NVIDIA DOCA DPI

Programming Guide

Table of Contents

Chapter 1. Introduction

> Important: No updates were made to the DOCA DPI library in DOCA 2.2. Please refer to DOCA 2.5 for a note regarding future DPI updates.

Deep packet inspection (DPI) is a method of examining the full content of data packets as they traverse a monitored network checkpoint.

DPI provides a more robust mechanism for enforcing network packet filtering as it can be used to identify and block a range of complex threats hiding in network data streams more accurately. This includes:

- ▶ Malicious applications
- ▶ Malware data exfiltration attempts
- ▶ Content policy violations
- ▶ Application recognition
- ▶ Load balancing

The DOCA DPI library supports inspection of the following protocols:

- ▶ HTTP 2.0/1.1/1.0
- ▶ TLS/SSL client hello and certificate messages
- ▶ DNS

TCP/UDP stream-based signatures may detect applications on other protocols.

Chapter 2.   Prerequisites

DPI-based applications can run either on the host machine, or on the NVIDIA® BlueField® DPU target. Since the DPI leverages the Regular Expressions (RegEx) Engine, users must make sure it is enabled.

The RegEx engine is enabled by default on the DPU. To use DPI directly on the host, run:
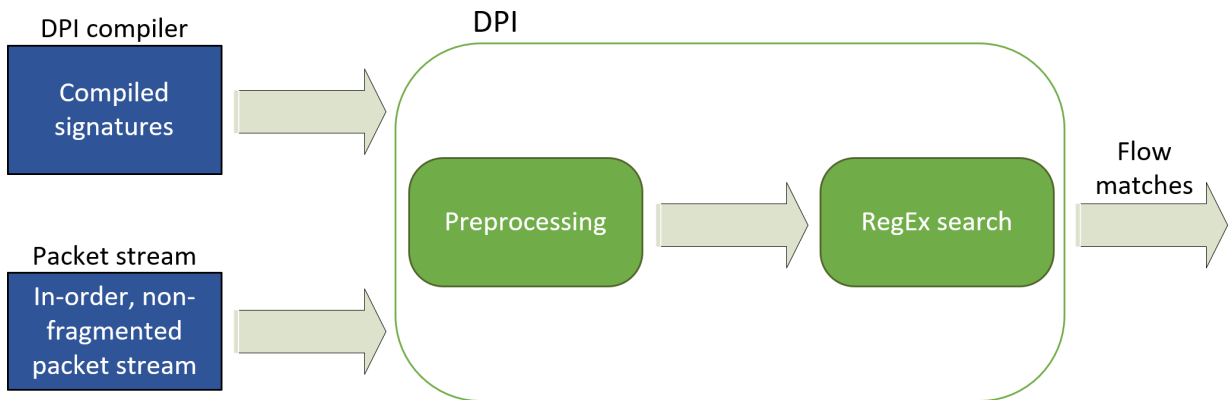
```
host> sudo /etc/init.d/openibd stop
dpu> echo 1 > /sys/class/net/p0/smart_nic/pf/regex_en
dpu> cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
400
# Make sure to allocate 200 additional hugepages
dpu> sudo echo 600 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
dpu> systemctl start mlx-regex
# Verify the service is properly running
dpu> systemctl status mlx-regex
host> sudo /etc/init.d/openibd start
```

> 🗩 Note: Commands with the `host` prompt must be run on the host. Commands with the `dpu` prompt must be run on BlueField (Arm).

Chapter 3. Architecture

The following diagram shows how the DPI library receives for processing a compiled signatures files and a stream of "in-order" and non-fragmented packets, and returns matches per flow.

DPI compiler

Compiled signatures

Packet stream

In-order, non-fragmented packet stream

DPI

Preprocessing

RegEx search

Flow matches

3.1.   Signature Database

The signature database is compiled into a CDO file by the DPI compiler. The CDO file includes:

▶ Compiled RegEx engine rules

▶ Other signature information

The application may load a new database while the DPI is processing packets.

For more information on the DPI compiler, please refer to the NVIDIA DOCA DPI Compiler document.

> Note: Programs using DOCA DPI library version 2.0 and above must use CDO files generated by DOCA DPI Compiler version 2.0 and above.

