be verified by calling `doca_sha_get_hardware_supported()`
- Installed OpenSSL version ⩾ 1.1.1

Architecture

The following diagram shows the software hierarchy of `doca_sha_offload_engine` and its location in the whole DOCA repository.

From the perspective of OpenSSL, this engine is an instantiation of the OpenSSL dynamic engine interface API by leveraging the `doca_sha` library.



Capabilities and Limitations

- Only one-shot OpenSSL SHA is supported
- The maximum message length ⩽ 2GB, the same as `doca_sha` library

OpenSSL Command Line Verification

Verify that the engine can be loaded:

```
$ openssl engine dynamic -pre NO_VCHECK:1 -pre
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_
-pre LOAD -vvv -t -c
(dynamic) Dynamic engine loading support
[Success]:
SO_PATH:${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_
[Success]: LOAD
```

```
Loaded: (doca_sha_offload_engine) Openssl SHA offloading engine
based on doca_sha
 [SHA1, SHA256, SHA512]
     [ available ]
     set_pci_addr: set the pci address of the doca_sha_engine
          (input flags): STRING
```

- For SHA-1:

  ```
  $ echo "hello world" | openssl dgst -sha1 -engine
  {DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
  -engine_impl
  ```

- For SHA-256:

  ```
  $ echo "hello world" | openssl dgst -sha256 -engine
  {DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
  -engine_impl
  ```

- For SHA-512:

  ```
  $ echo "hello world" | openssl dgst -sha512 -engine
  {DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
  -engine_impl
  ```

OpenSSL Throughput Test

`openssl-speed` is the OpenSSL throughput benchmark tool. For more information, consult official OpenSSL documentation. `doca_sha_offload_engine` throughput can also be measured using `openssl-speed`.

- SHA-1, each job 10000 bytes, using engine:

```
$ openssl speed -evp sha1 -bytes 10000 -elapsed --engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
```

- SHA-256, each job 10000 bytes, using engine, `async_jobs=256`:

```
$ openssl speed -evp sha256 -bytes 10000 -elapsed --engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
-async_jobs 256
```

- SHA-512, each job 10000 bytes, using engine, `async_jobs=256`, `threads=8`:

```
$ openssl speed -evp sha512 -bytes 10000 -elapsed --engine
{DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_
-async_jobs 256 -multi 8
```

Using DOCA SHA Offload Engine in OpenSSL Application

More information on the dynamic engine usage can be found in the official OpenSSL documentation.

1. To load the `doca_sha_offload_engine` (optionally, set engine PCIe address):

```
ENGINE *e;
const char *doca_engine_path =
"${DOCA_DIR}/infrastructure/doca_sha_offload_engine/libdoca_sha_offload_engine.so";
const char *default_doca_pci_addr = "03:00.0";
ENGINE_load_dynamic();
e = ENGINE_by_id(doca_engine_path);
```

```
ENGINE_ctrl_cmd_string(e, "set_pci_addr", doca_engine_pci_addr,
0);
ENGINE_init(e);
ENGINE_set_default_digests(e);
```

2. To perform SHA calculation by calling the OpenSSL high-level function EVP_XXX:

```
const EVP_MD *evp_md = EVP_sha1();
EVP_MD_CTX *mdctx = EVP_MD_CTX_create();
EVP_DigestInit_ex(mdctx, evp_md, e);
EVP_DigestUpdate(mdctx, msg, msg_len);
EVP_DigestFinal_ex(mdctx, digest, digest_len);
EVP_MD_CTX_destroy(mdctx);
```

3. To unload the engine:

```
ENGINE_unregister_digests(e);
ENGINE_finish(e);
ENGINE_free(e);
```

BlueField Scalable Function User Guide

This document provides an overview and configuration of scalable functions (sub-functions, or SFs) for NVIDIA® BlueField® DPU.

Introduction

Scalable functions (SFs), or sub-functions, are very similar to virtual functions (VFs) which are part of a Single Root I/O Virtualization (SR-IOV) solution. I/O virtualization is one of the key features used in data centers today. It improves the performance of enterprise servers by giving virtual machines direct access to hardware I/O devices. The SR-IOV specification allows one PCI Express (PCIe) device to present itself to the host as multiple distinct "virtual" devices. This is done with a new PCIe capability structure added to a traditional PCIe function (i.e., a physical function or PF).

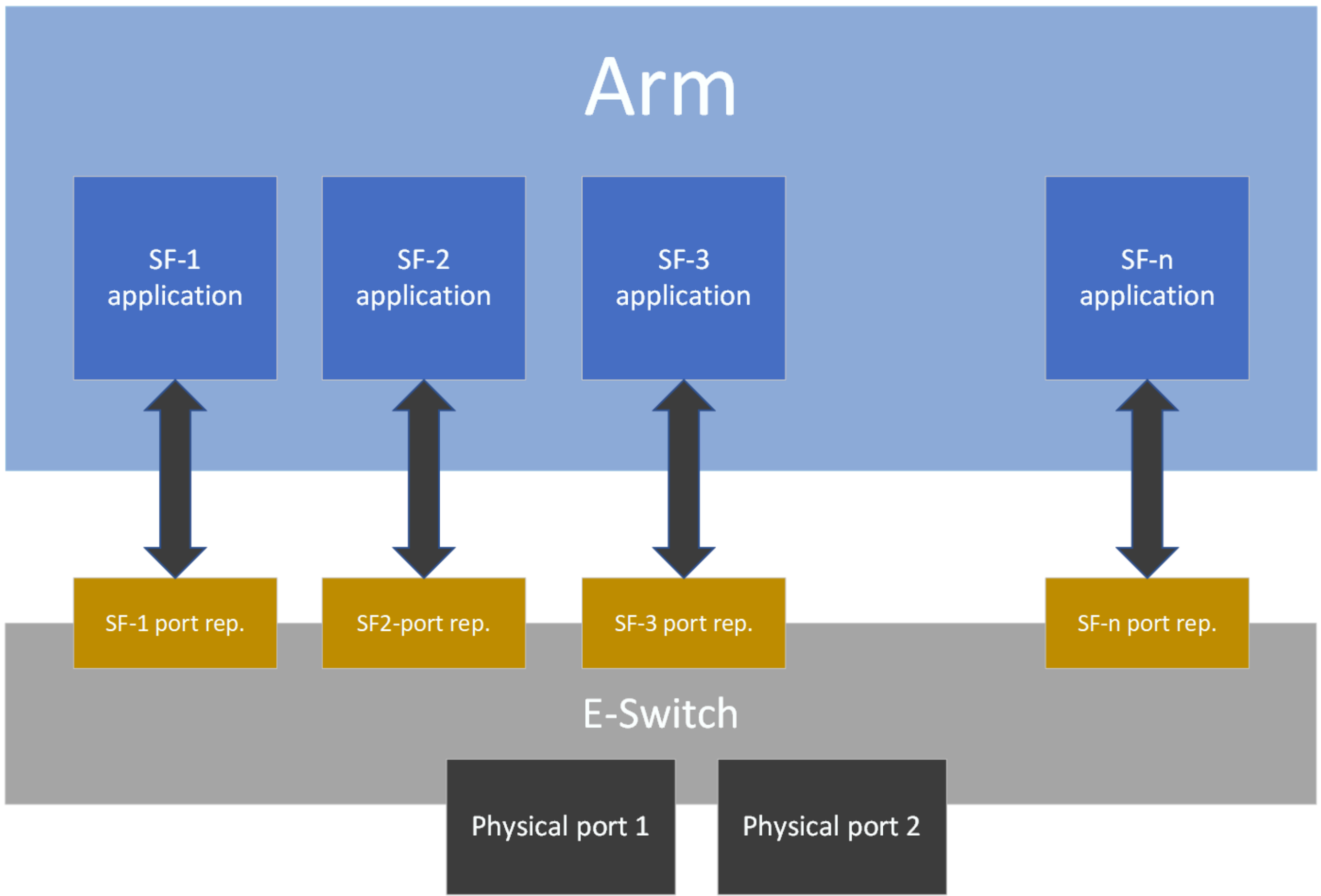
The PF provides control over the creation and allocation of new VFs. VFs share the device's underlying hardware and PCIe. A key feature of the SR-IOV specification is that VFs are very lightweight so that many of them can be implemented in a single device.

To utilize the capabilities of VF in the BlueField, SFs are used. SFs allow support for a larger number of functions than VFs, and more importantly, they allow running multiple services concurrently on the DPU.

An SF is a lightweight function which has a parent PCIe function on which it is deployed. The SF, therefore, has access to the capabilities and resources of its parent PCIe function and has its own function capabilities and its own resources. This means that an SF would also have its own dedicated queues (i.e., txq, rxq).

SFs co-exist with PCIe SR-IOV virtual functions (on the host) but also do not require enabling PCIe SR-IOV.

SFs support E-Switch representation offload like existing PF and VF representors. An SF shares PCIe-level resources with other SFs and/or with its parent PCIe function.



Prerequisites

Refer to the DOCA Installation Guide for Linux for details on how to install BlueField related software.

- Make sure your firmware version is 20.30.1004 or higher
- Enable s upport for Linux kernel mlx5 SFs by setting the following Kconfig flags:
    - `MLX5_ESWITCH`
    - `MLX5_SF`
- To enable SF support on the device, change the PCIe address for each port:

```
$ mlxconfig -d 0000:03:00.0 s PF_BAR2_ENABLE=0
PER_PF_NUM_SF=1 PF_TOTAL_SF=236
 PF_SF_BAR_SIZE=10
```

```
PF_BAR2_ENABLE: if this config is set, then all PFs and ECPFs
have the same number of SFs. This should be off (deprecated).
If set. PF_TOTAL_SF and PF_SF_BAR_SIZE won't work.
PER_PF_NUM_SF: If this config is set, each PF and ECPF
configure/control its own number of SFs.
THE ABOVE TWO CONFIGS AFFECS BOTH BF AND HOST, TREAT WITH
CARE!
Also, only one of them can be set. It is INVALID to set them
both

PF_TOTAL_SF: maximum number of SFs we wish to configure for
the given PF/ECPF.
PF_SF_BAR_SIZE: size of each SF at the BAR2. The size is in
powers of 2 in KB.
For example: PF_SF_BAR_SIZE=10 means each SF is taking 1MB of
the BAR.
             PF_TOTAL_SF=14 means this PCI function can
create up to 14 SFs.
             In total: FW will allocate 14MB of BAR2.
```

Note

Perform a BlueField system-level reset for the `mlxconfig` settings to take effect.

