P4 Portable NIC Architecture (PNA)

version 0.5

The P4 Language Consortium

2021-05-18

Abstract

P4 is a domain-specific language for describing how packets are processed by a network data plane. A P4 program comprises an architecture, which describes the structure and capabilities of the pipeline, and a user program, which specifies the functionality of the programmable blocks within that pipeline. The Portable NIC Architecture (PNA) is an architecture that describes the structure and common capabilities of network interface controller (NIC) devices that process packets going between one or more interfaces and a host system.

Contents

1. Introduction

Note that this document is still a working draft. Significant changes are expected to be made before version 1.0 of this specification is released.

The Portable NIC Architecture (PNA) is P4 architecture that defines the structure and common capabilities for network interface controller (NIC) devices. PNA comprises two main components:

1. A programmable pipeline that can be used to realize a variety of different "packet paths" going between the various ports on the device (e.g., network interfaces or the host system it is attached to), and

2. A library of types (e.g., intrinsict and standard metadata) and $P4_{16}$ externs (e.g., counters, meters, and registers).

PNA is designed to model the common features of a broad class of NIC devices. By providing standard APIs and coding guidelines, the hope is to enable developers to write programs that are portable across multiple NIC devices that are conformant to the PNA[1].

The Portable NIC Architecture (PNA) Model has four programmable P4 blocks and several fixed-function blocks, as shown in Figure 1. The behavior of the programmable blocks is specified using P4. The network ports, packet queues, and (optional) inline extern blocks are fixed-function blocks that can be configured by the control plane, but are not intended to be programmed using P4.

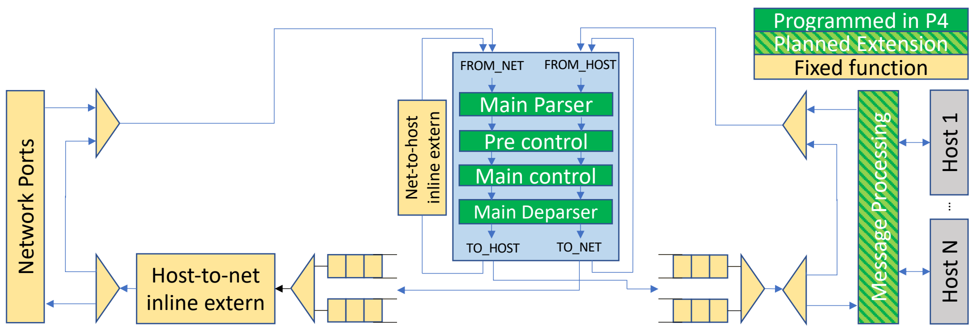

Figure 1. Programmable NIC Architecture Block Diagram

1.1. Packet processing in the network to host direction

Packets arriving from a network port first go through a `MainParser` and a `PreControl`. The `PreControl` can optionally perform table lookups. Its purpose is to determine whether a packet requires processing by the net-to-host inline extern block.

For example, the net-to-host inline extern block may perform decryption of packet payloads according to the IPsec protocol. In this case, the main parser and pre control would be programmed to identify whether the packet was encrypted using IPsec, and if so, what security association it belongs to. For instance, the `PreControl` code might drop the packet if the packet had an IPsec header, but one or more P4 table lookups determined that the packet does not belong to any security association that had been established by the control plane software. Note that the net-to-host inline extern block

[1] Of course, given the tight hardware resource constraints on NIC devices, there is no promise that a given P4 program that compiles on one device will also compile on another device. However, it should at least be the case that those P4 programs that are able to compile on multiple NIC devices should process packets as described in this document.

may modify the entire payload of the received packet—e.g., decrypting the encrypted portion of the payload. Hence, the resulting packet might contain not only the original headers that were parsed by the first invocation of the `MainParser`, but also headers that could not have been parsed as they were previously encrypted. See section B.3 for additional details.

After decryption, the `MainParser` can perform the full parsing required for implementing the desired data plane functionality.

The `MainControl` is typically where the bulk of code would be written for processing packets. It transforms headers, updates stateful elements like counters, meters, and registers, and optionally associates additional user-defined metadata with the packet. The `MainDeparser` serializes the headers back into a packet that can be sent onwards.

After the `MainDeparser`, the packet may either: proceed to the message processing part of the NIC, and then typically on to the host system, or turn around and head back towards the network ports. This enables on-NIC processing of port-to-port packets without ever traversing the host system.

Figure 1 shows multiple hosts. Some NICs support PCI Express connections to multiple host CPU complexes. It is also common for NICs to have an array of one or more CPU cores inside of the NIC device itself, and these can be the target for packets received from the network, and/or the source of packets sent to the network, just as the other hosts can be. For the purposes of the PNA, such CPU cores are considered as another host.

1.2. Message processing

The focus in the current version of this specification is on the four P4-programmable blocks mentioned above. The details of how one can use P4 to program the message processing portion of a NIC is left as a future extension of this specification. While there are options for exactly what packet processing functions can be performed in the four primary blocks described above, versus the message processing block, the division is expected to be:

- The primary programmable blocks deal solely with individual network packets, which are at most one network maximum transmission unit (MTU) in size.
- The message processing block is responsible for converting between large messages in host memory and network size packets on the network, and for dealing with one or more host operating systems, drivers, and/or message descriptor formats in host memory.

For example, in its role of converting between large messages and network packets in the host-to-network direction, message processing would implement features like large send offload (LSO), TCP segmentation offload (TSO), and Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE). In the network-to-host direction it would assist in such features as large receive offload (LRO) and RoCE.

In its role of handling different kinds of operating systems, drivers, and message descriptor formats, the message processing block may deal with one or more of the following standards: - VirtIO - SR-IOV

Another potential criteria for dividing packet processing functionality between message processing and the rest of the NIC is for division of control plane responsibilities. For example, in some network deployments the NIC message processing block configuration is tightly coupled with the host operating system, whereas the `MainControl` is controlled by network-focused control plane software.

