$P4_{16}$ Portable NIC Architecture (PNA)

(working draft)

The P4.org Architecture Working Group

2021-05-15

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

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].

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.

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.

1.1. Packet processing in the network to host direction

Packets arriving from a network port first go through a "main" parser and a "pre" control. The pre control 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 pre control code might drop the packet if the packet had an IPsec

[1] Of course, given the tight hardware resource

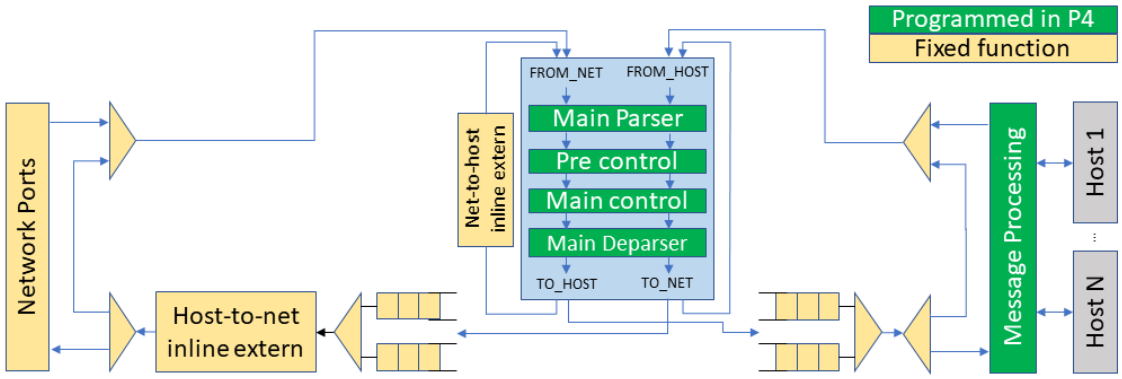

Figure 1. Programmable NIC Architecture Block Diagram

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 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 pre parser, but additional headers that could not have been parsed by the pre parser as they were previously encrypted. See section B.3 for additional details.

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

The main control 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 main deparser P4 code serializes the headers back into a packet that can be sent onwards.

After the main deparser, 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 main blocks are 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 main parser, pre control, main control, and deparser 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 main control 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 main control, 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 main control would indicate that this processing should be form via assigning appropriate values to standard metadata fields created for this purpose.

Next, the two main 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 main 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 the `main` `package` object, with the programmable blocks passed as arguments (see Section TBD for an example).

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_metadata_t`.
- 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.

4. PNA Data types

4.1. PNA type definitions

Each PNA implementation will have specific bit widths for the following types in the data plane. 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[^P4RuntimeAPI] 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 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.

[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 typically 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.

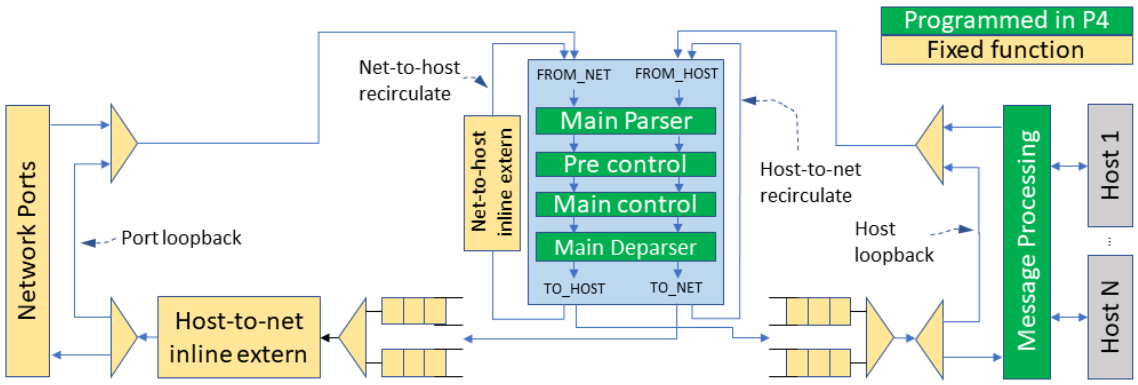

Figure 2. Packet Paths in PNA

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_parser_input_metadata_t {
    PortId_t                input_port;
    PNA_PacketPath_t        packet_path;
}

struct pna_pre_input_metadata_t {
    PortId_t                input_port;
    PNA_PacketPath_t        packet_path;
    ParserError_t           parser_error;
}

struct pna_pre_output_metadata_t {
}

struct pna_decrypt_input_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;
  PNA_PacketPath_t         packet_path;

