PPL Packet Processing Language

September 9, 2005

Copyright © IP Fabrics, Inc. 2005
## Contents

Note: Items that appear with a gray background are not available in the current release. They are included here in order to present the complete language definition.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related Specifications</td>
<td>5</td>
</tr>
<tr>
<td>IP Fabrics’ Packet Processing Language Overview</td>
<td>6</td>
</tr>
<tr>
<td>PPL Objectives</td>
<td>7</td>
</tr>
<tr>
<td>PPL On One Page</td>
<td>8</td>
</tr>
<tr>
<td>PPL Rules</td>
<td>8</td>
</tr>
<tr>
<td>PPL Policies</td>
<td>10</td>
</tr>
<tr>
<td>Events</td>
<td>10</td>
</tr>
<tr>
<td>Parallelism</td>
<td>11</td>
</tr>
<tr>
<td>Values and State</td>
<td>13</td>
</tr>
<tr>
<td>Layer 7 Inspection</td>
<td>14</td>
</tr>
<tr>
<td>Associative Table Structures</td>
<td>15</td>
</tr>
<tr>
<td>Packet Management</td>
<td>16</td>
</tr>
<tr>
<td>IP Fabrics’ PPL Packet Processing Language Specification and Usage</td>
<td>17</td>
</tr>
<tr>
<td>Overall Program Structure</td>
<td>17</td>
</tr>
<tr>
<td>Statements</td>
<td>18</td>
</tr>
<tr>
<td>Define Statement</td>
<td>19</td>
</tr>
<tr>
<td>Event, Run, Wait Statements</td>
<td>20</td>
</tr>
<tr>
<td>Run Groups</td>
<td>20</td>
</tr>
<tr>
<td>Packetless Events</td>
<td>21</td>
</tr>
<tr>
<td>Rules</td>
<td>22</td>
</tr>
<tr>
<td>Expressions</td>
<td>22</td>
</tr>
<tr>
<td>SCAN Expressions</td>
<td>22</td>
</tr>
<tr>
<td>Time Nature of Values</td>
<td>23</td>
</tr>
<tr>
<td>Values</td>
<td>24</td>
</tr>
<tr>
<td>Constants</td>
<td>24</td>
</tr>
<tr>
<td>Named Packet Fields</td>
<td>24</td>
</tr>
<tr>
<td>Dynamic Packet Fields</td>
<td>27</td>
</tr>
<tr>
<td>Packet State Values</td>
<td>28</td>
</tr>
<tr>
<td>Connection State Values</td>
<td>29</td>
</tr>
<tr>
<td>Array Values</td>
<td>30</td>
</tr>
<tr>
<td>Registers</td>
<td>31</td>
</tr>
<tr>
<td>Labels</td>
<td>31</td>
</tr>
<tr>
<td>Indexed Packet Values and Numerically Named Registers, and Their Widths</td>
<td>31</td>
</tr>
<tr>
<td>Operations on Values of Differing Widths</td>
<td>32</td>
</tr>
<tr>
<td>Actions</td>
<td>33</td>
</tr>
<tr>
<td>DROP</td>
<td>33</td>
</tr>
<tr>
<td>FORWARD</td>
<td>33</td>
</tr>
<tr>
<td>STOP</td>
<td>34</td>
</tr>
</tbody>
</table>
Compilation and Dynamic Compilation .......................................................... 93
Regular Expressions ..................................................................................... 94
Examples........................................................................................................ 96
  NAT ............................................................................................................ 96
  Intrusion Detection ................................................................................. 97
  DOS Attack ............................................................................................ 98
  Datagram Reconstruction from Fragments ............................................. 98
  Finding and Converting an IPv6 address in a Layer 7 Protocol .......... 101
  An Approach to Handling Fragments in the Normal Course .......... 102
Currently Unimplemented Language Parts and Restrictions .................. 103

Related Specifications

There are two aspects of the PPL language that are not documented herein. One is the PPL
DeviceMap statement, which describes how to map the PPL program to a particular NPU
architecture and hardware design; DeviceMap is described in a separate specification. The
other is an extension to PPL for ATM, described separately in a PPL ATM Addendum.
IP Fabrics’ Packet Processing Language Overview

PPL is a very-high-level language for describing the processing of network packets. Although it could apply in concept to any type of packet, the language is oriented toward layer 3 IP packets, toward specific protocols at layer 4 (e.g., TCP and UDP), and toward “deep” packet processing at layer 7. In most senses, PPL is a functional language as opposed to a procedural language such as C, although PPL does a few procedural constructs such that a procedural algorithm can be expressed in PPL. PPL also defines explicitly several concepts of concurrent processing.

In addition to being applicable to broad types of packet processing, PPL contains specific features oriented toward applications such as encryption, authentication, stateless and stateful firewall filtering, detection of intrusions and denial-of-service attacks, layer 7 filtering, traffic management, and content-based load balancing.