SF Configuration

To use an SF, a 3-step setup sequence must be followed first:

1. Create.
2. Configure.

3. Deploy.

These steps can be performed using `mlxdevm` tool.

Info

When working on top of an upstream-based kernel, on which the `mlxdevm` tool is unavailable, please refer to the Upstream Guide on Scalable Functions for instructions on using the `devlink` tool which should be used instead.

Configuration Using mlxdevm Tool

1. Create the SF.

   SFs are managed using the `mlxdevm` tool supplied with iproute2 package. The tool is found at `/opt/mellanox/iproute2/sbin/mlxdevm`.

   An SF is created using the `mlxdevm` tool. The SF is created by adding a port of `pcisf` flavor.

   To create an SF port representor, run:

   ```
   /opt/mellanox/iproute2/sbin/mlxdevm port add
   pci/<pci_address> flavour pcisf pfnum <corresponding pfnum>
   sfnum <sfnum>
   ```

> **Note**
>
> Each SF must have a unique number (`<sfnum>`).

For example:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add pci/0000:03:00.0
flavour pcisf pfnum 0 sfnum 4
```

Output example:

```
pci/0000:30:00.0/229409: type eth netdev eth0 flavour pcisf
controller 0 pfnum 0 sfnum 4
   function:
        hw_addr 00:00:00:00:00:00 state inactive opstate
detached roce true max_uc_macs 128 trust off
```

The number 229409 is required to complete the following two steps (i.e., configuration and deployment).

`pci/0000:03:00.0/229409` is called the SF index.

`pci/<pci_address>/<sf_index>` can be replaced with `<representor_name>`. For example:

```
pci/0000:03:00.0/229409 = en3f0pf0sf4
```

To see information about the created SF such as its MAC address, trust mode, or state (active/inactive), run the following command:

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4 eth0
flavor pcisf controller 0 pfnum 0 sfnum 4
    function:
      hw_addr 00:00:00:00:00:00 state inactive opstate
detached roce true max_uc_macs 128 trust off
```

Note

SF number ≥1 000 is reserved for the virtio-net controller.

Note

When an SF is added on the external controller (e.g., BlueField) users must supply the controller number. In a single host BlueField case, there is only one controller starting with controller number 1. The following is an example of adding an SF for PF0 of external controller 1:

```
$ mlxdevm port add pci/0000:03:00.0 flavour
pcisf pfnum 0 sfnum 88 controller 1
pci/0000:03:00.0/32768: type eth netdev eth6
flavour pcisf controller 1 pfnum 0 sfnum 88
splittable false
  function:
```

```
    hw_addr 00:00:00:00:00:00 state inactive
opstate detached
```

2. Configure the SF.

   A subfunction representor (SF port representor) is created but it is not deployed yet. Users should configure the hardware address (e.g., MAC address), set trust mode to on, and activate the SF before deploying it.

   The following steps can be executed as separate commands (at any order) or combined as one:

   - To configure the hardware address, run:

     ```
     /opt/mellanox/iproute2/sbin/mlxdevm port function set
     pci/<pci_address>/<sf_index>  hw_addr <MAC address>
     ```

   - To set the trust mode to on, run:

     ```
     /opt/mellanox/iproute2/sbin/mlxdevm port function set
     pci/<pci_address>/<sf_index> trust on
     ```

     Info

     A trusted function has additional privileges (e.g., the ability to update steering database).

   - To activate the created SF, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set
pci/<pci_address>/<sf_index> state active
```

Alternatively, to configure the MAC address, set trust mode on, and set the state as active, run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set
pci/<pci_address>/<sf_index>  hw_addr <mac_address> trust on
state active
```

For example:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set
pci/0000:03:00.0/229409 hw_addr 00:00:00:00:04:0 trust on state
active
```

Note

The SF capabilities above must be set before deploying the SF.

3. Deploy the SF.

To unbind the SF from the default config driver and bind the actual SF driver, run:

```
echo mlx5_core.sf.<next_serial> >
/sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
```

```
echo mlx5_core.sf.<next_serial>  >
/sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

For example:

```
echo mlx5_core.sf.4  >
/sys/bus/auxiliary/drivers/mlx5_core.sf_cfg/unbind
echo mlx5_core.sf.4  >
/sys/bus/auxiliary/drivers/mlx5_core.sf/bind
```

Note

`<next_serial>` is a number produced by the firmware when creating the SF (this is the gvmi number of the SF). `mlxdevm` tool when creating the SF. To obtain it, refer to the useful commands provided below.

Note

An application interested in using the SF netdevice and RDMA device must monitor the RDMA and netdevices either through udev monitor or poll the sysfs hierarchy of the SF's auxiliary device.

Useful commands:

- To see the available sub-functions, run:

```
$ devlink dev show
```

For example, if you run the command before creating, configuring, and deploying the SF (using the steps detailed earlier), the output would appear as follows:

```
pci/0000:03:00.0
pci/0000:03:00.1
auxiliary/mlx5_core.sf.2
auxiliary/mlx5_core.sf.3
```

After creating, configuring, and deploying the SF, the output would be:

```
pci/0000:03:00.0
pci/0000:03:00.1
auxiliary/mlx5_core.sf.2
auxiliary/mlx5_core.sf.3
auxiliary/mlx5_core.sf.4
```

Note that the `<next_serial>` number is 4 for the created SF.

- To see the `sfnum` of each sub-function, run:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.
<next_serial>/sfnum
```

For example:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
```

Example output:

```
cat /sys/bus/auxiliary/devices/mlx5_core.sf.4/sfnum
4
```

- To remove an SF, you must first make its state inactive and only then remove the SF representor.

  To make the SF's state inactive, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set
  pci/<pci_address>/<sf_index> state inactive
  ```

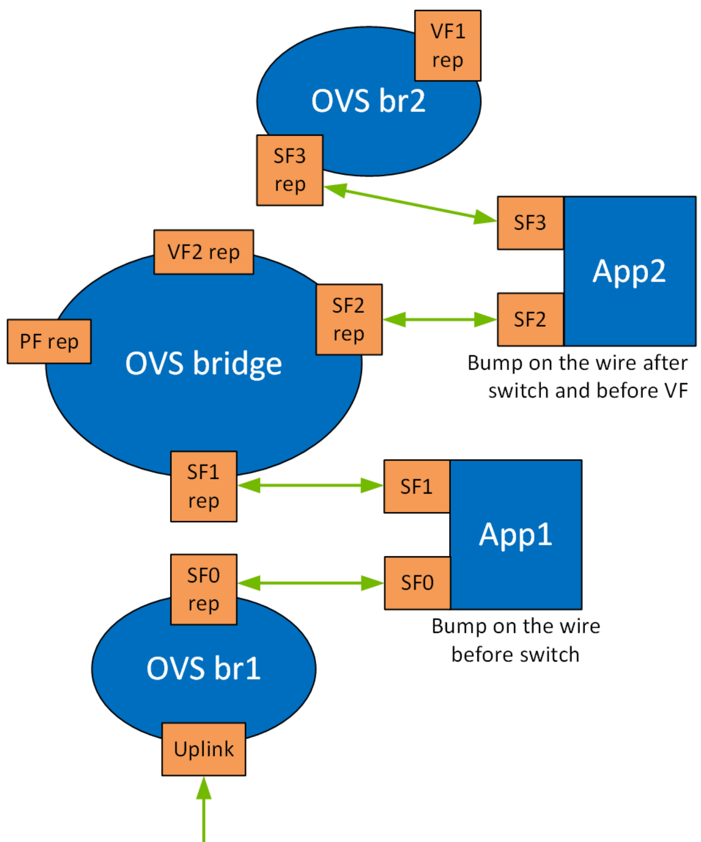
  To delete the SF port representor, run:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port del
  pci/<pci_address>/<sf_index>
  ```

  For example:

  ```
  /opt/mellanox/iproute2/sbin/mlxdevm port function set
  pci/0000:03:00.0/229409 state inactive
  /opt/mellanox/iproute2/sbin/mlxdevm port del
  pci/0000:03:00.0/229409
  ```

4. Use the SF.



Running the application on the DPU requires OVS configuration. By creating SFs, an SF representor for the OVS is also created and named `en3f0pf*sf*`. Therefore, each representor needs to be connected to the correct OVS bridge.

Note

Two SFs related to the same PCIe are necessary for the configuration in the illustration.

The following example configures 2 SFs and adds their representors to the OVS.

1. Create, configure, and deploy the SFs. Run:

```
/opt/mellanox/iproute2/sbin/mlxdevm port add
pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 4
```

```
/opt/mellanox/iproute2/sbin/mlxdevm port add
pci/0000:03:00.0 flavour pcisf pfnum 0 sfnum 5
```

Using the command `mlxdevm port show`, you can see the SF indices of the created SFs.

```
/opt/mellanox/iproute2/sbin/mlxdevm port show
```

Output example:

```
pci/0000:30:00.0/229409: type eth netdev en3f0pf0sf4
flavour pcisf controller 0 pfnum 0 sfnum 4
   function:
        hw_addr 00:00:00:00:00:00 state inactive opstate
detached roce true max_uc_macs 128 trust off
pci/0000:30:00.0/229410: type eth netdev en3f0pf0sf5
flavour pcisf controller 0 pfnum 0 sfnum 5
   function:
        hw_addr 00:00:00:00:00:00 state inactive opstate
detached roce true max_uc_macs 128 trust off
```

2. Configure the MAC address, set trust mode on, and activate the created SFs:

```
/opt/mellanox/iproute2/sbin/mlxdevm port function set
pci/0000:03:00.0/229409 hw_addr 02:25:f2:8d:a2:4c trust
on state active
/opt/mellanox/iproute2/sbin/mlxdevm port function set
pci/0000:03:00.0/229410 hw_addr 02:25:f2:8d:a2:5c trust
on state active
```

Using `ifconfig`, you may see that there are 2 added network interfaces: `en3f0pf0sf4` and `en3f0pf0sf5` for the two respective SF port representors.