  // input fields to main parser that are only initialized if
  // direction == NET_TO_HOST
  PortId_t                 input_port;

  // input fields to main parser that are only initialized if
  // direction == HOST_TO_NET
  VportId_t                input_vport;
}

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;
  PNA_PacketPath_t         packet_path;
  Timestamp_t              timestamp;
  ParserError_t            parser_error;
  ClassOfService_t         class_of_service;

  // input fields to main control that are only initialized if
  // direction == NET_TO_HOST
  PortId_t                 input_port;

  // input fields to main control that are only initialized if
  // direction == HOST_TO_NET
  VportId_t                input_vport;
}


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.

```
parser PreParserT<PH, PM>(
    packet_in pkt,
    out PH pre_hdr,
    out PM pre_user_meta,
    in  pna_pre_parser_input_metadata_t istd);

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>(
    PreControlT<PH, PM> pre_control,
```

```
    MainParserT<PM, MH, MM> main_parser,
    MainControlT<PM, MH, MM> main_control,
    MainDeparserT<MH, MM> main_deparser,
    // pre_parser is optional.  If not specified, it defaults to be the
    // same as main_parser.  An implementation that has a separate pre
    // parser is free to optimize away any part of it that is
    // unnecessary for executing the code in pre_control.
    PreParserT<PH, PM> pre_parser);
```

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 pre block

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

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 1.

For example, `Counter` being restricted to "Pre, Main" means that every `Counter` instance must be instantiated within either the `Pre` control block or the `Main` control 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 control block except `Main` or one of its descendants in the tree.

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

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

Table 1. Summary of controls that can instantiate and invoke externs.

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.

TBD: For the descriptions of each of the externs, simply reference the corresponding section of the PSA specification if they are the same between PNA and PSA.

7.2. Hashes

TBD: Probably Toeplitz hash algorithm needs to be added for NICs, since it is often used in receive side scaling.

7.3. Checksums

7.4. Counters

7.5. Meters

7.6. Registers

7.7. Random

7.8. Action Profile

7.9. Action Selector

7.10. Packet Digest

8. PNA Table Properties

Table 2 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.

Property name	Type	See also
pna_direct_counter	one DirectCounter instance name	Section [#sec-direct-counter]
pna_direct_meter	one DirectMeter instance name	Section 7.5
pna_implementation	instance name of one ActionProfile or ActionSelector	Sections 7.8, 7.9
pna_empty_group_action	action	Section 7.9
pna_idle_timeout	PNA_IdleTimeout_t	Section 8.1

Table 2. Summary of PNA table properties.

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

8.1. 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 `main` 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 main parser 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

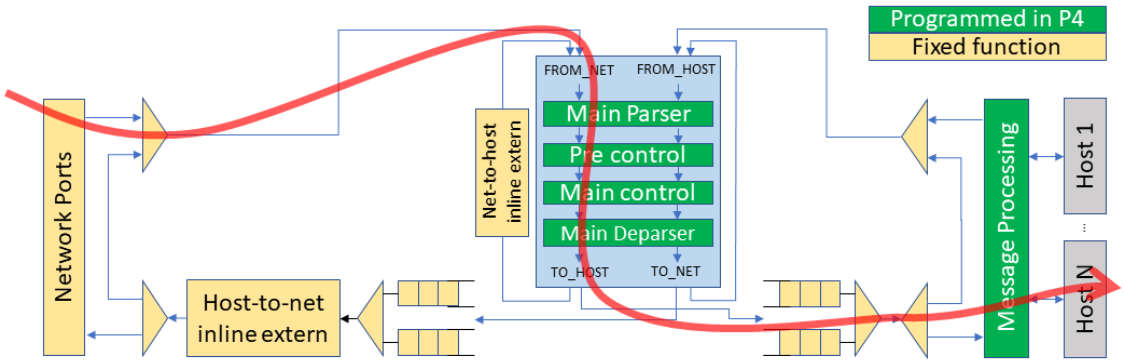

Figure 4. Network to host packet path

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 main control P4 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 main deparser 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-op in the path.