PPL is intended to be used in an implementation with one or more network processors with high-speed packet-classification capabilities, and as such it represents a very-high-level language for writing concurrent data-plane software or microcode. Although PPL could be compiled in the traditional sense to the low-level code of these network processors, an implementation that is generally more effective is compiling it to a software virtual machine atop the network processor(s). A few tangential aspects of the language assume an interpretive environment (e.g., dynamic compilation – the ability to change the PPL program during operation), but these can be discarded if a direct compilation is desired.

```
Group1: Policy PATTERNS DATABASE($idslist) FIND(Rr1,0,Fuf)
Intruderset: Policy ASSOCIATE NUMBER(10000) SEARCHKEYS(IP_SOURCE) TIMEOUT(10000)
Intruders: Policy RECALL SEARCHKEYS(IP_SOURCE) LINKED(Intruderset)
Secure1: Policy CIPHER ENCRYPT(AES,SHA1) LOCATION(CONTENT,0) PAD(SEQ)
Diversion: Policy PACKET ENCAPSULATE L3(20,IPv4)
Rule EQ(TCP_SYN,1) EQ(TCP_RST,1) DROP # Protocol anomaly
Rule EQ(TCP_SYN,1) EQ(TCP_FIN,1) DROP # Another protocol anomaly
Rule EQ(IP_SOURCE,MYIPADDR) DROP # Source spoofed packet
Rule EQ(IP_SOURCE,public) APPLY(Intruderset) DROP
Rule NE(IP_DEST,MYIPADDR) EQ(ICMP_TYPE,ECHO) DROP # no pings to the inside
Rule EQ(IP_DEST/24,190.10.10.0) SET(Rr0,192.68.0.0 + IP_DEST/24)
Rule SCAN("\x0D\x0A\x5B\x52\x30\x30\x32\x0D\x0A\") JUMP(found_subseven_trojan)
Rule EQ(IP_DEST/24,190.10.10.0) SET(IP_DEST,R0) SET(Rr0,192.68.0.0 + IP_DEST/24)
Rule EQ(IP_DEST/24,boston_gateway) EQ(IP_SOURCE,Portland_gateway/24) APPLY(secure1) FORWARD(1)
Rule EQ(IP_DEST/24,190.10.10.0) APPLY(Group1) FORWARD(2)
```

PPL program
PPL Objectives

As a language, we had several key objectives in mind for PPL. Understanding these objectives is helpful in understanding the language and its implementation.

Productive PPL provides the means to develop high-performance NPU applications in a tenth or less of the time and cost it would otherwise take. As an example, the “bump in the wire” example in Comer’s text *Network Systems Design using Network Processors* requires 570 lines of code using a combination of C and microassembler language, requires 76 lines of FPL code for Agere’s NPU, but requires only 3 lines of PPL code.

Powerful “Primitives” in PPL are complex networking operations. Such things as encrypting a packet, removing an outer header, tracking a TCP connection, and scanning for a regular expression are operations in the language.

Robust Banes of networking software are system hang-ups and security holes. PPL’s high-level primitives, protection mechanisms, and functional nature create far fewer opportunities for subtle bugs. Its implicit protocol-dependent checks on packets eliminate a common source of security holes. One of our favorite examples is the Comer’s text example mentioned above; the PPL code is correct where the C code and Agere FPL code contain a serious flaw (they would behave incorrectly given a fragmented packet).

Self-contained It is possible to write 100% of an NPU application (the data-plane part) in PPL and avoid other languages, learning a new suite of tools, and reading literally thousands of pages of manuals on the NPU.

Open On the other hand, we realize a PPL application needs to have mechanisms to interface with control-plane software and important standards such as the Network Processing Forum’s API, and also that one may want to write 90% of his application in PPL and 10% in a lower-level form.

High performance PPL programs need to provide performance comparable to other alternatives.

Intrinsically parallel Most NPUs consist of a set of highly parallel engines, which is one of the many things that makes writing software for them very difficult. PPL embeds several types of parallelism directly in the language.

Dynamic A major dilemma is that networking equipment needs to stay up 100% of the time, yet it needs to be amenable to software changes to add new functions, detect new security risks, etc. PPL is designed to be dynamically compiled - that is, a PPL program may be changed on the fly.

Portable PPL is not tied to any specific NPU model or even architecture family.
PPL On One Page

The major entities of PPL are shown in the diagram below. Fundamentally, a PPL program consists of a set of rules consisting of expressions and actions. Rule expressions are evaluated and the actions of true rules are executed sequentially and/or in concurrent groups. Actions represent simple actions or the invocation of policies.

### Program Structure and Concurrency

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Defines a starting point in a PPL program.</td>
</tr>
<tr>
<td>Run</td>
<td>Defines a rule group whose actions can be done concurrently with other rule groups.</td>
</tr>
<tr>
<td>Wait</td>
<td>Defines a synchronizing point where concurrency stops.</td>
</tr>
<tr>
<td>Rule</td>
<td>Defines a rule, whose expression(s) are evaluated in parallel with all other rules and whose actions, if true, are performed sequentially within the rule group.</td>
</tr>
<tr>
<td>Array</td>
<td>Defines an indexable list of values.</td>
</tr>
<tr>
<td>Devicemap</td>
<td>Defines the mapping to a specific implementation and hardware.</td>
</tr>
<tr>
<td>Policy</td>
<td>Defines a policy that is invoked upon invocation of an APPLY action in a true rule.</td>
</tr>
</tbody>
</table>

### Expression

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq</td>
<td>value, value</td>
</tr>
<tr>
<td>Ne</td>
<td>value, value</td>
</tr>
<tr>
<td>Ge</td>
<td>value, value</td>
</tr>
<tr>
<td>Le</td>
<td>value, value</td>
</tr>
<tr>
<td>Scan</td>
<td>string or regular expression, …</td>
</tr>
<tr>
<td>Scanb</td>
<td>string or regular expression, …</td>
</tr>
<tr>
<td>Scane</td>
<td>string or regular expression, …</td>
</tr>
</tbody>
</table>