3.2. DPI Queue

A DPI queue, implemented as a `doca_workq`, is designed to be used by a worker thread. The DPI queue holds the flow's state. Therefore, all packets from both directions of the flow must be submitted to the same DPI queue in-order. Each packet must be injected as a `doca_buf` along with a flow context and a direction.

A flow direction is usually represented by a 5-tuple, but it can also be a 3-tuple for other protocols.

> Note: From the application's side, the connection tracking (CT) logic must handle out-of-order packets as well as fragmented packets. Once a connection has timed out or terminated, the application must notify the DPI library as well.

Chapter 4. API

For the library API reference, refer to DPI API documentation in [NVIDIA DOCA Libraries API Reference Manual](#).

> 📝 Note: The pkg-config (`*.pc` file) for the DPI library library is included in DOCA's regular definitions (i.e., `doca`).

The DPI library is based on DOCA core objects and follows the standard execution model.

The following sections provide additional details about the library API and they indicate the order of operations.

4.1. Initialization and Configuration

A `doca_dpi` instance should be created with the following function:

```
doca_error_t doca_dpi_create(struct doca_dpi **dpi);
```

The following helper function can be used to map the `doca_dpi` instance to generic `doca_ctx`:

```
struct doca_ctx *doca_dpi_as_ctx(struct doca_dpi *dpi);
```

With the `doca_ctx`, standard DOCA core functions can be utilized to add a device to be used with DPI (`doca_ctx_dev_add()`) and to start the DPI context (`doca_ctx_start()`).

> 📝 Note: Adding a device to a `doca_dpi` instance prior to starting is mandatory. The API calls `doca_dpi_is_supported()`, and `doca_dpi_job_get_supported()` can be used to check the compatibility of a given DOCA device.

Prior to starting the context, the following API calls can be used to configure the DPI library:

API Call	Notes
`doca_dpi_set_per_workq_packet_pool_size()`	DPI workqs can clone and hold references to buffers even after they are returned to the application which allows cross-packet signatures to be detected within a flow when more packets arrive. This function sets the maximum number of such references that can be held per DPI workq.

API Call	Notes
	It is important that the application selects an appropriate value for the packet pool size. If the value is too low then buffer cloning may fail, leading to enqueue returning a `DOCA_ERROR_NO_MEMORY` error. However, setting a value too high may cause undesired behavior.
	For example, if the application is reading packets from a descriptor pool of size N, having a packet pool value greater than N could lead to situations where DPI holds clone references to all descriptors and none are released potentially causing stalemate.
	It is recommended that the application accounts for this and sets a value smaller than the total potential input buffers. If memory errors are returned on enqueue then the application should free descriptors by timing out old flows.
`doca_dpi_set_per_workq_max_flows()`	Each DPI workq can hold contexts for a given number of active flows at once. This value defaults to 10,000 but can be configured via this API call if the user suspects the value is too low or if they want to reduce pre-allocated memory overheads by setting a lower value.
`doca_dpi_set_max_sig_match_len()`	This function sets the maximum length signature match in bytes that the DPI library guarantees to find. The DPI library may return matches longer than this value but may miss some of them. It guarantees to find any matches of the given length or smaller within a flow. This defaults to 5000.
	The value selected here should be considered alongside the value set in `doca_dpi_set_per_workq_packet_pool_size()`. If a large value is used then more packets within a flow must be cloned and stored for longer, using more entries from the packet pool. For example, if a flow has received 4 packets of 1500 bytes and the max signature value is set to 10,000, then DPI must hold all 4 packets until further packets arrive as a signature of 10,000 bytes could start in the first packet and complete in packets not yet received (only 6000 bytes currently examined). However, if the max signature length is set to 1000 bytes, then DPI must only store the latest packet as no match of 1000 can start prior to the latest packet and not have already been detected.

API Call	Notes
`doca_dpi_set_in_order_mode()`	DPI uses a hardware-accelerated pattern matcher which can process data in parallel. Therefore, it is possible that some packets or buffers are processed quicker than others and that results are received in a different order than they are sent. If the application must receive results to jobs in the same order that they were sent, then the user can call this function to enforce in-order processing. This value is turned off by default as enabling in-order processing may negatively impact performance.