1.3. Packet processing in the host to network direction

Messages originating in one of the hosts are segmented into network MTU size packets (if the host has not already done so) in the message processing block, and are then sent to the main block for further processing.

The same `MainParser`, `PreControl`, `MainControl`, and `MainDeparser` that process packets from the network are also used to process packets from the host. PNA was designed this way for two reasons:

- It is expected that in many cases, the packet processing in both directions will have many similarities between them. Writing common P4 code for both eliminates code duplication that would occur if the code for each direction was written separately.
- Having a single `MainControl` in the P4 language enables tables and externs such as counters and registers to be instantiated once, and shared by packets being processed in both directions. The hardware of many NICs supports this design, without having to instantiate a physically separate table for each direction. Especially for large tables used by packet processing in both directions, this approach can significantly reduce the memory required. It is also critical for some stateful features (e.g. those using the table add-on-miss capability) to access the same table in memory when processing packets in either direction.

After finishing processing in the `MainControl`, the packet may be enqueued in one of several queues (the number of such queues is target specific). After queueing there may be a host-to-net inline extern. For example, the host-to-net inline extern block may perform encryption of packet payloads according to the IPsec protocol. In this case, the `MainControl` would indicate that this processing should be form via assigning appropriate values to standard metadata fields created for this purpose.

Next, the two primary choices for the next place the packet will go are:

- proceed to be emitted out of one of the network ports, or
- turn around and head back towards the host system, which enables on-NIC processing of VM-to-VM or host-to-host packets (i.e., on a system with multiple hosts).

The choices of which queue to use, what kind of processing to perform in the host-to-net inline extern, which network port to go to, or whether to loop back, are all controlled from the P4 code running in the `MainControl` block, via extern functions defined by this PNA specification.

Note that packets processed in the main block cannot "change direction" internally. That is, packets from the network must go out the to-host path, and packets from the host must go out the to-net path. There are loopback paths outside of the main block as shown in Figure 1.

1.4. PNA P4$_{16}$ architecture

A programmer targeting the PNA is required to provide P4 definitions for each of the programmable blocks in the pipeline (see section 5). The programmable block inputs and outputs are parameterized on the types of user defined headers and metadata. The top-level PNA program instantiates a package named `main` with the programmable blocks passed as arguments (see Section TBD for an example). Note that the `main` package is not to be confused with the `MainControl`.

A P4 programmer wishing to maximize the portability of their program should follow several general guidelines:

- Do not use undefined values in a way that affects the resulting output packet(s), or for side effects such as updating `Counter`, `Meter` or `Register` instances.

- Use as few resources as possible, e.g. table search key bits, array sizes, quantity of metadata associated with packets, etc.

This document contains excerpts of several $P4_{16}$ programs that use the `pna.p4` include file and demonstrate features of PNA. Source code for the complete programs can be found in the official repository containing the PNA specification[2].

2. Naming conventions

In this document we use the following naming conventions:

- Types are named using CamelCase followed by `_t`. For example, `PortId_t`.
- Control types and extern object types are named using CamelCase. For example `IngressParser`.
- Struct types are named using lower case words separated by _ followed by `_t`. For example `pna_ingress_input_meta`
- Actions, extern methods, extern functions, headers, structs, and instances of controls and externs start with lower case and words are separated using _. For example `send_to_port`.
- Enum members, const definitions, and #define constants are all caps, with words separated by _. For example `PNA_PORT_CPU`.

Architecture specific metadata (e.g. structs) are prefixed by `pna_`.

3. Packet paths

Figure 2 shows all possible paths for packets that must be supported by a PNA implementation. An implementation is allowed to support paths for packets that are not described here.
TBD: Create another figure with names for the paths.
Table 1 shows what can happen to a packet as a result of a single time being processed through the four programmable blocks of the packet processing part of PNA, referred to here as "main".

Note that each mirrored packet that results from `mirror_packet` operations will have its own next place that it will go to be processed, independent of the original packet, and independent of any other mirror copies made of the same original packet.

4. PNA Data types

4.1. PNA type definitions

Each PNA implementation will have specific bit widths in the data plane for the types shown in the code excerpt of Section 4.1.1. These widths are defined in the target specific `pna.p4` include file. They are expected to differ from one PNA implementation to another[3].

For each of these types, the P4 Runtime API[4] may use bit widths independent of the targets. These widths are defined by the P4 Runtime API specification, and they are expected to be at least as large as

[2]`https://github.com/p4lang/pna` in directory `examples`. Direct link: `https://github.com/p4lang/pna/tree/master/examples`

[3]It is expected that `pna.p4` include files for different targets will be nearly identical to each other. Besides the possibility of differing bit widths for these PNA types, the only expected differences between `pna.p4` files for different targets would be annotations on externs, etc. that the P4 compiler for that target needs to do its job.

[4]The P4Runtime Specification can be found here: `https://p4.org/specs`

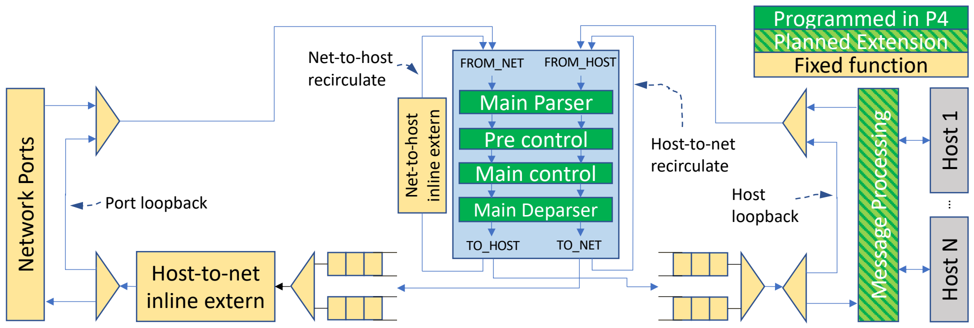

Figure 2. Packet Paths in PNA

the corresponding `InHeader_t` type below, such that they hold a value for any target. All PNA implementations must use data plane sizes for these types no wider than the corresponding `InHeader_t`-defined types.