### Action

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>value, value</td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>Log</td>
<td>value</td>
</tr>
<tr>
<td>Set</td>
<td>value, value, exp</td>
</tr>
<tr>
<td>Compute</td>
<td>function, value, value</td>
</tr>
<tr>
<td>Jump</td>
<td>value</td>
</tr>
<tr>
<td>Apply</td>
<td>value</td>
</tr>
<tr>
<td>Act</td>
<td>value</td>
</tr>
<tr>
<td>Lock/Unlock</td>
<td></td>
</tr>
</tbody>
</table>

### Policy

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associate</td>
<td>Defines a content-addressable table.</td>
</tr>
<tr>
<td>Recall</td>
<td>Looks up an entry in a table.</td>
</tr>
<tr>
<td>Disassociate</td>
<td>Removes an entry from a table.</td>
</tr>
<tr>
<td>Cipher</td>
<td>Performs encryption/authentication.</td>
</tr>
<tr>
<td>Classify</td>
<td>Does a multi-field, multi-criteria lookup.</td>
</tr>
<tr>
<td>Connections</td>
<td>Defines a table for tracking connections.</td>
</tr>
<tr>
<td>Defrag</td>
<td>Assembles fragmented packets.</td>
</tr>
<tr>
<td>Newpacket</td>
<td>Creates a new packet.</td>
</tr>
<tr>
<td>Packet</td>
<td>Manipulates packets.</td>
</tr>
<tr>
<td>Program</td>
<td>Invokes a software function outside of PPL.</td>
</tr>
<tr>
<td>Patterns</td>
<td>Compares the packet to a set of patterns.</td>
</tr>
<tr>
<td>Queues</td>
<td>Defines a set of queues.</td>
</tr>
<tr>
<td>Queue</td>
<td>Operates on a queue.</td>
</tr>
<tr>
<td>Rate</td>
<td>Maintains a time-based rate.</td>
</tr>
<tr>
<td>Newsuperpacket</td>
<td>Creates a superpacket.</td>
</tr>
<tr>
<td>Superpacket</td>
<td>Operates on a superpacket.</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>192.68.0.0</td>
</tr>
<tr>
<td>Predefined constant</td>
<td>IPv4, TCP, AH</td>
</tr>
<tr>
<td>Packet field</td>
<td>IP_DEST/24, TCP_SYN</td>
</tr>
<tr>
<td>Dynamic packet field</td>
<td>CONTENT(RR0), PFIELD(40).w</td>
</tr>
<tr>
<td>Packet state</td>
<td>PS_FRAG, PS_CONTENTSIZE</td>
</tr>
<tr>
<td>Current connection</td>
<td>CX_STATE</td>
</tr>
<tr>
<td>Array</td>
<td>Trf_class(0)</td>
</tr>
<tr>
<td>Registers</td>
<td>RR0</td>
</tr>
<tr>
<td>Statement label</td>
<td>RR0</td>
</tr>
</tbody>
</table>

**IP Fabrics Confidential**

11/14/2005
PPL Rules

Rules are the central part of PPL. A rule basically says “make these tests, and if all tests are true, take these actions.” A rule has the format

Rule expression expression ... action action ...

For instance, a very simple rule with one expression and one action is

\[
\text{Rule } \text{EQ}(\text{IP\_SOURCE}, \text{IP\_DEST}) \text{ DROP}
\]

which says – if the current packet is an IP packet and the source and destination IP addresses are identical, drop the packet. Note that it is possible to distinguish between IPv4 and IPv6 as needed, although it is also possible to write PPL programs that are independent and apply to each. In fact, the rule above applies to both IPv4 and IPv6 packets.

If multiple expressions are present, the action or actions are performed only if every expression is true. For instance, the rule below says that policy secure1 should be applied to a packet whose destination is not dmz_ipaddr and whose layer-4 protocol is TCP. (The Define statement used below does a pure character substitution in the program.)

\[
\text{Define } \text{dmz\_ipaddr}="192.168.1.120" \text{ \# IP address of the front-end port} \\
\text{Rule } \text{NE}(\text{IP\_DEST}, \text{dmz\_ipaddr}) \text{ EQ}(\text{IP\_PROT}, \text{TCP}) \text{ APPLY(secure1)}
\]

The above rule has the form

Rule NE(value, value) EQ(value, value) APPLY(value)

Values can be a variety of things, including literal numbers, names of fields in the packet, predefined state values, and registers (short-term values). A value can also have a mask applied to it, such as dmz_ipaddr/24 (24 leading one’s).

Actions on rules include:

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP</td>
<td>Drop the current packet</td>
</tr>
<tr>
<td>FORWARD</td>
<td>Forward the current packet</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop processing (of the current event)</td>
</tr>
<tr>
<td>LOG</td>
<td>Log the packet</td>
</tr>
<tr>
<td>APPLY</td>
<td>Apply the specified policy to the packet</td>
</tr>
<tr>
<td>JUMP</td>
<td>Transfer to the specified rule</td>
</tr>
<tr>
<td>SET</td>
<td>Perform an arithmetic or logical function and assign result (e.g., to a packet field, register, or state)</td>
</tr>
<tr>
<td>COMPUTE</td>
<td>Performs one of a set of computational functions, including converting a number in character form to a binary number, hashing, generating a random number, and getting current time.</td>
</tr>
<tr>
<td>LOCK</td>
<td>Lock a lock, waiting for the unlocked state if necessary</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>Unlock a lock</td>
</tr>
</tbody>
</table>
PPL Policies

Policies are broad, often complex, actions on or using a packet. Policies should be thought of as an adjunct to rules, because specific rules decide to apply specific policies. The policies in the language include:

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOCIATE</td>
<td>defines a content-addressable table and how entries are created</td>
</tr>
<tr>
<td>RECALL</td>
<td>retrieves an entry from an associate table</td>
</tr>
<tr>
<td>DISASSOCIATE</td>
<td>deletes an entry from an associate table</td>
</tr>
<tr>
<td>CIPHER</td>
<td>performs general incremental encryption / authentication</td>
</tr>
<tr>
<td>CLASSIFY</td>
<td>does a multi-field, multi-criteria lookup</td>
</tr>
<tr>
<td>CONNECTIONS</td>
<td>maintains a table of connections (e.g., TCP connections)</td>
</tr>
<tr>
<td>DEFRAG</td>
<td>assembles fragmented packets into nonfragmented ones</td>
</tr>
<tr>
<td>CONTROL</td>
<td>controls event processing</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>signals or invokes an external function</td>
</tr>
<tr>
<td>PATTERNS</td>
<td>compares the packet to a database of patterns</td>
</tr>
<tr>
<td>PACKET</td>
<td>manipulates the current packet or other packets</td>
</tr>
<tr>
<td>NEWPACKET</td>
<td>creates a new packet</td>
</tr>
<tr>
<td>QUEUES</td>
<td>Defines a set of queues</td>
</tr>
<tr>
<td>QUEUE</td>
<td>Operates on a queue</td>
</tr>
<tr>
<td>RATE</td>
<td>tracks a time-based rate</td>
</tr>
<tr>
<td>SUPERPACKET</td>
<td>performs a function on a superpacket</td>
</tr>
<tr>
<td>NEWSUPERPACKET</td>
<td>creates a new superpacket</td>
</tr>
</tbody>
</table>

For example, the following policy describes a particular mode of IPSec processing

```
Secure1: Policy CIPHER ENCRYPT(AES,SHA1) KEY(keytable(Rr0).q)
          LOCATION(CONTENT,0) PAD(SEQ,ssl)
```

A policy is invoked in a rule via the APPLY action. Typically the value of the APPLY action will be the label of a policy, although the value can also be dynamic (e.g., to select among a set of policies).

**Events**

Rules are grouped as events, such as

```
Event(1,2)  # Rules to be processed for incoming packets on ports 1 and 2
Rule EQ(TCP_SYNONLY,1) APPLY(TCPconnectionrate) FORWARD
Rule       LOG DROP
```

Events serve two purposes. First, they identify the occurrences that cause rules to be processed. The most typical circumstance is the arrival of a packet into the system, but events can also be associated with system startup, exceptions, and internal program communications (from PPL or from another program in the system). Second, as discussed in the next section, events represent a level of concurrency in PPL; rules associated with multiple events can be processed concurrently, and even multiple occurrences of the same event can be processed concurrently unless the program indicates otherwise (by labeling the event a SERIAL event).
Parallelism

A major attribute of PPL is concurrency. In general it is more appropriate to think of all of the rules in a PPL program as being processed concurrently than to think of the rules as being executed sequentially. However, as we will show, there are ways to have coarse- and fine-grained control over concurrency. There are four separate concepts of concurrency:

1. Concurrent evaluation of expressions in rules
2. Concurrent Run groups (smaller designated groups of rules that can be processed simultaneously with other groups within an event).
3. Different Events running concurrently (multiple sections of a PPL program that process asynchronously and concurrently)
4. Multiple concurrent instances of the same event

With one exception discussed later, the expressions of all rules are evaluated concurrently and then the actions are performed sequentially. Consider a program of three rules of the form

\[
\text{Rule } \text{expression} \text{ expression} \ldots \text{ action} \text{ action} \ldots \\
\text{Rule } \text{expression} \text{ expression} \ldots \text{ action} \text{ action} \ldots \\
\text{Rule } \text{expression} \text{ expression} \ldots \text{ action} \text{ action} \ldots
\]

All of the expressions of all of the rules are evaluated simultaneously, which might lead us to something like

\[
\text{Rule } \text{true} \ldots \text{ action} \text{ action} \ldots \\
\text{Rule } \text{false} \ldots \text{ action} \text{ action} \ldots \\
\text{Rule } \text{true} \ldots \text{ action} \text{ action} \ldots
\]

which would lead to performing the actions of rule 1 sequentially, and then the actions of rule 3 sequentially.

Now, let’s say that there is no reason to perform the actions sequentially and we’d like to communicate the fact that the system may execute the actions themselves concurrently. We can use the Run statement

\[
\text{Run Rule } \text{true} \ldots \text{ action} \text{ action} \ldots \\
\text{Run Rule } \text{false} \ldots \text{ action} \text{ action} \ldots \\
\text{Run Rule } \text{true} \ldots \text{ action} \text{ action} \ldots
\]

Run states that any rules thereafter until the next Run or Wait can have their actions performed concurrently. In the case above, the actions of the first and third rules are performed concurrently. Within a rule, actions are still performed sequentially.\(^2\)

The more typical use of Run is to identify groups of rules whose actions can be performed concurrently. Here we would write our program as

---

1 To keep things simple, run groups are not discussed in this overview.
2 If we wanted to perform the actions within a rule concurrently, we could simply write them as separate rules.
Run
Group of rules whose actions are processed sequentially
Run
Group of rules whose actions are processed sequentially
Wait
Group of rules whose actions are processed sequentially, but only when all concurrent run groups have finished processing and no action has occurred that terminated execution

One needs to realize that expression evaluation parallelism is always present, so that all three groups of rules above are still evaluated concurrently, independent of the use of Run and Wait. Run and Wait affect the concurrency of actions. Note that the sequential processing of actions allows one full control over the deterministic behavior of rules, something many people think is important in writing firewall rules, for instance.