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

See Figure 7. This is similar to the host to network path, except that the main control P4 code directs that the packet should be mirrored to a host, e.g. an inside-the-NIC CPU complex used for

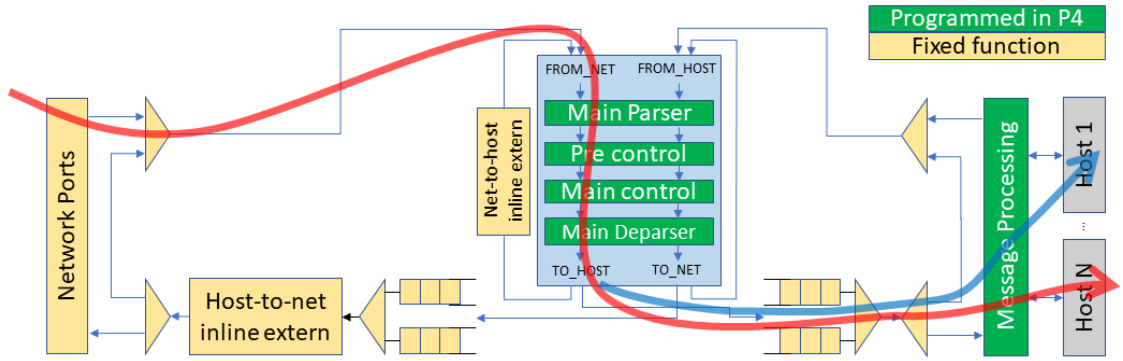

Figure 5. Network to host packet path, with mirror copy to second host

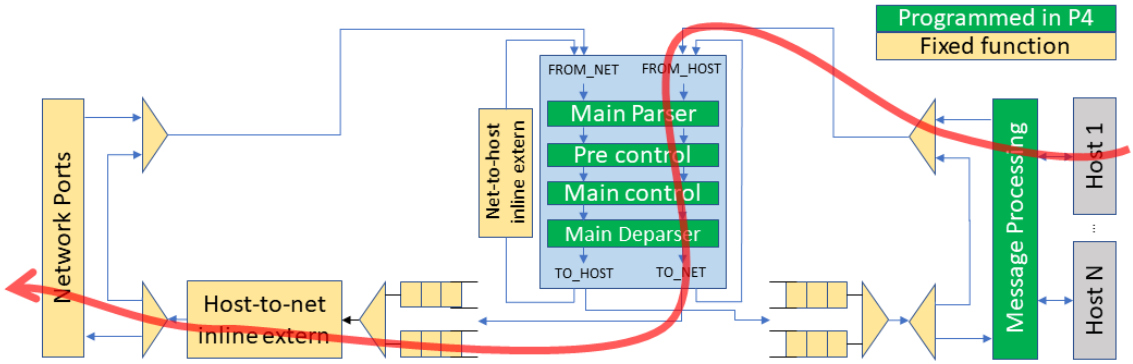

Figure 6. Host to network packet path

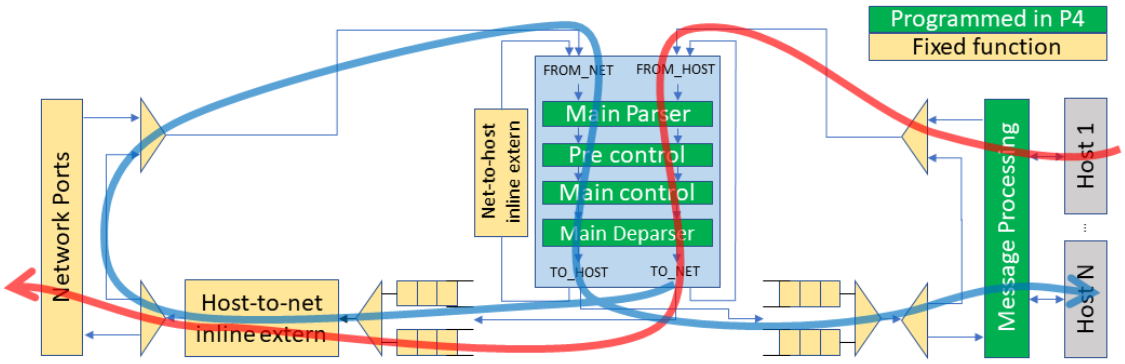

Figure 7. Host to network packet path, with mirror copy to a different host

exception packets. Logically, the copy occurs after the main deparser.