4.1.1. PNA type definition code excerpt

```
/* These are defined using `typedef`, not `type`, so they are truly
 * just different names for the type bit<W> for the particular width W
 * shown.  Unlike the `type` definitions below, values declared with
 * the `typedef` type names can be freely mingled in expressions, just
 * as any value declared with type bit<W> can.  Values declared with
 * one of the `type` names below _cannot_ be so freely mingled, unless
 * you first cast them to the corresponding `typedef` type.  While
 * that may be inconvenient when you need to do arithmetic on such
 * values, it is the price to pay for having all occurrences of values
 * of the `type` types marked as such in the automatically generated
 * control plane API.
 *
 * Note that the width of typedef <name>Uint_t will always be the same
 * as the width of type <name>_t. */
typedef bit<unspecified> PortIdUint_t;
typedef bit<unspecified> InterfaceIdUint_t;
```

Description	Processed next by	Resulting packet(s)
packet from network port	main, with direction `NET_TO_HOST`	Zero or more mirrored packets, plus at most one of: a net-to-host recirculated packet, or one to-host packet.
packet from net-to-host recirculate		
packet from port loopback		
packet from message processing	main, with direction `HOST_TO_NET`	Zero or more mirrored packets, plus at most one of: a host-to-net recirculated packet, or one to-net packet.
packet from host-to-net recirculate		
packet from host loopback		

Table 1. Result of packet processed one time by main.

```
typedef bit<unspecified> MulticastGroupUint_t;
typedef bit<unspecified> MirrorSessionIdUint_t;
typedef bit<unspecified> MirrorSlotIdUint_t;
typedef bit<unspecified> ClassOfServiceUint_t;
typedef bit<unspecified> PacketLengthUint_t;
typedef bit<unspecified> MulticastInstanceUint_t;
typedef bit<unspecified> TimestampUint_t;
typedef bit<unspecified> FlowIdUint_t;
typedef bit<unspecified> ExpireTimeProfileIdUint_t;
typedef bit<unspecified> PassNumberUint_t;

typedef bit<unspecified> SecurityAssocIdUint_t;

@p4runtime_translation("p4.org/pna/v1/PortId_t", 32)
type PortIdUint_t         PortId_t;
@p4runtime_translation("p4.org/pna/v1/InterfaceId_t", 32)
type InterfaceIdUint_t     InterfaceId_t;
@p4runtime_translation("p4.org/pna/v1/MulticastGroup_t", 32)
type MulticastGroupUint_t MulticastGroup_t;
@p4runtime_translation("p4.org/pna/v1/MirrorSessionId_t", 16)
type MirrorSessionIdUint_t MirrorSessionId_t;
@p4runtime_translation("p4.org/pna/v1/MirrorSlotId_t", 8)
type MirrorSlotIdUint_t MirrorSlotId_t;
@p4runtime_translation("p4.org/pna/v1/ClassOfService_t", 8)
type ClassOfServiceUint_t ClassOfService_t;
@p4runtime_translation("p4.org/pna/v1/PacketLength_t", 16)
```

```
type PacketLengthUint_t   PacketLength_t;
@p4runtime_translation("p4.org/pna/v1/MulticastInstance_t", 16)
type MulticastInstanceUint_t MulticastInstance_t;
@p4runtime_translation("p4.org/pna/v1/Timestamp_t", 64)
type TimestampUint_t      Timestamp_t;
@p4runtime_translation("p4.org/pna/v1/FlowId_t", 32)
type FlowIdUint_t      FlowId_t;
@p4runtime_translation("p4.org/pna/v1/ExpireTimeProfileId_t", 8)
type ExpireTimeProfileIdUint_t      ExpireTimeProfileId_t;
@p4runtime_translation("p4.org/pna/v1/PassNumber_t", 8)
type PassNumberUint_t      PassNumber_t;

@p4runtime_translation("p4.org/pna/v1/SecurityAssocId_t", 64)
type SecurityAssocIdUint_t      SecurityAssocId_t;

typedef error   ParserError_t;

const InterfaceId_t PNA_PORT_CPU = (PortId_t) unspecified;

const MirrorSessionId_t PNA_MIRROR_SESSION_TO_CPU = (MirrorSessiontId_t) unspecified;
```

4.1.2. PNA port types and values

There are two types defined by PNA for holding different kinds of ports: `PortId_t` and `InterfaceId_t`.

The type `PortId_t` must be large enough in the data plane to hold one of these values:

- a data plane id for one network port
- a data plane id for one vport

As one example, a PNA target with four Ethernet network ports could choose to use the values 0 through 3 to identify the network ports, and the values 4 through 1023 to identify vports.

PNA makes no requirement that the numeric values identifying network ports must be consecutive, nor for vports. PNA only requires that for every possible numeric value `x` with type `PortId_t`, exactly one of these statements is true:

- `x` is the data plane id of one network port, but not any vport
- `x` is the data plane id of one vport, but not any network port
- `x` is the data plane id of no port, neither a network port nor a vport