3. Delete existing OVS bridges (optional).

   For example, run the following command to delete an OVS bridge called `ovsbr1`:

   ```
   ovs-vsctl del-br ovsbr1
   ```

4. Create two bridges `sf_bridge1` and `sf_bridge2` and configure them as follows:

   ```
   ovs-vsctl add-br sf_bridge1
   ovs-vsctl add-br sf_bridge2
   ovs-vsctl add-port sf_bridge1 p0
   ovs-vsctl add-port sf_bridge2 pf0hpf
   ```

5. Add the port representors to the OVS bridges:

   ```
   ovs-vsctl add-port sf_bridge1 en3f0pf0sf4
   ovs-vsctl add-port sf_bridge2 en3f0pf0sf5
   ```

   The OVS bridges after adding the SF representors:

   ```
   Bridge sf_bridge1
        Port p0
            Interface p0
        Port sf_bridge1
            Interface sf_bridge1
   ```

```
                type: internal
        Port en3f0pf0sf4
            Interface en3f0pf0sf4
    Bridge sf_bridge2
        Port sf_bridge2
            Interface sf_bridge2
                type: internal
        Port en3f0pf0sf5
            Interface en3f0pf0sf5
        Port pf0hpf
            Interface pf0hpf
    ovs_version: "2.14.1"
```

Note

The interface might be down by default. Remember to `ifconfing` the interface to "up" status.

Note

When deleting the SF port representor, you must also de-attach it from the bridge it is connected to using the command `ovs-vsctl port-del en3f0pf0sf*`. Otherwise, the port representor will still be connected to the bridge but would not be recognizable.

To run the application, use the following command to initialize the SFs during runtime:

```
*Executable_binary* -a auxiliary:mlx5_core.sf.* -a
auxiliary:mlx5_core.sf.*
```

For example:

```
doca_<app_name> -a auxiliary:mlx5_core.sf.4 -a
auxiliary:mlx5_core.sf.5 -- [application_flags]
```

DOCA TLS Offload Guide

This guide provides an overview and configuration steps of TLS hardware offloading via kernel-TLS, using hardware capabilities of NVIDIA® BlueField® DPU.

Introduction

Transport layer security (TLS) is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP (VoIP), but its use in securing HTTPS remains the most publicly visible.

The TLS protocol aims primarily to provide cryptography, including privacy (confidentiality), integrity, and authenticity using certificates, between two or more communicating computer applications. It runs in the application layer and is itself composed of two layers: the TLS record and the TLS handshake protocols.

TLS works over TCP and consists of 3 phases:

1. Handshake – establishment of a connection
2. Application – sending and receiving encrypted packets
3. Termination – connection termination

TLS Handshake

In the handshake phase, the client and server decide on which cipher suites they will use, and exchange keys and certificates according to the following flow:

1. Client hello, provides the server at a minimum with the following:
    - A key exchange algorithm, to determine how symmetric keys are exchanged
    - An authentication or digital signature algorithm, which dictates how server authentication and client authentication (if required) are implemented
    - A bulk encryption cipher, which is used to encrypt the data

    - A hash/MAC (message authentication code) function, which determines how data integrity checks are carried out
    - The version of the protocol it understands
    - The cipher suites it is capable of working with
    - A unique random number, which is important to guard against replay attacks
2. Server hello:
    - Selects a cipher suite
    - Generates its own random number
    - Assigns a session ID to the TLS connection
    - Sends enough information to complete a key exchange—most often, this means sending a certificate including an RSA public key
3. Client:
    - Responsible for completing the key exchange using the information the server provided

At this point, the connection is secured, both sides have agreed on an encryption algorithm, a MAC algorithm, and respective keys.

kTLS

The Linux kernel provides TLS offload infrastructure. kTLS (kernel TLS) offloads TLS handling from the user-space to the kernel-space.

kTLS has 3 modes of operation:

- SW – all operation is handled in kernel (i.e., handshake, encryption, decryption)
- HW-offload (the focus of this guide) – handshake and error handling are performed in software. Packets are encrypted/decrypted in hardware. In this case, there is an additional offload from the kernel to the hardware.

- HW-record – all operations are handled by the hardware (driver and firmware) including the handshake. It also handles its own TCP session. This option is currently not supported.

Note

It is important to understand that Rx (receiving) and Tx (sending) can have two separate modes. For example, Rx can be dealt in SW mode but Tx in HW-offload mode (i.e., the hardware will only encrypt but not decrypt).

HW-offloading kTLS

In general, the TLS HW-offload performs best and provides optimal value on longer lived sessions, with relatively large packets. Scaling in terms of concurrent connections and connections per second is use-case dependent (e.g., the amount of active concurrent connections from the overall open concurrent connections is material).

It is necessary to learn the following terms before proceeding:

- The transport interface send (TIS) object is responsible for performing all transport-related operations of the transmit side. Messages from Send Queues (SQs) get segmented and transmitted by the TIS including all transport required implications. For example, in the case of a large send offload, the TIS is responsible for the segmentation. The NVIDIA® ConnectX® hardware uses a TIS object to save and access the TLS crypto information and state of an offloaded Tx kTLS connection.

- The transport interface receive (TIR) object is responsible for performing all transport-related operations on the receive side. TIR performs the packet processing and reassembly and is also responsible for demultiplexing the packets into different receive queues (RQs).

- Both TIS and TIR hold the data encryption key (DEK).

kTLS Offload Flow in High Level

> **Note**
>
> The following flow does not include resync and errors.

1. Establishes a TLS connection with remote host (server or client) by handling a TLS handshake by kernel on current host.
2. Initializes the following state for each connection, Rx and Tx:
    - Crypto secrets (e.g., public key)
    - Crypto processing state
    - Record metadata (e.g., record sequence number, offset)
    - Expected TCP sequence number

Tx flow:

1. Packets belonging to device offloaded sockets arrive to the kernel and it does not encrypt them.
2. Kernel performs record framing and marks the packet with a connection identifier.
3. Kernel sends packets to the device driver for offloading.
4. Device checks that the sequence number matches the state in the TIS and performs encryption and authentication.

Rx flow:

1. When the connection is created, a HW steering rule is added to steer packets to their respective TIR.
2. Device receives the packet then validates and checks that sequence number of TCP matches the state in the TIR.
3. Performs decryption and authentication, and indicates in the CQE (completion queue entry).

4. Kernel understands that the packet is already decrypted so it does not decrypt it itself and passes it on to the user-space.

Resync and Error Handling

When the sequence number does not match expectations or if any other error occurs, the hardware gives control back to the SW which handles the problem.

See more about kTLS modes, resync, and error handling in the Linux Kernel documentation.

Prerequisites

All commands in this section should be performed on host (not on BlueField) unless stated otherwise.

Checking Hardware Support for Crypto Acceleration

To check if the BlueField or ConnectX have crypto acceleration, run the following command from host:

```
host> mst start # turn on mst driver
host> flint -d <device under /dev/mst/ directory> dc | grep Crypto
```

The output should include `Crypto Enabled`. For example:

```
host> flint -d  /dev/mst/mt41686_pciconf0 dc | grep Crypto
....
;;Description = NVIDIA BlueField-2 E-Series Eng. sample DPU;
200GbE single-port QSFP56; PCIe Gen4 x16; Secure Boot Disabled;
Crypto Enabled; 16GB on-board DDR; 1GbE OOB management
```

```
....
```

Kernel Requirements

- Operating system must be either:
    - FreeBSD 13.0+.
    - A Linux distribution built on Linux kernel version 5.3 or later for Tx support and version 5.9 or later for Rx support. We recommend using the latest version when possible for the best available optimizations.

      Note

      TIS Pool optimization is added to Linux kernel version 6.0. Instead of creating TIS per new connection, unused TIS from previous connection, will be recycled. This will improve Tx connection rate. No further installations required beyond installing the kernel itself.

- Check the current kernel version on the host. Run:

```
host> uname -r
```

- The kernel must be configured to support TLS by setting the options `TLS_DEVICE` and `MLX5_TLS` to `y`. To check if TLS is configured, run:

```
host> cat /boot/config-$(uname -r) | grep TLS
```

Example output:

```
host> cat /boot/config-5.4.0-121-generic | grep TLS
...
CONFIG_TLS_DEVICE=y
CONFIG_MLX5_TLS=y
...
```

If the current kernel does not support one of the options, you can change the configur ations and recompile, or build a new kernel .

Note

Follow the build instructions provided with the kernel provider.

Schematic flow for building a Linux kernel:

1. Enter the Linux kernel directory downloaded (usually in `/usr/src/`):

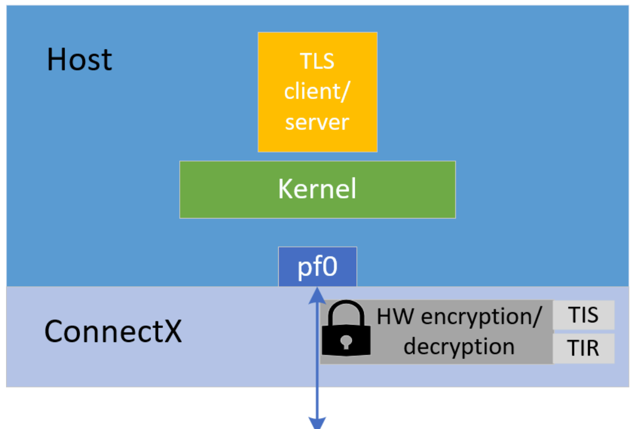
    ```
    host> make menuconfig # Set TLS_DEVICE=y and MLX5_TLS=y in options. Setting location in the menu can be found by pressing '/' and typing 'setting'.
    host> make -j <num-of-cores> && make -j <num-of-cores> modules_install && make -j <num of cores> install
    ```

2. Update the grub to the new configured kernel then reboot.

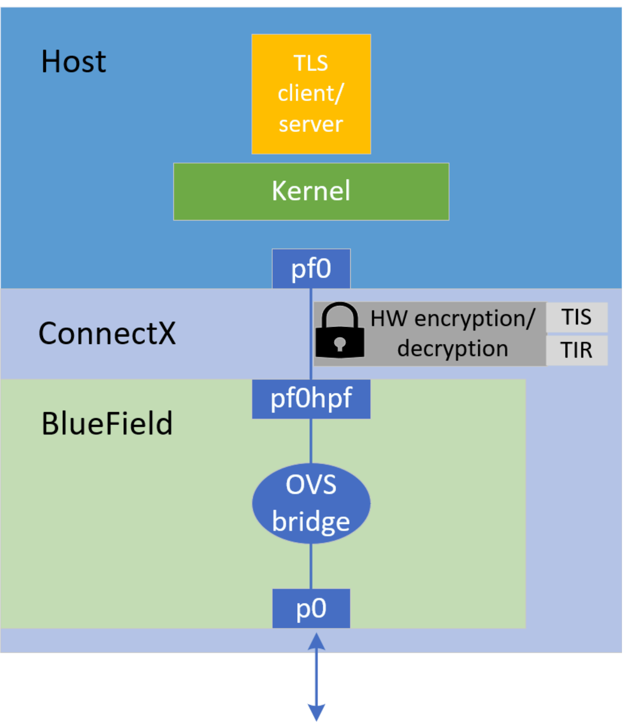
Configurations and Useful Commands

TLS Setup

Host + ConnectX Setup



Host + BlueField Setup



Finding NVIDIA Interfaces

```
host> mst start                    # if mst driver is not loaded.
host> mst status -v
```

NVIDIA's netdev interfaces are found be under the `NET` column.