4.2. Signature Loading

Before enqueueing packets for processing, the DPI library must be loaded with signatures by the main thread.

Signature loading can take place during the configuration of the `doca_dpi` instance or after it has been started using the following function:

```
doca_error_t doca_dpi_set_signatures(struct doca_dpi *dpi, const char *cdo_file);
```

The function can be used during runtime to update the currently loaded signatures. The DPI context does not need to be stopped for this to happen.

4.3. DPI Queue Creation

DPI queues are implemented as DOCA workqs. It is recommended that one queue is used per worker core in the application.

Workqs should be created after the `doca_ctx` instance is started and should be added to the `doca_dpi` instance using the standard DOCA core calls `doca_workq_create()` and `doca_ctx_workq_add()`.

4.4. Flow Life Cycle and Packet Processing

1. Once a new flow is detected by the connection tracking software, the user should notify the DPI library using the following function:

```
doca_error_t doca_dpi_flow_create(struct doca_dpi *dpi, struct doca_workq *workq,
                                  const struct doca_dpi_parsing_info
 *parsing_info,
                                  struct doca_dpi_flow_ctx **flow_ctx);
```

The user must inform DPI which `doca_workq` the flow should be created on.
The extracted L3 and L4 information should also be passed in a populated

`doca_dpi_parsing_info` struct. The function gives the calling application an opaque pointer to a flow context.

2. Every incoming packet classified for this flow should be packaged into a `doca_dpi_job` and enqueued to the same workq the flow is created on by using the standard DOCA workq function `doca_workq_submit()`.

Packets or job buffers must be represented as type `doca_buf` which is a standard format for passing data to DOCA libs. DOCA provides libraries (e.g., `doca_dpdk_bridge`) which allow the conversion of packet acquisition formats such as DPDK's mbuf to `doca_buf`.

A `doca_dpi_job` format is as follows:

```
struct doca_dpi_job {
        struct doca_job base;
        const struct doca_buf *pkt;
        bool initiator;
        uint32_t payload_offset;
        struct doca_dpi_flow_ctx *flow_ctx;
        struct doca_dpi_result *result;
};
```

A standard `doca_job` struct should begin the job followed by a pointer to the `doca_buf` representing the packet, an indicator of the direction of the packet within the flow, any offset of the payload within the `doca_buf`, and the flow context pointer for the given flow (previously generated with `doca_dpi_flow_create()`).

The final field is a pointer to a `doca_dpi_result` struct. This memory must be held by the calling application and maintained until the job response is dequeued. It is this memory location that holds the job result information at dequeue time.

> 🗨 Note: It is user responsibility to ensure that the L3 and L4 information of the enqueued packet matches that of the parsing info of the respective flow (the direction can be different).

> 🗨 Note: For every `doca_buf` injected, the user is not allowed to free the `doca_buf` until the `doca_buf` is dequeued.

3. To poll the results, the application must call the DOCA core workq function `doca_workq_progress_retrieve()`.

On success, the function returns a `doca_event` popped from the associated workq, and the `doca_dpi_result` result field for that job is populated. A pointer to this result data can be accessed from the `result` field in the event struct.

There is also a `user_data` field available in the `doca_job` struct of the enqueue job that is returned in the `doca_event` if the application must sync further information between the enqueue and dequeue.

The result data will be in the following format:

```
struct doca_dpi_result {
        const struct doca_buf *pkt;
        bool matched;
        struct doca_dpi_sig_info info;
        int status_flags;
}
```

The result passes back a pointer to the packet that triggered the job, whether or not the packet has a match (based on the entire flow), signature information on the highest priority match, and status flags indicating the status of the flow and if the match detected is new or had previously been detected in the flow.

> 📝 Note: The returning of the `doca_buf` in the result does not guarantee that the DPI library is finished with it. The DPI library may still hold a clone of the buffer which increases ref counters and may prevent it from being properly released by the application.

4. When the connection tracking software detects that the flow is terminated or aged-out, the application should notify the DPI library by calling `doca_dpi_flow_destroy()`. This function only requires the flows associated `doca_dpi_flow_ctx` as a parameter. Calling this function releases all resources held by the flow including any cloned buffer references.