4.2. PNA supported metadata types

```
enum PNA_PacketPath_t {
    // TBD if this type remains, whether it should be an enum or
    // several separate fields representing the same cases in a
    // different form.
    FROM_NET_PORT,
    FROM_NET_LOOPEDBACK,
    FROM_NET_RECIRCULATED,
    FROM_HOST,
    FROM_HOST_LOOPEDBACK,
    FROM_HOST_RECIRCULATED
}

struct pna_pre_input_metadata_t {
    PortId_t                 input_port;
    ParserError_t            parser_error;
    PNA_Direction_t          direction;
    PassNumber_t             pass;
    bool                     loopedback;
}

struct pna_pre_output_metadata_t {
    bool                     decrypt;  // TBD: or use said==0 to mean no decrypt?

    // The following things are stored internally within the decrypt
    // block, in a table indexed by said:

    // + The decryption algorithm, e.g. AES256, etc.
    // + The decryption key
    // + Any read-modify-write state in the data plane used to
    //   implement anti-replay attack detection.

    SecurityAssocId_t        said;
    bit<16>                  decrypt_start_offset;  // in bytes?

    // TBD whether it is important to explicitly pass information to a
    // decryption extern in a way visible to a P4 program about where
    // headers were parsed and found.  An alternative is to assume
    // that the architecture saves the pre parser results somewhere,
    // in a way not visible to the P4 program.
}

struct pna_main_parser_input_metadata_t {
    // common fields initialized for all packets that are input to main
    // parser, regardless of direction.
    PNA_Direction_t          direction;
    PassNumber_t             pass;
    bool                     loopedback;
    // If this packet has direction NET_TO_HOST, input_port contains
    // the id of the network port on which the packet arrived.
    // If this packet has direction HOST_TO_NET, input_port contains
    // the id of the vport from which the packet came
    PortId_t                 input_port;   // network port id
}

struct pna_main_input_metadata_t {
    // common fields initialized for all packets that are input to main
    // parser, regardless of direction.
    PNA_Direction_t          direction;
    PassNumber_t             pass;
    bool                     loopedback;
    Timestamp_t              timestamp;
    ParserError_t            parser_error;
    ClassOfService_t         class_of_service;
    // See comments for field input_port in struct
    // pna_main_parser_input_metadata_t
    PortId_t                 input_port;
}


struct pna_main_output_metadata_t {
  // common fields used by the architecture to decide what to do with
  // the packet next, after the main parser, control, and deparser
  // have finished executing one pass, regardless of the direction.
  ClassOfService_t         class_of_service; // 0
}
```

4.3. Match kinds

TBD: Consider simply referencing the corresponding section of the PSA specification for this, unless we want to have something different in PNA.

4.4. Data plane vs. control plane data representations

5. Programmable blocks

The following declarations provide a template for the programmable blocks in the PNA. The P4 programmer is responsible for implementing controls that match these interfaces and instantiate them in a package definition.

It uses the same user-defined metadata type `IM` and header type `IH` for all ingress parsers and control blocks. The egress parser and control blocks can use the same types for those things, or different types, as the P4 program author wishes.

```
control PreControlT<PH, PM>(
    in    PH pre_hdr,
    inout PM pre_user_meta,
    in    pna_pre_input_metadata_t  istd,
    inout pna_pre_output_metadata_t ostd);

parser MainParserT<PM, MH, MM>(
    packet_in pkt,
    //in    PM pre_user_meta,
    out   MH main_hdr,
    inout MM main_user_meta,
    in    pna_main_parser_input_metadata_t istd);

control MainControlT<PM, MH, MM>(
    //in    PM pre_user_meta,
    inout MH main_hdr,
    inout MM main_user_meta,
    in    pna_main_input_metadata_t  istd,
    inout pna_main_output_metadata_t ostd);

control MainDeparserT<MH, MM>(
    packet_out pkt,
    in    MH main_hdr,
    in    MM main_user_meta,
    in    pna_main_output_metadata_t ostd);

package PNA_NIC<PH, PM, MH, MM>(
    MainParserT<PM, MH, MM> main_parser,
    PreControlT<PH, PM> pre_control,
    MainControlT<PM, MH, MM> main_control,
    MainDeparserT<MH, MM> main_deparser);
```

6. Packet Path Details

Refer to section 3 for the packet paths provided by PNA.
TBD: Need to decide where multicast replication can occur, and in what conditions.
TBD: Need to decide where packet mirroring occurs, and in what conditions, and how the mirrored packets differ from the originals.

6.1. Initial values of packets processed by main parser

6.1.1. Initial packet contents for packets from ports

Packet is as received from Ethernet port.
User-defined metadata is empty?

6.1.2. Initial packet contents for packets looped back from host-to-network path

Packet is as came out of host-to-net received from Ethernet port.

There can be user-defined metadata included with these packets.

6.2. Behavior of packets after pre block is complete

Cases: drop vs. not, do something in net-to-host inline extern block or not.

6.3. Initial values of packets processed in network-to-host direction by main block

6.3.1. Initial packet contents for normal packets

The packet should be either: + exactly as arrived at the pre parser, if the net-to-host inline extern was directed not to modify the packet + exact as output by the net-to-host inline extern

The user-defined metadata should be exactly as output by the pre control.

The standard metadata contents should be specified in detail here.

6.3.2. Initial packet contents for recirculated packets

Give any differences between this case and previous section.

6.4. Behavior of packets after main block is complete in network-to-host direction

Cases: drop, recirculate, loopback to host-to-net direction, to message processing. Describe the conditions in which each occurs.

6.5. Initial values of packets processed in host-to-network direction by main block

6.5.1. Initial packet contents for normal packets

This is for packets from the message processing block.

6.5.2. Initial packet contents for recirculated packets

Give any differences between this case and previous section.

6.5.3. Initial packet contents for packets looped back after network-to-host main processing

6.6. Behavior of packets after main block is complete in host-to-network direction

Cases: drop, recirculate, to queues. Describe the conditions in which each occurs.

6.7. Contents of packets sent out to ports

6.8. Functions for directing packets

6.8.1. Extern function send_to_port

```
extern void send_to_port(PortId_t dest_port);
```

The extern function `send_to_port` is used to direct a packet to a specified network port, or to a vport. Invoking `send_to_port(x)` is supported only within the main control. There are four cases to consider, detailed below.

- current packet direction is `HOST_TO_NET`, and `x` is a network port id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that the packet will be transmitted out of the network port with id `x`, without being looped back.

- current packet direction is `NET_TO_HOST`, and `x` is a network port id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that when the packet is finished with the main control and main deparser, it will loop back in the host side, and later return to be processed by the main control in the HOST_TO_NET direction. The hidden state will remain associated with the packet during that loopback, so that even if no further forwarding functions are called for the packet, it will be transmitted out of network port `x`.