For example:

```
host> mst status -v
....
DEVICE_TYPE             MST                            PCI
RDMA            NET                         NUMA
BlueField2(rev:0)       /dev/mst/mt41686_pciconf0.1    b1:00.1
mlx5_1          net-ens5f1                  1
```

```
BlueField2(rev:0)       /dev/mst/mt41686_pciconf0     b1:00.0
mlx5_0          net-ens5f0                 1
```

In this example, the interfaces `ens5f1` and `ens5f0` are NVIDIA's netdev interfaces.

Configuring TLS Offload

- To check if the offload option is on or off, run:

  ```
  host> ethtool -k $iface | grep tls
  ```

  Example output:

  ```
  tls-hw-tx-offload: on
  tls-hw-rx-offload: off
  tls-hw-record: off [fixed]
  ```

  Note

  `tls-hw-record` is not required for the device as kTLS does not support "HW Record" mode.

- To turn Tx offload on or off:

  ```
  host> ethtool -K $iface tls-hw-tx-offload <on | off>
  ```

- To turn Rx offload on or off:

```
host> ethtool -K $iface tls-hw-rx-offload <on | off>
```

Configuring OVS Bridge on BlueField

When the host is connected to a BlueField device, an OVS bridge must be configured on the BlueField so traffic passes bidirectionally from host to uplink. If no OVS bridge is configured, the host is isolated from the network (see diagram above).

Note

On BlueField image version 3.7.0 or higher the default OVS configuration can be used without additional modifications.

To configure the OVS bridge on BlueField, run the following commands on BlueField:

```
dpu> for br in $(ovs-vsctl list-br); do ovs-vsctl del-br $br; done # erasing existing bridges
dpu> ovs-vsctl add-br ovs-br0 && ovs-vsctl add-port ovs-br0 p0 && ovs-vsctl add-port ovs-br0 pf0hpf
dpu> ovs-vsctl add-br ovs-br1 && ovs-vsctl add-port ovs-br1 p1 && ovs-vsctl add-port ovs-br1 pf1hpf
dpu> ovs-vsctl set Open_vSwitch . other_config:hw-offload=true && systemctl restart openvswitch-switch
```

Where `p0`/`p1` are the uplink interfaces and `pf0hpf`/`pf1hpf` are the interfaces facing the host.

Common Use Cases

OpenSSL

OpenSSL is an all-around cryptography library that offers open-source application of the TLS protocol. It is the main library for using kTLS and other applications since Nginx depends on it as their base library.

Note

The kTLS and HW offloading do not depend on OpenSSL. Any program that can implement a TLS stack can be run instead. However, because of the vast use of OpenSSL, this guide addresses installation recommendations.

kTLS is supported only in OpenSSL version 3.2.0 or higher, and only on the supported kernel versions. The supported OpenSSL version is available for download from distro packages, or it can be downloaded and compiled from the OpenSSL GitHub.

Warning

Many modules depend on OpenSSL. Changing the default version may cause problems. Adding `--prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl` in the `./Configure` command below may prevent the built OpenSSL from becoming the default one used by the system. Make sure the directory of the OpenSSL you build manually is not located in any paths listed in the PATH environment variable.

1. Check the version of the default OpenSSL:

```
host> openssl version
```

2. Follow OpenSSL installation instructions from OpenSSL's supplied guides. During the configuration process, make sure to set the `enable-ktls` option before building it by running it from within the OpenSSL directory (works in version 3.2.0 and higher). For example:

```
host> ./Configure linux-$(uname -p) enable-ktls --prefix=/var/tmp/ssl --openssldir=/var/tmp/ssl # Add "threads" as well for multithread support
```

3. Check if kTLS is enabled in OpenSSL by running the following command from within the OpenSSL directory, and check whether `ktls` is listed under `Enabled features`:

```
host> perl configdata.pm --dump | less
```

If OpenSSL has been downloaded manually, the OpenSSL executable would be located in the `/<openssl-dir>/apps/` directory. For example, checking the version from within OpenSSL directory is done using the command `./apps/openssl version`.

Note

Installing a new OpenSSL requires recompiling user tools that were configured over OpenSSL (e.g., Nginx).

Note

In OpenSSL's master source code, there is a feature "Support for kTLS Zero-Copy sendfile() on Linux" (Zero-Copy commit). If the Zero-Copy option is set, `SSL_sendfile()` uses the Zero-Copy TX mode which means that the data itself is not copied from the user space to Kernel space. This gives a performance boost when used with kTLS hardware offload. Be aware that invalid TLS records may be transmitted if the file is changed while being sent.

Nginx

Nginx is a free and open-source software web server that can also be used as a reverse proxy , load balancer , mail proxy and HTTP cache . Nginx can be configured to depend on OpenSSL library and therefore Nginx could have the great advantages of TLS HW-offload on ConnectX-6 Dx, ConnectX-7 or the DPU.

Prerequisites

Refer to the OpenSSL section for setting OpenSSL.

Configuration

1. Install dependencies. For Ubuntu distribution, for example:

```
host> apt install libpcre3 libpcre3-dev
```

2. Clone Nginx's repository and enter directory:

```
host> git clone https://github.com/nginx/nginx.git && cd
nginx
```

3. Configure Nginx components to support kTLS:

```
host> ./auto/configure --with-
openssl=/<insert_path_to_openssl_directory> --with-debug --
with-http_ssl_module --with-openssl-opt="enable-ktls -
DOPENSSL_LINUX_TLS -g3"
```

4. Build Nginx:

```
host> make -j <num of cores> && sudo make -j <num-of-cores>
install
```

> **Note**
>
> If `make` fails with a `deprecated openssl functions` error, remove `-Werror` for `CFLAGS` in objs/Makefile and try again.

5. Add the following lines to the end of the `/usr/local/nginx/conf/nginx.conf` file (before the last closing bracket):

```
server {
    listen 443 ssl default_server reuseport;
    server_name localhost;
    root /tmp/nginx/docs/html/;

    include /etc/nginx/default.d/*.conf;
    ssl_certificate /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key /usr/local/nginx/conf/key.pem;
    ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256;
    ssl_conf_command Options KTLS;
```

```
    ssl_protocols TLSv1.2;

 location / {
    index index.html;
 }

 error_page 404 /404.html;
    location = /40x.html {
 }

 error_page 500 502 503 504 /50x.html;
    location = /50x.html {
 }
}
```

6. Notice that the key and certificate of the Nginx server should be located in `/usr/local/nginx/conf/`. Therefore, after creating a key and certificate (as mentioned in section "Adding Certificate and Key") they should be copied to the aforementioned directory:

```
host> cp key.pem /usr/local/nginx/conf/ && cp cert.pem
/usr/local/nginx/conf/
```

7. To run Nginx:

```
host> cd nginx && objs/nginx
```

This command starts Nginx Server in the background.

Stopping Nginx

```
host> pkill nginx
```

Wrk – Client

A simple client for requesting Nginx's server is "wrk". It can be installed by running the following:

```
host> git clone https://github.com/wg/wrk.git && cd wrk/ && make
-j <num-of-cores>
```

Using Wrk

The following is an example of using the wrk client to request the page `index.html` from the Nginx server in address `4.4.4.4` (run within wrk's directory):

```
host> taskset -c 0 ./wrk -t1 -c10 -d30s
https://4.4.4.4:443/index.html
```

Note

Testing the kTLS offload (with or without hardware offload) is in the same manner as mentioned in section "Testing kTLS". TBD

Testing Offload via OpenSSL

This chapter demonstrates how to test the kTLS hardware offload.

Note

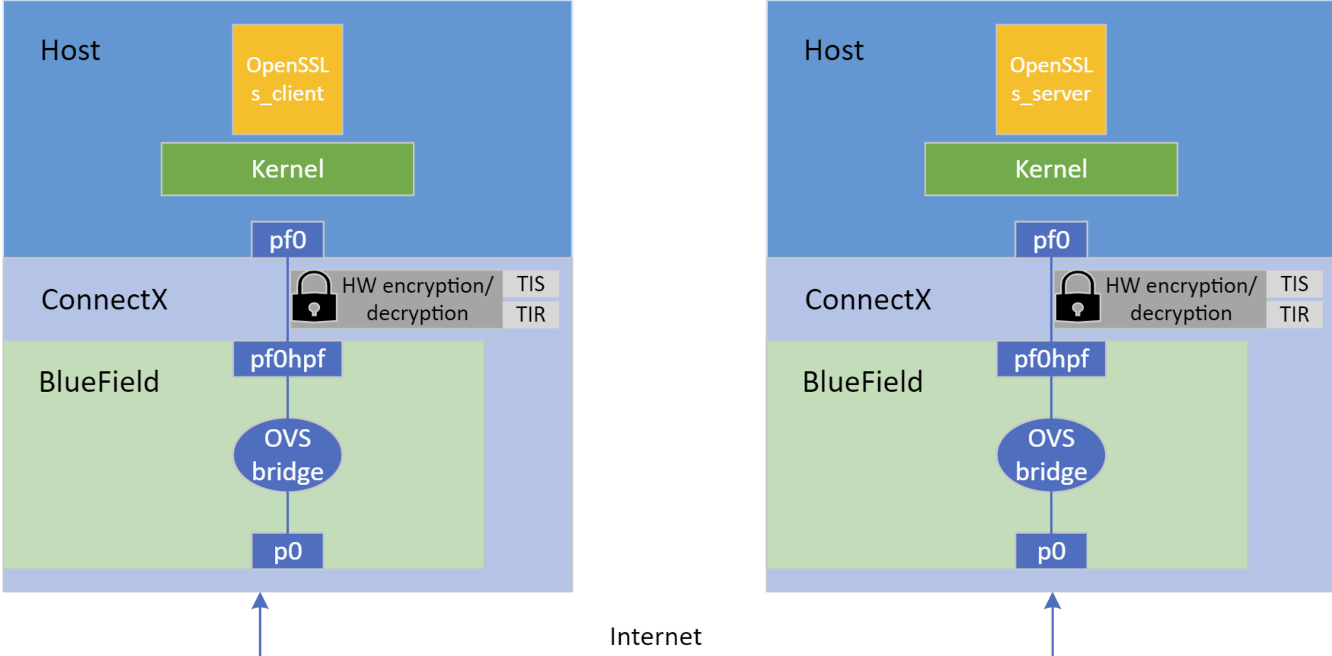
Make sure to refer to section "OpenSSL" before proceeding.