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

C.5. Host to host

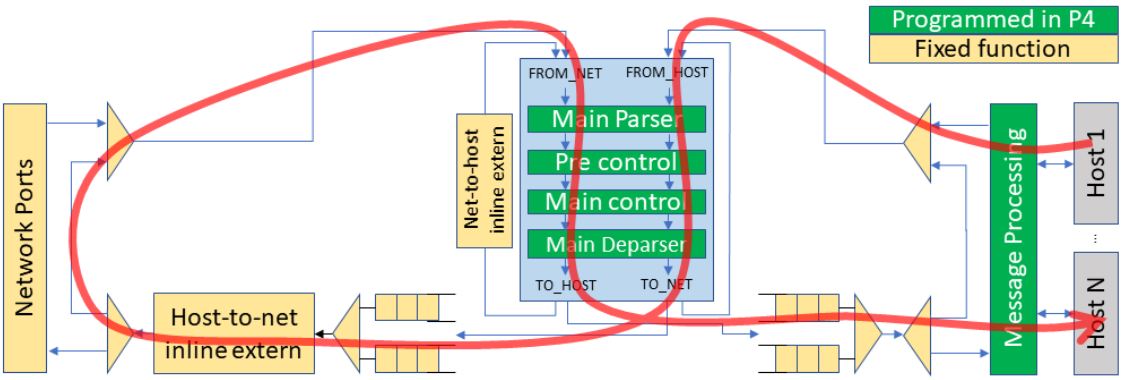

Figure 8. Host to host packet path

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 main control. The net-to-host inline extern may be no-op or perform decryption, as directed by the P4 code in the pre control.

C.6. Port to port

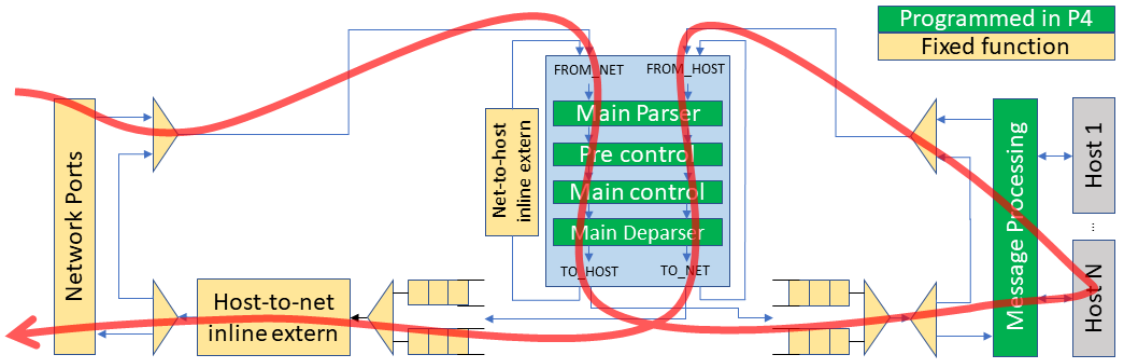

Figure 9. Port to port packet path

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.

D. Appendix: Packet ordering

E. Appendix: Revision History

Release	Release Date	Summary of Changes
0.1	November 5, 2020	Initial skeleton.