- current packet direction is `HOST_TO_NET`, and `x` is a vport id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that when the packet is finished with the main control and main deparser, it will take the port loopback path, and later return to be processed by the main control in the NET_TO_HOST direction. The hidden state will remain associated with the packet during that loopback, so that even if no further forwarding functions are called for the packet, it will be sent to the vport with id `x` in the host.

- current packet direction is `NET_TO_HOST`, and `x` is a vport id.

Calling `send_to_port(x)` modifies hidden state for this packet, so that the packet will be sent to the vport with id `x` in the host, without being looped back.

6.9. Packet Mirroring

```
extern void mirror_packet(MirrorSlotId_t mirror_slot_id,
                          MirrorSessionId_t mirror_session_id);
```

The extern function `mirror_packet` is used to cause a mirror copy of the packet currently being processed to be created. Invoking `mirror_packet(x)` is supported only within the main control.

PNA enables multiple mirror copies of a packet to be created during a single execution of `MainControl`, by calling `mirror_packet` with different mirror slot id values. PNA targets should support `mirror_slot_id` values in the range 0 through 3, at least, but are allowed to support a larger range.

When `MainControl` begins execution, all mirror slots are initialized so that they do not create a copy of the packet.

After calling `mirror_packet(slot_id, session_id)`, then when the main control finishes execution, the target will make a best effort to create a copy of the packet that will be processed according to the parameters configured by the control plane for the mirror session numbered `session_id`, for mirror slot `slot_id`. Note that this is best effort – if the target device is already near its upper limit of its ability to create mirror copies, then some later mirror copies may not be made, even though the P4 program requested them.

Each of the mirror slots is independent of each other. For example, calling `mirror_packet(1, session_id)` has no effect on mirror slots 0, 2, or 3.

Mirror session id 0 is reserved by the architecture, and must not be used by a P4 developer.

If multiple calls are made to `mirror_packet()` for the same mirror slot id in the same execution of the main control, only the last `session_id` value is used to create a copy of the packet. That is, every call to `mirror_packet(slot_id, session_id)` overwrites the effects of any earlier to `mirror_packet()` with the same `slot_id`.

The effects of `mirror_packet()` calls are independent of calls to `drop_packet()` and `send_to_port()`. Regardless of which of those things is done to the original packet, up to one mirror packet per mirror slot can be created.

The control plane code can configure the following properties of each mirror session, independently of other mirror sessions:

- `packet_contents`

If `PRE_MODIFY`, then the mirrored packet's contents will be the same as the original packet as it was when the packet began the execution of the main control that invoked the `mirror_packet()` function.

If `POST_MODIFY`, then the mirrored packet's contents will be the same as the original packet that is being mirrored, after any modifications made during the execution of the main control that invoked the `mirror_packet()` function.

- `truncate`

`true` to limit the length of the mirrored packet to the `truncate_length`. `false` to cause the mirrored packet not to be truncated, in which case the `truncate_length` property is ignored for this mirror session.

- `truncate_length`

In units of bytes. Targets may limit the choices here, e.g. to a multiple of 32 bytes, or perhaps even a subset of those choices.

- `sampling_method`

One of the values: `RANDOM_SAMPLING`, `HASH_SAMPLING`.

If `RANDOM_SAMPLING`, then a mirror copy requested for this mirror session will only be created with a configured probability given by the `sample_probability` property.

If `HASH_SAMPLING`, then a target-specific hash function will be calculated over the packet's header fields resulting in a hash output value `H`. A mirror copy will be created if `(H & sample_hash_mask) == sample_hash_value`.

- `meter_parameters`

If the conditions specified by the `sampling_method` and other sampling properties are passed, then a P4 meter dedicated for use by this mirror session will be updated. If it returns a `GREEN` result, then the mirror copy will be created (still with best effort, if the target device's implementation is still oversubscribed with requests to create mirror copies).

If the meter update returns any result other than `GREEN`, then no mirror copy will be created.

- `destination_port`

A network port id, or a vport id.

If `destination_port` is a network port id, that network port is the destination of mirrored copy packets created by this session. If the `mirror_packet()` call for this session was invoked in the `NET_TO_HOST` direction, mirror copy packets created will loop back in the host side of the target, and later come back for processing in the main block in the `HOST_TO_NET` direction, already destined for the network port `destination_port`. That port can be overwritten by calls to forwarding functions.

If `destination_port` is a vport id, that vport is the destination of mirrored copy packets created by this session. If the `mirror_packet()` call for this session was invoked in the `HOST_TO_NET` direction, mirror copy packets created will loop back in the network port side of the NIC, and later come back for processing in the main block in the `NET_TO_HOST` direction, already destined for the vport `destination_port`. That vport can be overwritten by calls to forwarding functions.

TBD: When a mirror copied packet comes back to the main control, it will have some metadata indicating it is mirror copy. We should define a way in PNA to recognize such mirror copies, e.g. some new extern function call returning true if the packet was created by a `mirror_packet` operation.

6.10. Packet recirculation

7. PNA Extern Objects

7.1. Restrictions on where externs may be used

All instantiations in a $P4_{16}$ program occur at compile time, and can be arranged in a tree structure we will call the instantiation tree. The root of the tree `T` represents the top level of the program. Its child is the node for the package `PNA_NIC` described in Section 5, and any externs instantiated at the top level of the program. The children of the `PNA_NIC` node are the packages and externs passed as parameters to the `PNA_NIC` instantiation. See Figure 3 for a drawing of the smallest instantiation tree possible for a P4 program written for PNA.

Figure 3. Minimal PNA instantiation tree

If any of those parsers or controls instantiate other parsers, controls, and/or externs, the instantiation tree contains child nodes for them, continuing until the instantiation tree is complete.

For every instance whose node is a descendant of the `Ingress` node in this tree, call it an `Ingress` instance. Similarly for the other ingress and egress parsers and controls. All other instances are top level instances.