TLS Testing Setup

For testing purposes, a server and a client are required. The testing section only tests a single setup of a host and BlueField-2 or a host ConnectX which will participate either as a server or as a client. Setting a back-to-back setup of the same kind and installing the same OpenSSL version can help avoid misconfigurations. Nevertheless, it is required to have the same OpenSSL version on both the client and server.

Make sure the desired kTLS is configured as detailed in section "Configuring TLS Offload". To test hardware offload, make sure `tls-hw-tx-offload` and/or `tls-hw-rx-offload` are on. To test kTLS software mode, make sure to turn them off.

In addition, make sure both hosts (server and client) can communicate bidirectionally through ConnectX or BlueField. One can set the interface that supports the offload (on the host) with an IP, in same subnet. Make sure that when using BlueField, an OVS bridge is set on BlueField as shown in "Configuring OVS Bridge on BlueField".



Adding Certificate and Key

The server side should create a certificate and key. The client can also use a certificate, but it is not necessary for this test case. Run the following command in the installed OpenSSL directory and fill in all the requested details:

```
host> openssl req -x509 -newkey rsa:2048 -keyout key.pem -out
cert.pem -days 365 -nodes
```

The following files are created:

- `key.pem` – private-key file used to generate the CSR and, later, to secure and verify connections using the certificate
- `cert.pem` – certificate signing request (CSR) file used to order your SSL certificate and, later, to encrypt messages that only its corresponding private key can decrypt

> **Note**

The server side should be run before client side so that client's request are answered by server.

Running Server Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_server -key key.pem -cert cert.pem -tls1_2 -
cipher ECDHE-RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

Note

Notice the `-ktls` flag.

Note

Refer to official OpenSSL documentation on `s_server` for more information.

In this example, the key and certificate are provided, the cipher suite and TLS version are configured, and the server listens to port 443 and is instructed to use kTLS.

Running Client Side

The following example works on OpenSSL version 3.1.0:

```
host> openssl s_client -connect 4.4.4.4:443 -tls1_2 -ktls
```

Where 4.4.4.4 is the IP of the port/interface under test (e.g., `enp177s0f0np0`).

Note

Refer to official OpenSSL documentation on `s_client` for more information.

Testing kTLS

After the connection is established (handshake is done), a prompt will open and the user, both on the client and server side, can send a message to other side in a chat-like manner. Messages should appear on the other side once they are received.

The following example checks kTLS hardware offload on the tested setup by tracking Rx and Tx TLS on device counters:

```
host> ethtool -S $iface | grep -i 'tx_tls_encrypted\|rx_tls_decrypted' # ($iface is the interface that offloads)
```

To check kTLS over kernel counters:

```
host> cat /proc/net/tls_stat
```

Output example:

Note

The comments are not part of the output and are added as explanation.

```
host> cat /proc/net/tls_stat
TlsCurrTxSw                         0                    # Current Tx connections opened in SW mode
TlsCurrRxSw                         0                    # Current Rx connections opened in SW mode
TlsCurrTxDevice                     0                    # Current Tx connections opened in HW-offload mode
TlsCurrRxDevice                     0                    # Current Rx connections opened in HW-offload mode
TlsTxSw                             2323828              # Accumulated number of Tx connections opened in SW mode
TlsRxSw                             1                    # Accumulated number of Rx connections opened in SW mode
TlsTxDevice                         12203652             # Accumulated number of Tx connections opened in HW-offload mode
TlsRxDevice                         0                    # Accumulated number of Rx connections opened in HW-offload mode
TlsDecryptError                     0                    # Failed record decryption (e.g., due to incorrect authentication tag)
TlsRxDeviceResync                   0                    # Rx resyncs sent to HW's handling cryptography
TlsDecryptRetry                     0                    # All Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
TlsRxNoPadViolation                 0                    # Data Rx records re-decrypted due to TLS_RX_EXPECT_NO_PAD misprediction
```

Note

More information about the kernel counters can be found in the Statistics section of the Kernel TLS documentation.

Optimizations over kTLS

XLIO

The NVIDIA accelerated IO (XLIO) software library boosts the performance of TCP/IP applications based on Nginx (e.g., CDN, DoH) and storage solutions as part of SPDK. XLIO is a user-space software library that exposes standard socket APIs **with kernel-bypass** architecture, enabling a hardware-based direct copy between an application's user-space memory and the network interface. In particular, XLIO can boost the performance of applications that use the kTLS hardware offload as OpenSSL and Nginx. Read more about XLIO in the NVIDIA XLIO Documentation and XLIO TLS HW-offload over kTLS in the TLS HW Offload section.

Note

Even though XLIO is a kernel-bypass library, the kernel must support kTLS for the bypass to work properly.

Performance Tuning Options

TLS offload performance is related to how fast data can be pumped though the offload engine. In the case of user space applications, certain system configurations can be tuned to optimize its performance.

The following are items that can be tuned for optimal performance, mainly focusing on dedicating the server's work to the NUMA, or non-uniform memory access, cores:

> **Note**
>
> Non-uniform memory access (NUMA) cores are cores with a dedicated memory for each of them, granting cores fast access to their own memory and slower access to others'. This architecture is best for scenarios when it is not necessary to share memory between cores.

1. Add NUMA cores of the NIC to the `isolcpus` kernel boot arguments for each server so that the kernel scheduler does not interrupt the core's running user thread. The following are examples of adding commands:
    1. Identify the NIC NUMA node (see NUMA column):

       ```
       host> mst status -v
       DEVICE_TYPE          MST                           PCI
       RDMA        NET                     NUMA
       ConnectX6DX(rev:0)   /dev/mst/mt4125_pciconf0 41:00.0
       mlx5_0      net-enp65s0f0np0    1
       ```

    2. Identify the cores of the NIC NUMA node using the NUMA node number acquired from the previous output:

       ```
       host> lscpu | grep "NUMA node1"
       NUMA node1 CPU(s): 1,3,5,7,9,11,13,15,17,19,21,23
       ```

    3. Add the NIC NUMA cores to a grub file (e.g., `/etc/default/grub`) by adding the line
       `GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=<NUMA-cores-from-previous-output>"`
       . For example:

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,3,5,7,9,11,13,15,17
```

4. Update grub:

```
host> sudo update-grub
```

5. Reboot and check that the configuration has been applied:

```
host> cat /proc/cmdline
BOOT_IMAGE=/vmlinuz-5.10.12 root=UUID=1879326c-711f-
4f95-a974-d732af14ef04 ro department=general
user_notifier=dovd osi_string None BOOTIF=01-90-b1-1c-
14-02-44 quiet splash
isolcpus=1,3,5,7,9,11,13,15,17,19,21,23
```

2. Disable `irqbalance` service:

Note

Interrupt request, or IRQ, determines what hardware interrupts arrive to each core.

```
host> service irqbalance stop
```

3. Run `set_irq_affinity.sh` to redistribute IRQs to various cores.

> **Note**
>
> The script is within MLNX_OFED's sources:
>
> 1. You can find it in MLNX_OFED downloads.
> 2. Under "Download" select the correct version and download the "SOURCES" `.tgz` file.
> 3. Extract the `.tgz`.
> 4. Under SOURCES, extract the `mlnx_tools`.
>
> You should find both files `set_irq_affinity.sh` and its helper file `common_irq_affinity.sh` under the `sbin` directory.

```
host> ./set_irq_affinity.sh
<ConnectX_or_BlueField_network_interface>
```

4. Set the interface RSS to the number of cores to use:

```
host> ethtool -X <ConnectX_or_BlueField_network_interface>
equal <number_of_isolcpus_cores>
```

5. Set the interface queues for number of cores to use:

```
host> ethtool -L <ConnectX_or_BlueField_network_interface>
combined <number_of_isolcpus_cores>
```

6. Pin the application with `taskset` to the `isolcpus` cores used. For example:

```
host> taskset -c 1,3,5,7,9,11,13,15,17,19,21,23 openssl
s_server -key key.pem -cert cert.pem -tls1_2 -cipher ECDHE-
RSA-AES128-GCM-SHA256 -accept 443 -ktls
```

Additional Reading

- Linux kernel TLS documentation
- Linux kernel TLS offload documentation
- Autonomous NIC offloads research paper

DOCA Troubleshooting

DOCA troubleshooting topics are available in the *NVIDIA BlueField Platform Software Troubleshooting Guide*.

NVIDIA BlueField Reset and Reboot Procedures

BlueField System Reboot

This section describes the necessary operations to load new NIC firmware, following NVIDIA® BlueField® NIC firmware update. This procedure deprecates the need for full server power cycle.

The following steps are executed in the BlueField OS:

1. Issue a query command to ascertain whether BlueField system reboot is supported by your environment:

   ```
   mlxfwreset -d 03:00.0 q
   ```

   If the output includes the following lines, proceed to step 2:

   ```
   3: Driver restart and PCI reset
       -Supported (default)
   ...
   1: Driver is the owner
   -Supported (default)
   ```

Note

If it says `Not Supported` instead, then proceed to the instructions under section "BlueField System-level Reset".

2. Issue a BlueField system reboot:

```
mlxfwreset -d 03:00.0 -y -l 3 --sync 1 r
```

BlueField System-level Reset

This section describes the way to perform system-level reset (SLR) which is necessary for firmware configuration changes to take effect.

- SLR for BlueField running in DPU mode
- SLR for BlueField running in NIC mode
- SLR for BlueField running in DPU mode on hosts with separate power control (special use case)

System-level Reset for BlueField in DPU Mode

The following is the high-level flow of the procedure:

1. Graceful shutdown of BlueField Arm cores.
2. Query BlueField state to affirm shutdown reached.

Info

In systems with multiple BlueField networking platforms, repeat steps 1 and 2 for all devices before proceeding.

3. Warm reboot the server.

Step by step process:

Info

Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of BlueField Arm cores.

   Info

   This operation is expected to finish within 15 seconds.

   Possible methods:

   - From the BlueField OS:

     ```
     shutdown -h now
     ```

     Or:

     ```
     mlxfwreset -d /dev/mst/mt*pciconf0 -l 1 -t 4 --sync 0 r
     ```

   - From the host OS:

Info

Not relevant when the BlueField is operating in Zero-Trust Mode.