The third concept of concurrency is events. The highest level division of a PPL program is rules that execute upon the occurrence of a specific event. The simplest example of an event is the arrival of a packet on a particular port. For instance, if one’s system was a “bump in the wire,” it would typically have two ports, and one would structure one’s PPL program into rules that handle incoming traffic on each port. This would be represented by two events. Each would be a parallel “thread of execution” in the PPL program.

The following is an example of concurrency. First, all nine rules are evaluated simultaneously. Then any true rules in the first group of four have their actions processed, and in parallel with that any true rules in the second group of two have their actions processed. Then, once we’re done with both groups (and providing we didn’t terminate the program by STOP), we process the actions of any of the true last three rules.

```
group1: Policy ASSOCIATE NUMBER(1000) SEARCHKEYS(IP_DEST)
group2: Policy ASSOCIATE NUMBER(1000) SEARCHKEYS(IP_DEST)

Event(0)
Run
  Rule EQ(IP_SOURCE,MYIPADDR) DROP STOP #refuse source addr spoofed packet
  Rule EQ(IP_SOURCE,0.0.0.0) DROP STOP #refuse bad source address
  Rule EQ(IP_SOURCE,255.255.255.255) DROP STOP #refuse bad source address
  Rule EQ(IP_DEST,127.0.0.0) DROP STOP #refuse loopback address
Run
  Rule GE(L4_DPORT,1024) EQ(IP_PROTO,TCP) DROP STOP
  Rule GE(L4_DPORT,2049) LE(L4_SPORT,4045) DROP STOP
Wait
  Rule EQ(L4_DPORT,80) APPLY(group1)
  Rule NE(L4_DPORT,80) APPLY(group2)
  Rule FORWARD(1)
```

Now, given the above, the fact that this is defined as one event means that all of this (e.g., processing of actions) can be done in parallel with any other event in the program.

---

3 This is a trivial example where the computational load is so light that there is no reason to worry about concurrency. Furthermore, the rules in the two groups are actually all independent from each other, so we could have indicated that every one of them can be concurrent if we wanted.
The last form of concurrency is that the PPL programmer can specify whether concurrent instances of a single event can occur. In the example above, this would actually happen unless the program were written as

```
Event(0) SERIAL
```

SERIAL specifies that multiple instances of performing the event must not occur. If SERIAL is omitted, if packets are available to event 0 more quickly than event 0 processes them, event 0 will be “spawned” if processing resources permit such that multiple event 0’s, each processing a different packet from the same source, will be created.

We mentioned earlier the existence of an exception to the statement that all rules’ expressions are evaluated in parallel at the start and that Run and Wait apply only to the processing of actions. There exists a way to denote that the evaluation of an expression should be deferred until the processing of actions. Such an expression in a rule appears in the list of actions and is evaluated serially amidst the actions; if false, the remainder of the rule is not processed. So if we wrote

```
Rule EQ(IP_DEST,A) SET(Rr0,X) EQ(Rr0,1) . . .
```

the rule is evaluated as true or false based on the comparison of IP_DEST to A. Later, if rule one is true, we perform its actions (and deferred expressions), meaning first we set Rr0 to the value of X and then we test Rr0 for 1 and if false terminate processing of the rule, or otherwise continue.

In general, “normal” expressions lead to best performance. Deferred expressions are sometimes needed when making decisions based on prior actions in the event. Typically you would expect to see expressions that use registers defined as deferred expressions.4

### Values and State

Values appear in expressions, in some actions, and in some policies. The following table summarizes the types of values in PPL.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>10.10.0.0/16</td>
</tr>
<tr>
<td>Predefined constant</td>
<td>IPv4, TCP, AH</td>
</tr>
<tr>
<td>Packet field</td>
<td>IP_DEST, TCP_SEQNUM</td>
</tr>
<tr>
<td>Dynamic packet field</td>
<td>CONTENT(RR0), PFIELD(40).w</td>
</tr>
<tr>
<td>Packet state</td>
<td>PS_PACKETHANDLE</td>
</tr>
<tr>
<td>Current connection</td>
<td>CX_STATE</td>
</tr>
<tr>
<td>Array</td>
<td>Trf_class(0)</td>
</tr>
<tr>
<td>Register</td>
<td>Rr0</td>
</tr>
<tr>
<td>Statement label</td>
<td>group1</td>
</tr>
</tbody>
</table>

4 Except perhaps for the use of global registers. Expressions that are not deferred ones and use non-global registers are still well behaved, although not terribly useful (any register at this point would have the value FuF, a unique value in PPL used to denote errors).
Within an event, there is a concept of the current packet, and its contents can be accessed and changed via named packet fields (e.g., IP_DEST), with the notation PFIELD (a movable window anywhere in the packet), and with the notation CONTENT (a movable window in the packet payload typically used to access data in the layer 7 content area).

PPL defines several sets of registers with fixed names such as Rg0 – Rg255, but the define statement lets one substitute meaningful names as needed. One register, Rr0, is special in that certain expressions and actions use and/or modify it. Since PPL provides for concurrently processed groups of rules, there is an RR0 set per concurrent group (meaning there is a unique Rr0 in each group of rules denoted by Run, Wait, and Event).