A PNA implementation is allowed to reject programs that instantiate externs, or attempt to call their methods, from anywhere other than the places mentioned in Table 2.

For example, `Counter` being restricted to “Pre, Main” means that every `Counter` instance must be instantiated within either the `PreControl` control block or the `MainControl` block, or be a descendant of one of those nodes in the instantiation tree. If a `Counter` instance is instantiated in Main, for example, then it cannot be referenced, and thus its methods cannot be called, from any block except `MainControl` or one of its descendants in the tree.

Extern type	Where it may be instantiated and called from
`ActionProfile`	PreControl, MainControl
`ActionSelector`	PreControl, MainControl
`Checksum`	MainParser, MainDeparser
`Counter`	PreControl, MainControl
`Digest`	MainDeparser
`DirectCounter`	PreControl, MainControl
`DirectMeter`	PreControl, MainControl
`Hash`	PreControl, MainControl
`InternetChecksum`	MainParser, MainDeparser
`Meter`	PreControl, MainControl
`Random`	PreControl, MainControl
`Register`	PreControl, MainControl

Table 2. Summary of controls that can instantiate and invoke externs.

PNA implementations need not support instantiating these externs at the top level. PNA implementations are allowed to accept programs that use these externs in other places, but they need not. Thus P4 programmers wishing to maximize the portability of their programs should restrict their use of these externs to the places indicated in the table.

All methods for type `packet_out`, e.g., `emit`, are restricted to be within deparser control blocks in PNA, because those are the only places where an instance of type `packet_out` is visible. Similarly all methods for type `packet_in`, e.g. `extract` and `advance`, are restricted to be within parsers in PNA programs. $P4_{16}$ restricts all `verify` method calls to be within parsers for all $P4_{16}$ programs, regardless of whether they are for the PNA.

See the PSA specification for definitions of all of these externs. There is work under way as of this writing that may result in these extern defintions being moved from the PSA specification into a separate standard library of P4 extern definitions, and if this is done, both the PSA and PNA specifications will reference that.

8. PNA Table Properties

Table 3 lists all P4 table properties defined by PNA that are not included in the base $P4_{16}$ language specification.

A PNA implementation need not support both of a `pna_implementation` and `pna_direct_counter` property on the same table.

Similarly, a PNA implementation need not support both of a `pna_implementation` and `pna_direct_meter` property on the same table.

A PNA implementation must implement tables that have both a `pna_direct_counter` and `pna_direct_meter` property.

A PNA implementation need not support both `pna_implementation` and `pna_idle_timeout` properties on the same table.

Property name	Type	See also
add_on_miss	boolean	Section 8.1
pna_direct_counter	one DirectCounter instance name	
pna_direct_meter	one DirectMeter instance name	
pna_implementation	instance name of one ActionProfile or ActionSelector	
pna_empty_group_action	action	
pna_idle_timeout	PNA_IdleTimeout_t	Section 8.2

Table 3. Summary of PNA table properties.

8.1. Tables with add-on-miss capability

PNA defines the `add_on_miss` table property. If the value of this property is `true` for a table, the P4 developer is allowed to define a default action for the table that calls the `add_entry` extern function. `add_entry` adds a new entry to the table whose default action calls the `add_entry` function. The new entry will have the same key that was just looked up.

The control plane API is still allowed to add, modify, and delete entries of such a table, but any entries added via the `add_entry` function do not require the control plane software to be involved in any way. It is expected that PNA implementations will be able to sustain `add_entry` calls at a large fraction of their line rate, but it need not be at the same packet rate supported for processing packets that do not call `add_entry`. The new table entry will be matchable when the next packet is processed that applies this table.

```
extern bool add_entry<T>(string action_name,
                         in T action_params);
```

It is expected that many PNA implementations will restrict `add_entry()` to be called with the following restrictions:

- Only from within an action
- Only if the action is a default action of a table with property `add_on_miss` equal to `true`.
- Only for a table with all key fields having match_kind `exact`.
- Only with an action name that is one of the hit actions of that same table. This action has parameters that are all directionless.
- The type `T` is a struct containing one member for each directionless parameter of the hit action to be added. The member names must match the hit action parameter names, and their types must be the same as the corresponding hit action parameters.

The new entry will have the same key field values that were searched for in the table when the miss occurred, which caused the table's default action to be executed. The action will be the one named by the string that is the file of the parameter action_name.

If the attempt to add a table entry succeeds, the return value is `true`, otherwise `false`.

8.2. Table entry timeout notification

PNA uses the `pna_idle_timeout` to enable a table implementation send notifications from the PNA device when a configurable time has passed since an entry was last matched. The property may take one of two values – `NO_TIMEOUT`, and `NOTIFY_CONTROL`. `NO_TIMEOUT` disables idle timeout support for the table and it is the default value when the property is not present. `NOTIFY_CONTROL` enables the notification. A PNA implementation will then generate an API for the control plane to set time-to-live (TTL) values for table entries and if at any time during its lifetime, the table entry is not "hit" (i.e. not selected by any packet lookup) for a lapse of time greater or equal to its TTL, the device should generate a notification to the control plane. The rate and mode of how the notifications are generated and delivered to the control plane are subject to configuration parameters specified by the control plane API.

Example:

```
enum PNA_IdleTimeout_t {
  NO_TIMEOUT,
  NOTIFY_CONTROL
}

table t {
  action a1 () { ... }
  action a2 () { ... }
  key = { hdr.f1: exact; }
  actions = { a1; a2; }
  default_action = a2;
  pna_idle_timeout = PNA_IdleTimeout_t.NOTIFY_CONTROL;
}
```

Restrictions on the TTL values and notifications:

- It is likely that any hardware implementation will have a limited number of bits to represent the values, and, since the values are programmed at runtime, it is the responsibility of the runtime (P4Runtime or other controller software) to guarantee that the TTL values can be represented in the device. This can be done by scaling the values to the number of bits available on the platform, ensuring that the range of values between different entries are representable. A PNA implementation should only enable the programming of such tables, and return an error if the device does not support the idle timeout at all.
- If no value is programmed for a table entry, even though the table has enabled the idle timeout property, the entry will not generate a notification.
- PNA does not require a timeout value for a default action entry. The reason for not making this mandatory in the specification is that tthe default action may not have an explicit table entry to represent it, and also there are no known compelling use cases for a controller knowing when no misses have occurred for a particular table for a long time. The default action entry will not be aged out.
- Currently, tables implemented using ActionSelectors and ActionProfiles do not support the `pna_idle_timeout` property. Future versions of the specification may remove this restriction.