```
mlxfwreset -d <mst-device> -l 1 -t 4 r
```

- Using the BlueField BMC:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password> power soft
```

Or using Redfish (BlueField-3 and above):

```
curl -k -u root:<password> -H "Content-Type: application/json" -X POST https://<bmc_ip>/redfish/v1/Systems/Bluefield/Actions/ComputerSystem.Reset -d '{"ResetType": "GracefulShutdown"}'
```

2. Query BlueField state.

   Possible methods:

   - From the host OS:

Not relevant when the BlueField is operating in Zero-Trust Mode.

```
echo DISPLAY_LEVEL 2 > /dev/rshim0/misc
cat /dev/rshim0/misc
```

Expected output:

```
INFO[BL31]: System Off
```

- Utilizing the BlueField BMC:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P
<password> raw 0x32 0xA3
```

Expected output: `06`.

3. Warm reboot the server f rom the host OS:

```
mlxfwreset -d <mst-device> -l 4 r
```

Note

If multiple BlueField devices are present in the host, this command must run only once. In this case, the MST device can

be of any of the BlueFields for which the reset is necessary and participated in step 1.

Or:

```
reboot
```

Note

For external hosts which do not toggle PERST# in their standard reboot command, use the `mlxfwreset` option.

System-level Reset for BlueField in NIC Mode

Perform warm reboot of the host OS:

```
mlxfwreset -d <mst-device> -l 4 r
```

Or:

```
reboot
```

Note

For external hosts which do not toggle PERST# in their standard reboot command, use the `mlxfwreset` option.

System-level Reset for Host with Separate Power Control

This procedure is a special use case relevant only to host platforms with separate power control for the PCIe slot and CPUs, in which the BlueField (running in DPU mode) is provided power while host OS/CPUs may be in shutdown or similar standby state (this allows the BlueField device to be operational while the host CPU is in shutdown/standby state).

The following is the high-level flow of the procedure:

1. Graceful shutdown of host OS or similar CPU standby.
2. Graceful shutdown of BlueField Arm cores.
3. Query BlueField state to affirm shutdown reached.
4. Full BlueField Reset
5. Query BlueField state to affirm operational state reached

   Info

   In systems with multiple BlueField networking platforms, repeat steps 1 through 5 for all devices before proceeding.

6. Power on the server.

Step by step process:

Info

Some of the following steps can be performed using different methods, depending on resource availability and support in the user's environment.

1. Graceful shutdown of host OS by any means preferable.
2. Graceful shutdown of BlueField Arm cores.

Info

This step normally takes up to 15 seconds to complete.

   - From the BlueField OS:

```
shutdown -h now
```

   - Utilizing the BlueField BMC:
      - Using IPMI:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P
<password> power soft
```

      - Using Redfish (for BlueField-3 and above):

```
curl -k -u root:<password> -H "Content-Type: application/json" -X
POST
```

```
https://<bmc_ip>/redfish/v1/Systems/Bluefield/Actions/ComputerSystem.Reset -d
'{"ResetType": "GracefulShutdown"}'
```

3. Query the BlueField's state utilizing the BlueField BMC:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password>
raw 0x32 0xA3
```

Expected output: `06`.

4. Perform BlueField hard reset utilizing the BlueField BMC:

> **Info**
>
> This step takes up to 2 minutes to complete .

- Using IPMI:

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P
<password> power cycle
```

- Using Redfish (for BlueField-3 and above):

```
curl -k -u root:<password> -H "Content-Type: application/json" -X
POST https://<bmc_ip>/redfish/v1/Systems/Bluefield/Actions/ComputerSystem.Reset -d
'{"ResetType" : "PowerCycle"}'
```

5. Query BlueField operational state u tilizing the BlueField BMC :

Info

At this point, the BlueField is expected to b e operational .

```
ipmitool -C 17 -I lanplus -H <bmc_ip> -U root -P <password>
raw 0x32 0xA3
```

Expected output: `05`.

6. Power on/boot up the host OS.

BlueField DPU Reset Using a BMC Platform

The BlueField DPU can also be reset from a BMC platform using NC-SI command over I2C. This option is more common in DPU BMC absence, when the BlueField DPU is running in NIC mode or when it is used as a controller.

The reset is performed using the *Reset BlueField-3 DPU (Command=0x12, Parameter=0xB)*, which allows a BMC platform to reset the NVIDIA BlueField-3 DPU device. This command is only applicable to BlueField-3 devices.

The Reset BlueField-3 DPU command is addressed to the package only. When the internal reset is complete, the BMC platform should reconfigure the device.

Info

The *Reset BlueField-3 DPU* command is supported on BlueField-2 and later devices.

Reset BlueField-3 DPU Format

Bytes/Bits	31:24	23:16	15:8	7:0	
0:15	NC-SI Header (OEM Command)				
16:19	NVIDIA Manufacture ID (IANA) = 0x8119				
20:23	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x0B	NICR	Mode
24:27	Checksum 31:0				

The parameter descriptions for Reset BlueField-3 DPU command are provided below.

Reset BlueField-3 DPU Parameters

Field	Description
NICR	• 0 - NIC does not reset. Only the embedded CPU will reset. • 1 - Reset the embedded CPU and the NIC
Mode	This field defines the type of conditions to use before performing the internal reset • 0 - The internal reset will start after sending the response to this command • 1 - The internal reset will start only when all the hosts asserts their PERST# signals low • 2 - The internal reset will start only when all the hosts disabled their PCIe links. This may or may not include assertion of their respective PERST# signals low. • Other - Reserved

Reset BlueField-3 DPU Response

The ConnectX adapter responds to a Reset BlueField-3 DPU command when the package ID matches, and with no checksum error.

Reset BlueField-3 DPU Response Format

Bytes/Bits	31:24	23:16	15:8	7:0
0:15	NC-SI Header (OEM Command)			
16:19	Response Code		Reason Code	
20:23	NVIDIA Manufacture ID (IANA) = 0x8119			

Bytes/Bits	31:24	23:16	15:8	7:0	
24:27	Command rev=0x00	MLNX Cmd ID=0x12	Parameter=0x0B	NICR	Mode
28:31	Checksum 31:0				

DOCA Virtual Functions User Guide

This guide provides an overview and configuration of virtual functions for NVIDIA® BlueField® and demonstrates a use case for running the DOCA applications over x86 host.

Introduction

Single root IO virtualization (SR-IOV) is a technology that allows a physical PCIe device to present itself multiple times through the PCIe bus. This technology enables multiple virtual instances of the device with separate resources. NVIDIA adapters are able to expose virtual instances or functions (VFs) for each port individually. These virtual functions can then be provisioned separately.

Each VF can be seen as an additional device connected to the physical interface or function (PF). It shares the same resources with the PF, and its number of ports equals those of the PF.

SR-IOV is commonly used in conjunction with an SR-IOV-enabled hypervisor to provide virtual machines direct hardware access to network resources, thereby increasing its performance.

There are several benefits to running applications on the host. For example, one may want to utilize a strong and high-resource host machine, or to start DOCA integration on the host before offloading it to the BlueField DPU.

The configuration in this document allows the entire application to run on the host's memory, while utilizing the HW accelerators on BlueField.

When VFs are enabled on the host, VF representors are visible on the Arm side which can be bridged to corresponding PF representors (e.g., the uplink representor and the host representor). This allows the application to only scan traffic forwarded to the VFs as configured by the user and to behave as a simple "bump-on-the-wire". DOCA installed on the host allows access to the hardware capabilities of the BlueField DPU without comprising features which use HW offload/steering elements embedded inside the eSwitch.

Prerequisites

To run all the reference applications over the host, you must install the host DOCA package. Refer to the DOCA Installation Guide for Linux for more information on host installation.

VFs must be configured as trusted for the hardware jump action to work as intended. The following steps configure "trusted" mode for VFs:

1. Delete all existing VFs
    1. To delete all VFs on a PF run the following on the host:

       ```
       host $ echo 0 >
       /sys/class/net/<physical_function>/device/sriov_numvfs
       ```

       For example:

       ```
       host $ echo 0 >
       /sys/class/net/ens1f0/device/sriov_numvfs
       ```

2. Delete all existing SFs.

Info

Refer to BlueField Scalable Function User Guide for instructions on deleting SFs.

3. Stop the main driver on the host:

```
host $ /etc/init.d/openibd stop
```

4. Before creating the VFs, set them to "trusted" mode on the device by running the following commands on the DPU side.

    1. Setting VFs on port 0:

    ```
    host $ mlxreg -d /dev/mst/mt41686_pciconf0 --reg_id 0xc007 --reg_len 0x40 --indexes "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
    ```

    2. Setting VFs on port 1:

    ```
    host $ mlxreg -d /dev/mst/mt41686_pciconf0.1 --reg_id 0xc007 --reg_len 0x40 --indexes "0x0.0:32=0x80000000" --yes --set "0x4.0:32=0x1"
    ```

Note

These commands set trusted mode for all created VFs/SFs after their execution on the DPU.

Note

Setting trusted mode should be performed once per reboot.

5. Restart the main driver on the host by running the following command:

```
host $ /etc/init.d/openibd restart
```

VF Creation

1. Make sure mst driver is running:

   ```
   host $ sudo mst status
   ```

   If it is not loaded, run:

   ```
   host $ sudo mst start
   ```

2. Enable SR-IOV. Run:

   ```
   host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s
   SRIOV_EN=1
   ```

3. Set number of VFs. Run:

   ```
   host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s
   NUM_OF_VFS=X
   ```

   Note

Perform a BlueField system reboot for the `mlxconfig` settings to take effect.

```
host $ echo X >
/sys/class/net/<physical_function>/device/sriov_numvfs
```

For example:

```
host $ sudo mlxconfig -y -d /dev/mst/mt41686_pciconf0 s
NUM_OF_VFS=2
host $ reboot
host $ echo 2 > /sys/class/net/ens1f0/device/sriov_numvfs
```

After enabling VF, the representor appears on the DPU. The function itself is seen at the x86 side.

4. To verify that the VFs have been created. Run:

```
host $ lspci | grep Virtual
b1:00.3 Ethernet controller: Mellanox Technologies ConnectX
Family mlx5Gen Virtual Function (rev 01)
b1:00.4 Ethernet controller: Mellanox Technologies ConnectX
Family mlx5Gen Virtual Function (rev 01)
b1:01.3 Ethernet controller: Mellanox Technologies ConnectX
Family mlx5Gen Virtual Function (rev 01)
```

Note

2 new virtual Ethernet devices are created in this example.

Running DOCA Application on Host

Note

Allocate the required number of VFs as explained previously.

Note

Allocate any other resources as specified by the application (e.g., huge pages).