As discussed earlier, it is important to understand sequencing and concurrency in PPL and how state does and doesn’t change during the processing of the program, since PPL is quite different in this regard. The PPL program is processed when an event occurs, and In general the time ordering is

1. All rule expressions, except deferred expressions, are evaluated in parallel. The values used are “event entry” values (the value at the time processing of the event starts). The rule is considered true if all of the expressions (except deferred expressions) evaluate to true; otherwise the rule is false.
2. True rules are processed, concurrency and sequence determined by the use of Run and Wait. Values used are current values. If a true rule contains deferred expressions, its processing may be terminated at the point of a deferred expression if that expression evaluates to false.

The first example below shows the difference between expression evaluation occurring first and true-rule processing occurring second. Rule 1 is always true. Rule 2 is true if L4_SPORT had the value 60 when the event occurred.

<table>
<thead>
<tr>
<th>Rule</th>
<th>SET(L4_SPORT,60) # 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>EQ(L4_SPORT,60) SET(L4_SPORT,80) # 2</td>
</tr>
</tbody>
</table>

The examples below show the time nature of values used in a variety of scenarios.

<table>
<thead>
<tr>
<th>Rule</th>
<th>EQ(IP_DEST,1.1.1.1) . . . # Evaluated at start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>one or more actions EQ(IP_DEST,Rr0) . . . # Evaluated during processing</td>
</tr>
<tr>
<td></td>
<td># using current IP_DEST # and Rr0 values</td>
</tr>
<tr>
<td>Rule</td>
<td>EQ(IP_DEST,IP_TABLE(0)) . . . # Evaluated at start</td>
</tr>
<tr>
<td>Rule</td>
<td>one or more actions EQ(IP_DEST,IP_TABLE(Rr0)) . . .</td>
</tr>
<tr>
<td></td>
<td># Evaluated during processing, using</td>
</tr>
<tr>
<td></td>
<td># current IP_DEST, Rr0, and selected</td>
</tr>
<tr>
<td></td>
<td># IP_TABLE value</td>
</tr>
</tbody>
</table>

**Layer 7 Inspection**

PPL contains powerful mechanisms to examine information in the content part of a packet. These can be used for a variety of purposes, for example, detecting viruses, locating intrusion signatures, scrubbing application protocols for DoS attacks, and using packet content for purposes of load balancing.
A basic mechanism is provided by relocatable value CONTENT. This can be positioned at any offset in the content and allow one to use and/or modify one-, two-, four-, and eight-byte fields.

A powerful capability is the SCAN expression, which examines the content part of a packet for a string. SCAN can express a simple character or hexadecimal string, or it can also bring the power of regular expressions to bear. A simple example of SCAN is

```
Rule SCAN(“|0D0A5B52504C5D3030320D0A|”) LOG(found_subseven_trojan)
```

Alternative forms – SCANB and SCANE – return the offset of the beginning or end of the discovered string.

SCAN can express a character string, and this string can be a regular expression. For instance, let’s suppose we wish to examine the content part of all packets going to TCP port 80 to see if they are a GET HTTP transaction with a URL ending with redirect.html and containing a session cookie. The PPL rule would be

```
Rule SCAN(re"GET.*?redirect.html[:space:]*HTTP/1.*?Cookie:","0,0)
EQ(IP_PROT,TCP) EQ(L4_DPORT,80) APPLY(group1)
```

Because SCAN is an expression, it follows the rules of concurrency in PPL, meaning that it is intended to be done concurrently with all other expression evaluations (including other SCANs).

SCAN has other capabilities, such as the ability to start the scan at a specific offset, terminate the scan after reaching a specific offset, and anchor the scan, and scan “backward.”

Finally, PPL contains a PATTERNS policy that invokes an algorithm to do multiple-pattern matching.

**Associative Table Structures**

PPL defines two means to create and use associatively searched tables. One, expressed in the CONNECTIONS policy, is a predefined table oriented toward tracking layer-4 communications (e.g., TCP and UDP). Because there is a generally accepted way to do this and because it is so commonly used, a specific mechanism was placed in PPL for it.

The second mechanism is a generalization of this, basically the ability to create a two-dimensional table of arbitrary length and width and place associatively searched keys in it. As examples of some uses, a basic use would be for a pool of dynamically allocated NAT addresses or ports. It could be used to monitor any amount of layer 3-7 information for tracking or QoS or billing purposes. It could be used as a tool by a system administrator to understand network traffic in the general case, or to probe for very specific types of traffic. It could be used for traffic management or “bandwidth policing” by regulating how much of the network bandwidth can be used by particular users, systems, protocols, etc.

As a simple example, suppose we wish to maintain a table that tracks unique combinations of IP addresses and ports on packets that have passed through. We could say
Here the rule will create a unique entry in the table each time the values of these four fields in the current packet don’t match a current entry. If the entry already existed, no change would be made.

Packet Management

Most applications in which PPL is used are anticipated as dealing with a single packet, that is, a packet arrives, is examined, is possibly transformed, and is either forwarded on or dropped. For more complex applications, PPL also allows one to manipulate multiple packets. There is the concept of a packet handle by which a packet is known. One could put the handle of the current packet in a table, for instance, and go back to that packet at some later point in time. One can also create a new packet by applying the NEWPACKET policy. A variety of means are provided to encapsulate a packet and the opposite (strip an outer header), including bundling this with encryption and authentication per the IPSec standard.

PPL also provides for the creation, management, and manipulation of “superpackets.” A superpacket is a set of packets that are to be treated as one. Uses for superpackets include IP fragment reassembly and TCP stream reassembly.