9. Timestamps

10. Atomicity of control plane API operations

A. Appendix: Open Issues

B. Appendix: Rationale for design

B.1. Why a common pipeline, instead of separate pipelines for each direction?

TBD: Andy can write this one. Basic reasons are summarized in existing slides.

B.2. Why separate programmable pre blocks for pre-decryption packet processing?

TBD: Andy can write this one. Basic reasons are summarized in existing slides.

B.3. Is it inefficient to have the `MainParser` redo work?

If the only changes made by the inline extern in the network-to-host direction were to decrypt parts of the packet that were previously encrypted, but everything before the first decrypted byte remained exactly the same, then it seems like it is a waste of effort that the main parser starts parsing the packet over again from the beginning.

It is true that an IPsec decryption inline extern is unlikely to change an Ethernet header at the beginning of the packet, but it does seem likely that it could make the following kinds of changes to parts of the packet before the first decrypted byte:

- Remove headers: If the received packet was IPsec tunnel mode, it might be useful if the inline extern removes the outer IP header, since it was added to the packet at the point of IPsec encryption. The software sending the packet (before IPsec encryption occurred) did not create that header, and the corresponding layer of software receiving the decrypted packet does not want to see such IPsec-specific headers.
- Modify headers: If the received packet was IPsec transport mode, it might be useful if the IP header whose protocol was equal to the standard numbers for AH or ESP was changed to be the next header after the AH and ESP headers are removed by the inline extern. Again, what an IPsec decryption block does might be useful to make similar to what the IPsec layer of software does in a software IP stack. The layer of software processing the decrypted packet should see what the last layer of software sent before it was encrypted.

If any or all of the above are true of the inline extern block's changes to the packet, then it seems that the only way you could save the main parser some work is to somehow encode the results of the pre parser, and also undo those results for any headers that were modified in the inline extern. Then you would also need the main parser to be able to start from one of multiple possible states in the parser state machine, and continue from there.

That is all possible to do, but it seems like an awkward thing to expose to a P4 developer, e.g. should we require them to write a main parser that has a start state that immediately branches one of 7 ways based upon some intermediate state the the pre parser reached, as modified by the inline extern if it modified or removed some of those headers?

A NIC implementation might do such things, and it seems likely an implementation might use some of the techniques mentioned in the previous paragraph, but hidden from the P4 developer. The proposed PNA design should not prevent this, if an implementer is willing to go to that effort.

C. Appendix: Packet path figures

C.1. Network to host

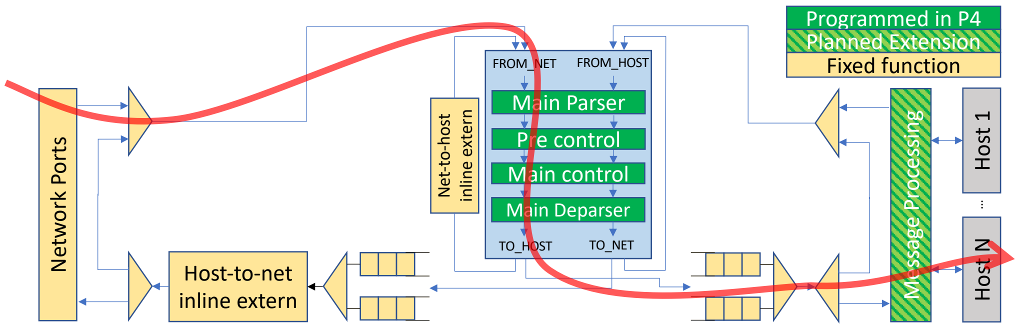

Figure 4. Network to host packet path

See Figure 4. If decryption is desired, the net-to-host inline extern performs it. If no decryption is required, the net-to-host inline extern outputs the same packet payload that it received, i.e. it is a no-op in the path.

C.2. Network to host with mirror copy to different host

See Figure 5. This is similar to the network to host path, except that the `MainControl` code directs that the packet should be mirrored to a second host, e.g. an inside-the-NIC CPU complex used for exception packets. Logically, the copy occurs after the main deparser.

One possibility for writing P4 code to do this is by having the `MainDeparser` optionally invoke a mirror extern, which could provide an extra header to include before the mirrored copy.

C.3. Host to network

See Figure 6. If encryption is desired, the host-to-net inline extern performs it. If no encryption is required, the host-to-net inline extern outputs the same packet payload that it received, i.e. it is a no-