The following is the CLI example for running a reference application over the host using VF:

```
doca_<app_name> -a "pci address VF0" -a "pci address VF1" -c 0xff
-- [application flags]
```

The following is an example with specific PCIe addresses for the VFs:

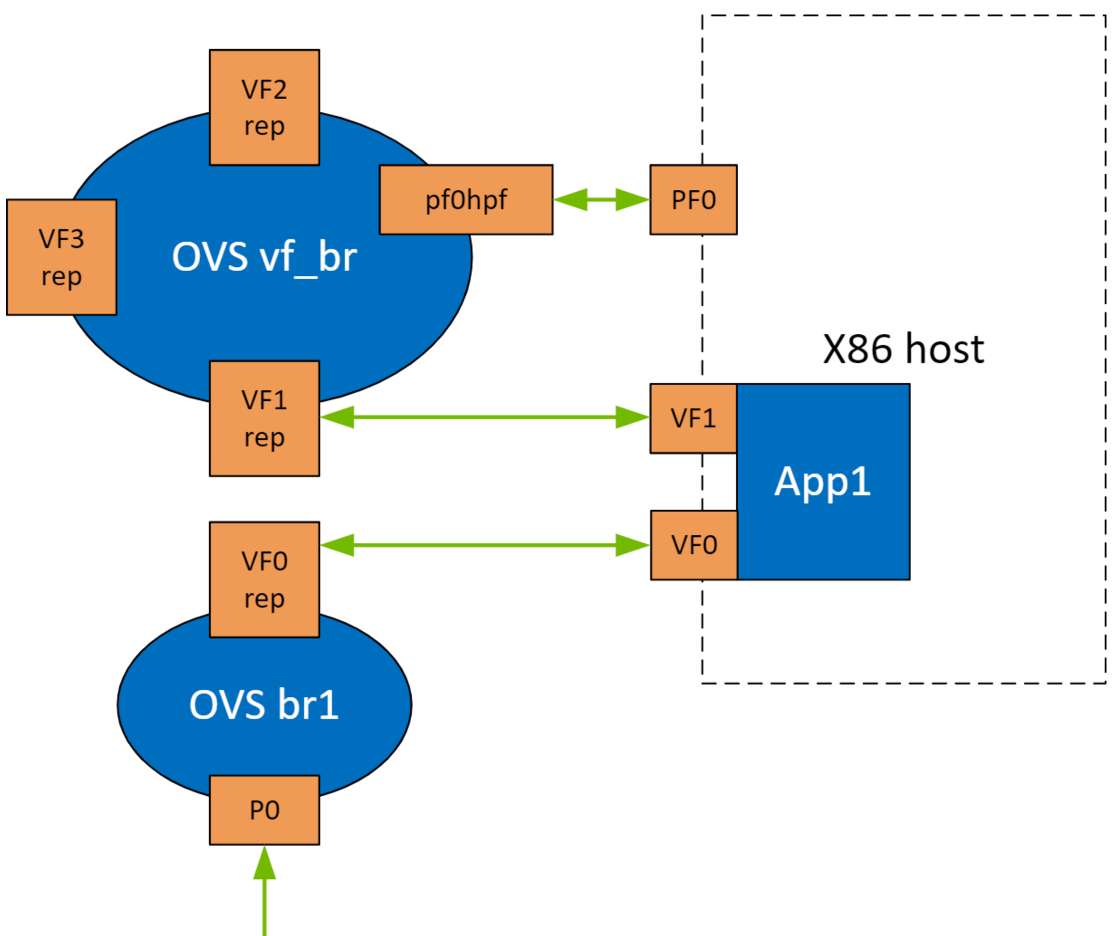
```
doca_<app_name> -a b1:00.3 -a b1:00.4 -c 0xff -- -l 60
```

Note

By default, a DPDK application initializes all the cores of the device. This is usually unnecessary and may even cause unforeseeable issues. It is recommended to limit the number of cores, especially when using an AMD-based system, to 16 cores using the `-c` flag when running DPDK.

Topology Example

The following is a topology example for running the application over the host.



Configure the OVS on BlueField as follows:

```
Bridge vf_br:
    Port vf_br
       Interface vf_br
```

```
            type: internal
     Port pf0hpf
        Interface pf0hpf
    Port pf0vf1
        Interface pf0vf1
Bridge ovsbr1
    Port ovsbr1
        Interface ovsbr1
            type: internal
    Port p0
        Interface p0
    Port pf0vf0
        Interface pf0vf0
```

When enabling a new VF over the host, VF representors are created on the Arm side. The first OVS bridge connects the uplink connection (`p0`) to the new VF representor (`pf0vf0`), and the second bridge connects the second VF representor (`pf0vf1`) to the host representors (`pf0phf`). On the host, the 2 PCIe addresses of the newly created function must be initialized when running the applications.

When traffic is received (e.g., from the uplink), the following occurs:

1. Traffic is received over `p0`.
2. Traffic is forwarded to `pf0vf0`.
3. Application "listens" to `pf0vf0` and `pf0vf1` and can, therefore, acquire the traffic from `pf0vf0`, inspect it, and forward to `pf0vf1`.
4. Traffic is forwarded from `pf0vf1` to `pf0hpf`.

VF Creation on Adapter Card

Note

Supported only for NVIDIA® ConnectX®-6 Dx based adapter cards and higher.

The following steps are required only when running DOCA applications on an adapter card.

1. Set trust level for all VFs. Run:

```
host $ mlxreg -d /dev/mst/mt4125_pciconf0 --reg_name
VHCA_TRUST_LEVEL --yes --set "all_vhca=0x1,trust_level=0x1" --indexes
"vhca_id=0x0,all_vhca=0x0"
```

2. Create X VFs (X being the required number of VFs) and run the following to turn on trusted mode for the created VFs:

```
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/X/trust
```

For example, if you are creating 2 VFs, the following commands should be used:

```
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/0/trust
echo ON | tee /sys/class/net/enp1s0f0np0/device/sriov/1/trust
```

3. Create a VF representor using the following command, replace the PCIe address with the PCIe address of the created VF:

```
echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/unbind
echo 0000:17:00.2 > /sys/bus/pci/drivers/mlx5_core/bind
```

BlueField SR-IOV

The NVIDIA® BlueField® SR-IOV solution is based on asymmetric VF and enables per-ECPF and per PF control over number of VF allocation .

ECPF VFs are intended to be used in switchdev mode. Like SFs and host VFs, ECPF VFs have a representor. Representor naming for ECPF VFs start after the host VFs. For example, if the host has 32 VFs enabled, then the host VF representors are named `pf0vf0`-`pf0vf31`, and the Arm representors continue at `pf0vf32` onward.

To enable BlueField SR-IOV, apply the following configuration in the BlueField OS:

```
mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF_VALID=1
```

> **Note**
>
> Once `PF_NUM_OF_VF_VALID` is set, the `NUM_OF_VFS` mlxconfig option is not relevant and the user must set `PF_NUM_OF_VF` for each host and EC function. It is recommended for the number of VFs for each ECPF and each host PF be the same.

The BlueField should now support setting asymmetric VF configuration per port.

The following are examples for configuring the number of VFs per port:

1. In the BlueField, issue the following commands to configure 32 VFs per port:

    ```
    bf> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=32
    bf> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=32
    ```

> **Note**
>
> The BlueField ECPF driver in the BlueField's Arm OS limits the number of VFs it supports to 32 per port.

2. In the host OS, issue the following commands to configure up to 126 VFs per port:

```
host> mlxconfig -d 03:00.0 -y s PF_NUM_OF_VF=126
host> mlxconfig -d 03:00.1 -y s PF_NUM_OF_VF=126
```

3. Perform a BlueField system reboot for the `mlxconfig` settings to take effect.
4. Create ECPF VFs:

```
echo 1 > /sys/class/net/p0/device/sriov_numvfs
```

> **Note**
>
> BlueField SR-IOV VFs do not support the following legacy SR-IOV functionalities:
>
> - Virtual switch tagging (VF VLAN)
> - Spoof check
> - VF trust
> - VF rate

fTPM over OP-TEE

> **Note**
>
> fTMP over OP-TEE is supported on NVIDIA® BlueField®-3 DPUs and higher only on host OS Ubuntu 22.04 or Oracle Linux.

The Trusted Computing Group (TCG) is responsible for the specifications governing the trusted platform module (TPM). In many systems, the TPM provides integrity measurements, health checks and authentication services.

Attributes of a TPM:

- Support for bulk (symmetric) encryption in the platform
- High quality random numbers
- Cryptographic services
- Protected persistent store for small amounts of data, sticky bits, monotonic counters, and extendible registers
- Protected pseudo-persistent store for unlimited amounts of keys and data
- Extensive choice of authorization methods to access protected keys and data
- Platform identities
- Support for platform privacy
- Signing and verifying digital signatures
- Certifying the properties of keys and data
- Auditing the usage of keys and data

With TPM 2.0., the TCG creates a library specification describing all the commands or features that could be implemented and may be necessary in servers, laptops, or embedded systems. Each platform can select the features needed and the level of security or assurance required. This flexibility allows the newest TPMs to be applied to many embedded applications.

Firmware TPM (fTPM) is implemented in protected software. The code runs on the main CPU so that a separate chip is not required. While running like any other program, the code is in a protected execution environment called a trusted execution environment (TEE) which is separate from the rest of the programs running on the CPU. By doing this, secrets (e.g., private keys perhaps needed by the TPM but should not be accessed by others) can be kept in the TEE creating a more secure environment.

Info

fTPM provides similar functionality to a chip-based TPM, but does not require extra hardware. It complies with the official TCG reference implementation of the TPM 2.0 specification . The source code of this implementation is located here.

Info

fTPM f ully supports TPM2 Tools and the TCG TPM2 Software Stack (TSS).