4.5. Teardown

When the application is finished with the DPI library it should follow the standard DOCA core template to end a run.

The workqs must be removed from the context and destroyed using `doca_ctx_workq_rm()` and `doca_workq_destroy()`.

Following this, the DOCA context can be stopped with `doca_ctx_stop()` and the DPI instance removed with `doca_dpi_destroy()`.

> 📝 Note: As stated previously, the DPI library may hold references to application-generated buffers that it has cloned. Destroying a DPI workq releases the clones that are still present on that workq. Therefore, it is advisable to remove and destroy the DPI workqs before attempting to close or free the buffer generation utility.
>
> For example, if `doca_dpdk_bridge` had been used to generate `doca_buf`s sent to DPI, then you may not be able to release the bridge due to buffers still being in use if you attempt to do it before removing the DPI workqs.

NVIDIA DOCA DPI   MLNX-15-060466 _v2.2.0 | 9

Chapter 5. DOCA DPI Samples

This section provides DPI sample implementation on top of the BlueField DPU.

5.1. Sample Prerequisites

DPI-based applications can run either on the host machine or on the DPU target.

As DPI leverages the regular expressions (RegEx) engine, users must make sure it is enabled:

1. Allocate huge pages on the host side if running from x86:
   ```
   sudo echo 2048 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```
2. Make sure the RegEx engine is active on the DPU:
   ```
   systemctl status mlx-regex
   ```
   If the status is inactive (`Active: failed`), run:
   ```
   systemctl start mlx-regex
   ```
3. DPI compiler must be used since the RegEx engine accepts only `.cdo` files. The CDO files are constructed by compiling a signature file written in Cta open-source format.

   To compile the signature file, run the following:
   ```
   doca_dpi_compiler -i <path to signature file> -o /tmp/signatures.cdo -f suricata
   ```
   The `.cdo` file is created in the output path flagged as the `-o` input path of the compiler.

5.2. Running the Sample

1. Refer to the following documents:

   ▶ NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

   ▶ NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:
   ```
   cd /opt/mellanox/doca/samples/doca_dpi/<sample_name>
   meson build
   ```

```
ninja -C build
```

> 📝 Note: The binary `doca_<sample_name>` is created under `./build/`.

3. Sample (e.g., `dpi_scan`) usage:

```
Usage: doca_dpi_scan [DPDK Flags] -- [DOCA Flags] [Program Flags]
DOCA Flags:
  -h, --help                    Print a help synopsis
  -v, --version                 Print program version information
  -l, --log-level               Set the log level for the program <CRITICAL=20,
 ERROR=30, WARNING=40, INFO=50, DEBUG=60>

Program Flags:
  -s, --sig-file                Signatures file path
  -a, --pci-addr                DOCA DPI device PCI address
```

4. For additional information per sample, use the `-h` option after the `--` separator:

```
./build/doca_<sample_name> -- -h
```

5. DOCA DPI samples are based on DPDK libraries. Therefore, the user is required to provide DPDK flags. The following is an example from an execution on the DPU:

```
./build/doca_dpi_scan -a auxiliary:mlx5_core.sf.4,sft_en=1 -- -s /tmp/
signatures.cdo -a 03:00.0
```

In contrast to DOCA DPI-based applications, the samples require only one network port. This is because they parse the packets they received without forwarding them.

> 📝 Note: When running on the DPU using the command above sub-functions must be enabled according to the Scalable Function Setup Guide.

> 📝 Note: When running on the host, virtual functions must be used according to the instructions in the NVIDIA DOCA Virtual Functions User Guide.

5.3. Samples

5.3.1. DPI Scan

This sample illustrates how to use DPI programming interface on a single packet.

The sample logic includes:

1. Initializing DOCA DPI.
2. Loading the signature `.cdo` file into the RegEx engine.
3. Creating DPI flow for the incoming packet.
4. Inserting packet into DPI work queue.
5. Polling packet from work queue when processing ends.
6. Retrieving matched signature information if there is a match.
7. Retrieving statistics information.
8. Destroying DPI structures.

Reference:

▶ `/opt/mellanox/doca/samples/doca_dpi/dpi_scan/dpi_scan.c`