Interfaces to External Data-Plane, Control-Plane, Management-Plane Software

If there is a need from within a PPL program to communicate directly with other software in the system, the PROGRAM policy can be used. Examples might be

- Generating alerts to send to higher-level control or management software
- Forwarding packets up to specific control-plane tasks for exception handling (e.g., TCP proxying, IKE key exchange)
- Invoking “secret sauce” customer-written data-plane microcode

Although outside of the definition of the language itself, the implementation is expected to provide runtime interfaces (APIs) to allow external programs to forward a packet to a PPL event and signal a PPL event (invoke an event with no packet).
IP Fabrics’ PPL Packet Processing Language
Specification and Usage

In as readable way as possible, this describes the complete language syntax and semantics and gives examples of the usage of many features. We assume the reader is already familiar with the PPL Overview document.

Overall Program Structure

The minimal structure of a PPL program is

```
Event
Rule
```

and the general structure of a program is

```
All Define statements might appear here
All Policy and Array statements appear here
Event
  Rule # One or more rules whose actions are performed first
  Run
    Rule # One or more rules whose actions are performed concurrently
    Run
      Rule # One or more rules whose actions are performed concurrently
    Wait
      Rule # One or more rules whose actions are performed after the above
Event
  Rule # One or more rules whose actions are performed first
  Run
    Rule # One or more rules whose actions are performed concurrently
    Run
      Rule # One or more rules whose actions are performed concurrently
    Wait
      Rule # One or more rules whose actions are performed after the above
```

We have shown two events with two run groups and one wait group, but there may be an arbitrary number of each, including zero.
**Statements**

The statements in the language are specified in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[label:] Rule</td>
<td>Specifies zero or more expressions to be evaluated and zero or more actions to be taken</td>
</tr>
<tr>
<td>[label:] Policy</td>
<td>Specifies major operations on a packet or system state changes</td>
</tr>
<tr>
<td>Define</td>
<td>Defines a compile-time substitution of strings in the PPL program</td>
</tr>
<tr>
<td>Event</td>
<td>Denotes that the group or groups of rules following this are to be invoked when the designated events occurs</td>
</tr>
<tr>
<td>Run</td>
<td>The group of rules following this, and up to the next Run, Wait, or Event statement or the end of the program, can have their actions performed concurrently with other groups</td>
</tr>
<tr>
<td>Wait</td>
<td>No rule actions beyond this shall be performed until any groups of rules associated with the event have completed their actions</td>
</tr>
<tr>
<td>label: Array</td>
<td>Defines an indexable list of equal-size data</td>
</tr>
<tr>
<td>DeviceMap</td>
<td>Defines the mapping of the PPL environment to a specific implementation and hardware(^5)</td>
</tr>
</tbody>
</table>

Every Policy statement in a program is assigned a unique number by the compiler, and every rule statement is assigned a unique number. The language does not specify the starting numbers, but it does require that each policy be numbered sequentially in order of appearance in the program, and each rule within an event be numbered sequentially. So in the example below

```
group1: Policy ASSOCIATE . . .
group2: Policy ASSOCIATE . . .
Event(. . .)
    Rule EQ(. . .)
case: Rule NE( . . .)
```

the two policies are numbered sequentially (e.g., 0 and 1) and the two rules numbered sequentially (e.g., 0 and 1). When a label is placed on a policy or rule, the label takes on the corresponding value.

One place outside of the PPL definition where rule numbers are used is in logging. Within the language, one can take advantage of the numbering to refer to one of a sequence of policies dynamically and jump to a sequence of rules dynamically.

A comment is a program line beginning with the # character, or the remainder of a program line beginning with the # character.

PPL is case insensitive (i.e., for language keywords and program labels). Characters within a quoted string are not case insensitive.\(^6\)

---

\(^5\) DeviceMap is not discussed further in this document. See the document *PPL IXP2000-Family Device Map*.

\(^6\) For string operations (SCAN) there is an explicit way to denote that the characters are case insensitive.
Define Statement

Define statements have the form

\[
\text{Define } \text{stringbody}=\text{string}
\]

Before the program processes, any and all occurrences of the sequential characters \textit{stringbody} (whether standalone or within a larger string) are replaced by the characters within \textit{string}.

Examples:

```
Define icmp_echo_counter="Rg50"
Define subseven_signature="|0D0A5B52504C5D3030320D0A|
```

To place a double-quote mark within a string, precede it with another double quote. For instance

```
Define mum="""Mum"" is the word"
```

The names \textit{stringbody} and \textit{label} must begin with a letter or underscore. Successive characters can be letters, numerals, and underscore.

A Define statement may appear anywhere within the program, but must appear prior to the symbol being replaced.
Event, Run, Wait Statements

These statement types, like Define, are instructions to the compiler. They define how rules are grouped to form the overall program structure. Each set of rules in a program that immediately follows one of these statements and that ends just before another occurrence of one of these statements or the end of the program text is called a group. We will call a group that immediately follows a Run statement a "run group."

The Event statement has the form

\[
\text{Event}([\text{list of constants}]) \text{ [SERIAL]}
\]

where \text{list of constants} identifies one or more events. If SERIAL is not specified, when one of the events in the list occurs, the group following the Event statement is processed. If SERIAL is specified, when one of the events in the list occurs the group following the Event statement will be processed as soon as the PPL program is not still processing a previous event in the list.\(^7\)

Other than a reserved set of event numbers (990-999), event numbers map directly to logical port numbers. Mapping of logical port numbers to real entities is done in the implementation-specific DeviceMap statement. Logical port numbers typically map to physical network ports and to other programs in the system outside of the context of PPL.