Characteristics of an fTPM:

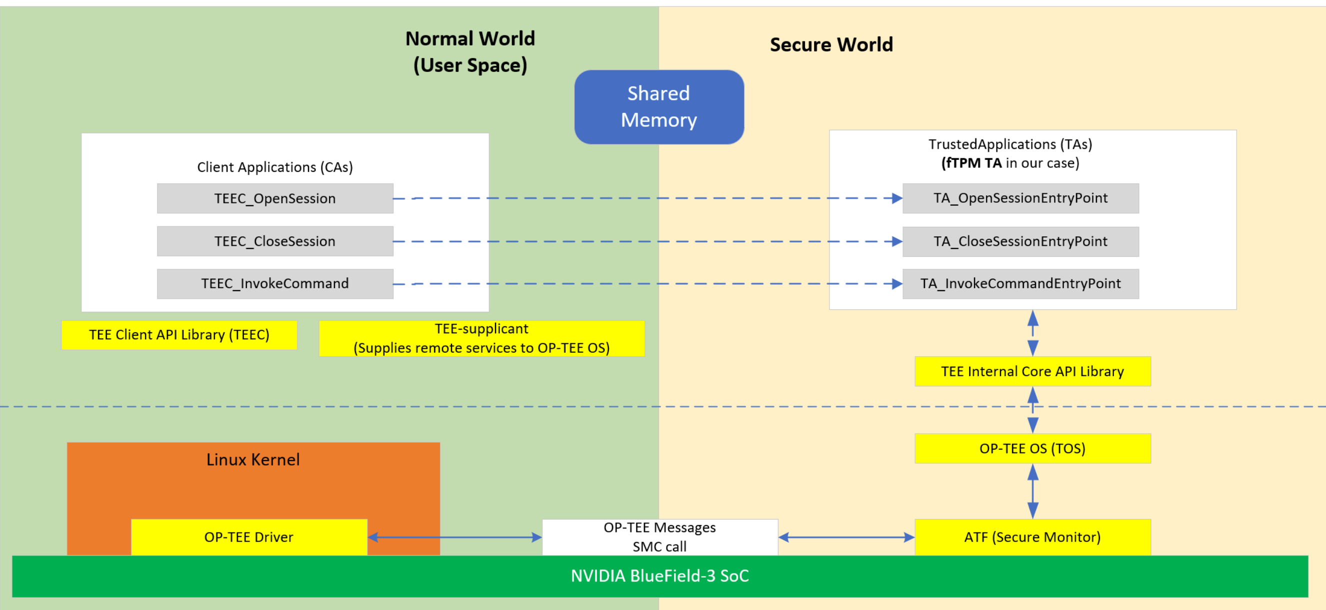
- Emulated TPM using an isolated hardware environment
- Executes in an open-source trusted execution environment (OP-TEE)
- fTPM trusted application (TA) is part of the OP-TEE binary. This allows early access on bootup, runs only in secure DRAM.

Info

Currently, the only TA supported is fTPM.

- fTPM is not a task waiting to be woken up. It only executes when TPM primitives are forwarded to it from the user space. It is guaranteed shielded execution via the TEE OS and, when invoked via the TEE Dispatcher, runs to completion.

The fTPM TA is the only TA BlueField-3 currently supports. Any TA loaded by OP-TEE must be signed (signing done externally) and then authenticated by OP-TEE before being allowed to load and execute.

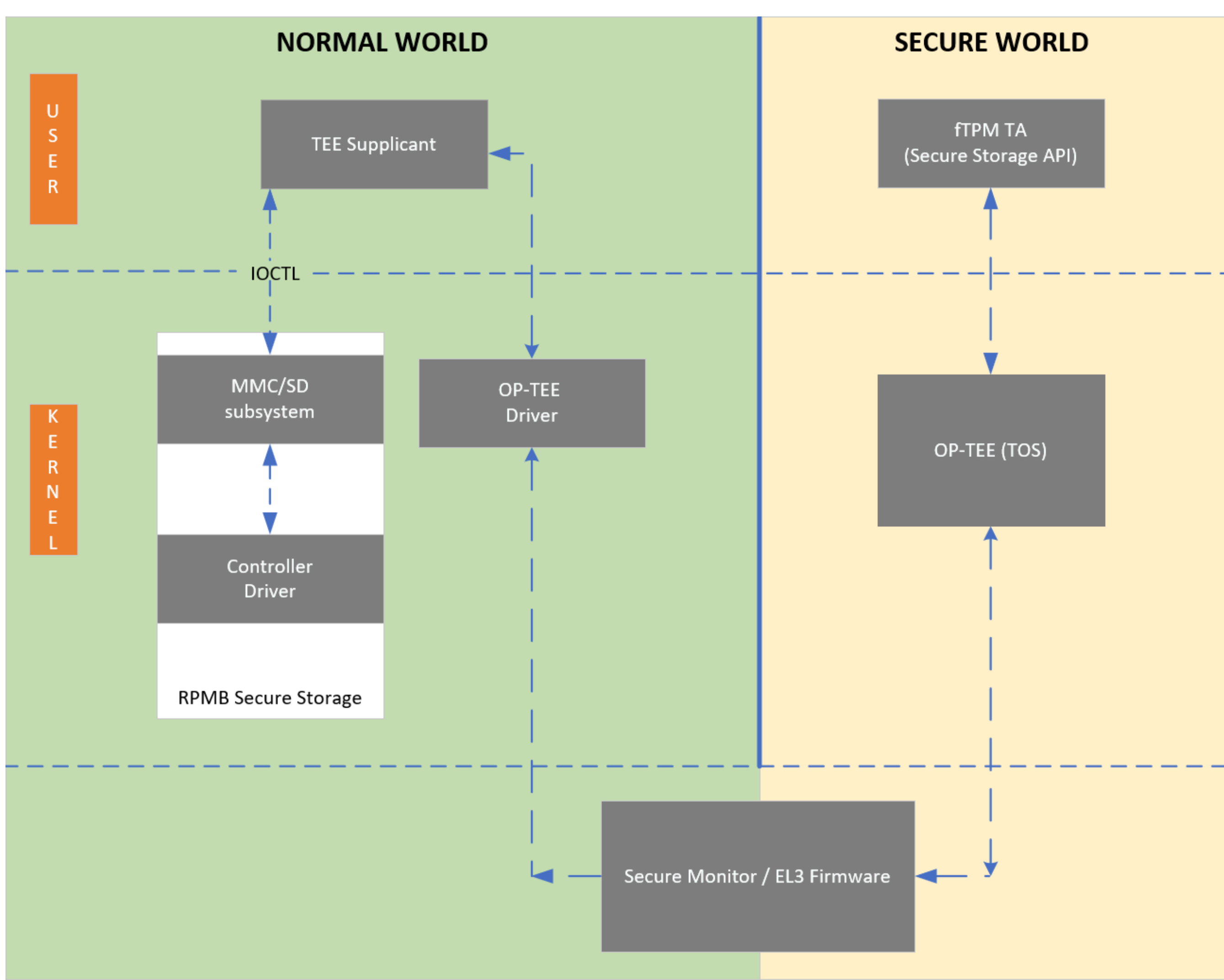


A replay-protected memory block (RPMB) is provided as a means for a system to store data to the specific memory area in an authenticated and replay-protected manner, making it readable and writable only after a successful authentication read/write accesses. The RPMB is a dedicated partition available on the eMMC, which makes it possible to store and retrieve data with integrity and authenticity support. A signed access to an RPMB is supported by first programming authentication key information to the eMMC memory (shared secret). The RPMB authentication key is programmed into BlueField at manufacturing time.

Info

RPMB features a 4MB partition secure storage for BlueField-3.

There is no eMMC controller driver in OP-TEE. All device operations have to go through the normal world via the TEE-supplicant daemon, which relies on the Linux kernel's ioctl interface to access the device. All writes to the RPMB are atomic, authenticated, and encrypted. The RPMB partition stores data in an authenticated, replay-protected manner, making it a perfect complement to fTPM for storing and protecting data.



Enabling OP-TEE on BlueField-3

Enable OP-TEE in the UEFI menu:

1. ESC into the UEFI on BlueField boot.
2. Navigate to Device Manager > System Configuration.
3. Check "Enable OP-TEE".
4. Save the change and reset/reboot.

5. Upon reboot OP-TEE is enabled.

Note

OP-TEE is essentially dormant (does not have an OS scheduler) and reacts to external inputs.

Verifying BlueField-3 is Running OP-TEE

Users can see the OP-TEE version during BlueField-3 boot:

```
Nvidia BlueField-3 rev1 BL1 V1.0
INFO: psc supervisor init.
INFO: psc_irq_init...
INFO: force_crs_enable=0 pcr.lock0 = 0, time = 111291
INFO: enter idle task.
NOTICE:  Running as 9009D3B400ENEA system
NOTICE:  BL2: v2.2(release):4.5.0-16-g2bd9b06e2-dirty
NOTICE:  BL2: Built : 15:43:42, Sep  7 2023
NOTICE:  BL2 built for hw (ver 2)
NOTICE:  # Finished initializing DDR MSS1
NOTICE:  DDR POST passed.
INFO: mailbox rx: channel = 2, code = 0x43544c44
NOTICE:  BL31: v2.2(release):4.5.0-16-g2bd9b06e2-dirty
NOTICE:  BL31: Built : 15:43:44, Sep  7 2023
NOTICE:  BL31 built for hw (ver 2), lifecycle Production

PTM:171288:2:0:6~
I/TC: 
I/TC: OP-TEE version: 3.10.0-21-g450b24a (gcc version 8.3.0 (GCC)) #1 Sat Aug 26 11:54:32 UTC 2023 aarch64
I/TC: Primary CPU initializing
I/TC: Primary CPU switching to normal world boot
UEFI firmware (version BlueField:4.5.0-16-g0e7fa9c192-BId0 built at 20:53:10 on Sep  6 2023)
```

The following indicators should all be present if fTPM over OP-TEE is enabled:

- Check "dmesg" for the OP-TEE driver initializing

```
root@localhost ~]# dmesg | grep tee
[    5.646578] optee: probing for conduit method.
[    5.653282] optee: revision 3.10 (450b24ac)
[    5.653991] optee: initialized driver
```

- Verify that the following kernel modules are loaded (running):

```
[root@localhost ~]# lsmod | grep tee
tpm_ftpm_tee           16384  0
optee                  49152  1
tee                   49152  3 optee,tpm_ftpm_tee
```

- Verify that the proper devices are created/available (4 in total):

```
[root@localhost ~]# ls -l /dev/tee*
crw------- 1 root root 234,  0 Sep  8 18:24 /dev/tee0
crw------- 1 root root 234, 16 Sep  8 18:24 /dev/teepriv0

[root@localhost ~]# ls -l /dev/tpm*
crw-rw---- 1 tss root  10,   224 Sep  8 18:24 /dev/tpm0
crw-rw---- 1 tss tss  252, 65536 Sep  8 18:24 /dev/tpmrm0
```

- Verify that the required processes are running (3 in total):

```
[root@localhost ~]# ps axu | grep tee
root         707  0.0  0.0  76208  1372 ?        Ssl  14:42
0:00 /usr/sbin/tee-supplicant
root         715  0.0  0.0      0     0 ?        I<   14:42
0:00 [optee_bus_scan]

[root@localhost ~]# ps axu | grep tpm
root         124  0.0  0.0      0     0 ?        I<   18:24
0:00 [tpm_dev_wq]
```

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the

consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright 2025. PDF Generated on 05/05/2025