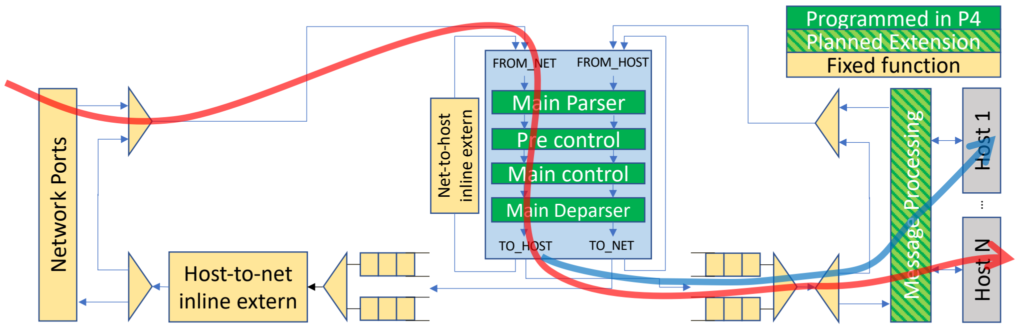

Figure 5. Network to host packet path, with mirror copy to second host

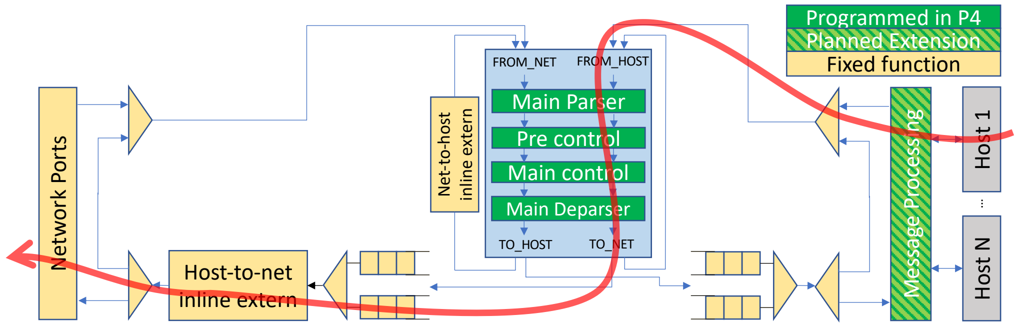

Figure 6. Host to network packet path

op in the path.

C.4. Host to network with mirror copy to a different host

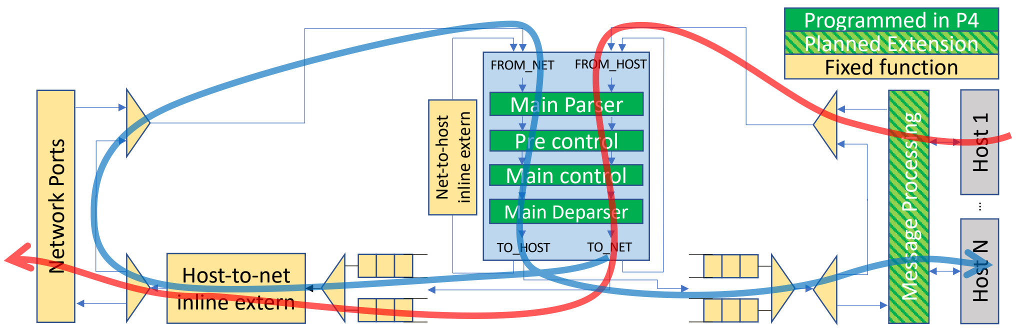

Figure 7. Host to network packet path, with mirror copy to a different host

See Figure 7. This is similar to the host to network path, except that the `MainControl` code directs that the packet should be mirrored to a host, e.g. an inside-the-NIC CPU complex used for exception packets. Logically, the copy occurs after the main deparser.

One possibility for writing P4 code to do this is by having the `MainDeparser` optionally invoke a mirror or mirror extern, which could provide an extra header to include before the mirrored copy.

C.5. Host to host

See Figure 8. This path may more often be called the VM to VM path.

The host-to-net inline extern may be no-op or perform encryption, as directed by the P4 code in the `MainControl`. The net-to-host inline extern may be no-op or perform decryption, as directed by the P4 code in the `PreControl`.

C.6. Port to port

See Figure 9. This path is shown going through the memory of one of the hosts. The host could be a CPU core complex within the NIC device itself, with its own memory, or it could be a host CPU complex and its DRAM.

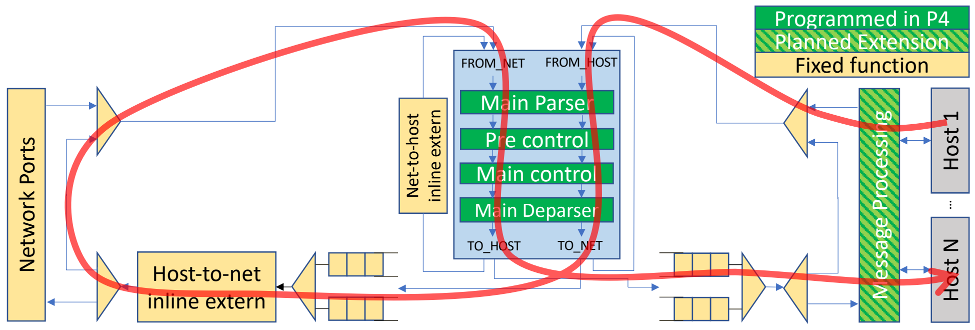

Figure 8. Host to host packet path

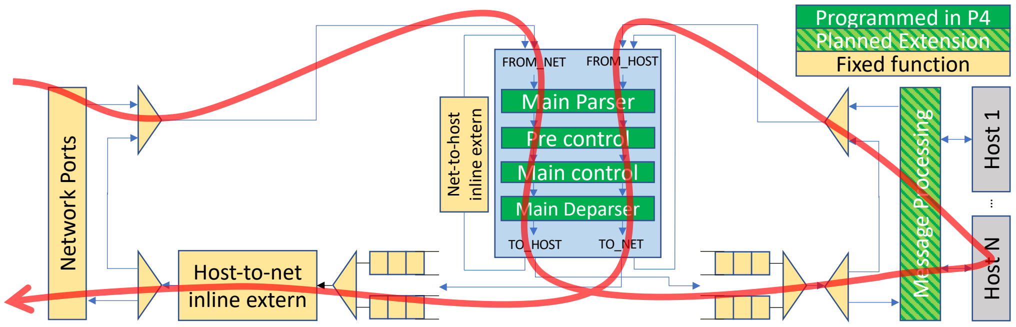

Figure 9. Port to port packet path

D. Appendix: Packet ordering

E. Appendix: Revision History

Release	Release Date	Summary of Changes
0.1	November 5, 2020	Skeleton specification.
0.5	May 15, 2021	Initial draft.