Events 990-997 are reserved and should not be used. Event 998 denotes an exception in the PPL program. Event 999 is invoked when the system has just started operation and before any other events are processed.

Processing of actions relative to an event terminates explicitly upon execution of a STOP action, implicitly by reaching an Event statement, or physical end of the program. If processing terminates implicitly and there is a current packet, the packet is dropped.

When an event is not associated with the arrival of a packet (e.g., event 999), there is no current packet; any reference to either will cause an exception. Also, such events may receive parameters, which are placed in the first N event registers. Event 998 (exception handler) is an example of an event receiving parameters.

Run Groups

A run group is a set of rules whose actions can be performed concurrently with those of any other run group residing adjacently above or below in the program text but with no intervening

\(^7\) When SERIAL is not present, the rules in the Event could be executing multiple times concurrently, for instance on behalf of multiple packets of a stream of packets. In a sense one can think of this as being reentrant. If the PPL program logic is consistent with this, this yields the best performance. If events need to be "queued" such that at most one is permitted through the Event group at any one time, specify SERIAL.

Note that this orthogonal to any maintenance of “per flow” packet ordering done by the implementation. PPL does not specify whether or not this occurs. The separate specification for the IXP2000-Family PPL DeviceMap does specify how per-flow packet processing occurs relative to events, as well as an option to disable this.
Wait, Run, or Event statement. The role of Wait is to denote a boundary in the program where the groups above the Wait are *not* invoked concurrently with the groups below the Wait.

Each group (actually each instance of each group) has its own registers RR0-RR3.

Within a group, actions of “true” rules are performed sequentially, and if a “true” rule specifies multiple actions, those actions are also performed sequentially. (See, however, the discussion of the JUMP action.)

**Packetless Events**

Processing of an event begins in one of two states: either the event has been initiated by a packet, in which case there is a current packet and the event registers are initialized to FuF, or the event has been initiated by some other means, in which case there is no current packet and the first N event registers are filled with parameters from the source initiating the event.

Packetless events can be initiated in the following ways:

- The startup event (event 999)
- The exception event (event 998). Parameters are provided to this event as described later in the chapter on exceptions.
- An event triggered from a policy (e.g., see CONTROL, DEFRAg, and SUPERPACKET policies).
- An event triggered by another program. For instance, this could be a control- or data-plane program outside of the context of PPL through an API. It could also come indirectly from the PPL program itself via use of the PROGRAM policy.
Rules

A rule consists of

1. An optional label
2. Zero or more expressions
3. An action
4. Zero or more additional actions and/or deferred expressions

Expressions

A rule is considered true if it has no expressions or if every expression (except deferred expressions) is evaluated as true. Expressions of all rules are evaluated concurrently at the beginning of program execution, unless the expression is a deferred expression.

If a rule is true, the actions and deferred expressions specified are performed in sequential order. If a deferred expression evaluates to false, the processing of the rule terminates.

An expression in a rule can have the following forms

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ(value,value)</td>
<td>Equal</td>
</tr>
<tr>
<td>NE(value,value)</td>
<td>Not equal</td>
</tr>
<tr>
<td>GE(value,value)</td>
<td>Greater than or equal (first value to second value)</td>
</tr>
<tr>
<td>LE(value,value)</td>
<td>Less than or equal (first value to second value)</td>
</tr>
<tr>
<td>SCAN(string[,svalue[,dvalue]])</td>
<td>Scans the packet for the value expressed as a string. Svalue contains the offset of the starting point to scan in the content. If the whole string is found, the expression is true.</td>
</tr>
<tr>
<td>SCANB(string[,svalue[,dvalue]])</td>
<td>Scans the packet for the value expressed as a string. Svalue contains the offset of the starting point to scan in the content. If the whole string is found, the expression is true.</td>
</tr>
<tr>
<td>SCANE(string[,svalue[,dvalue]])</td>
<td>Scans the packet for the value expressed as a string. Svalue contains the offset of the starting point to scan in the content. If the whole string is found, the expression is true.</td>
</tr>
</tbody>
</table>

Unless otherwise explicitly noted, any mention of expression applies to both non-deferred and deferred expressions.

SCAN Expressions

SCAN can search on character strings, hexadecimal strings, regular expressions, and combinations of the above. The scan can be started at a particular offset, terminated beyond a particular other offset, and/or anchored.

The form of string above is

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;characters...&quot;</td>
<td>Case sensitive character comparison</td>
</tr>
<tr>
<td>ul&quot;characters&quot;</td>
<td>Case insensitive character comparison</td>
</tr>
<tr>
<td>re&quot;regular expression&quot;</td>
<td>Regular expression (case sensitive)</td>
</tr>
<tr>
<td>reul&quot;regular expression&quot;</td>
<td>Regular expression (case insensitive)</td>
</tr>
</tbody>
</table>

If |....| appear in a character string, the |’s enclose hexadecimal encodings (e.g., “the end.|0D|”). If one wants to include | or " as an actual character in the string, write it as \\ and ". Also, writing \\ means: include one \ in the string.
Svalue and dvalue represent the point at which the scan starts and optionally stops. The way their values control the semantics of SCAN is defined below.

<table>
<thead>
<tr>
<th>Svalue and dvalue omitted</th>
<th>Scan starts at offset 0 and proceeds forward until a match occurs or the end of the payload is encountered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dvalue omitted</td>
<td>Scan starts at offset svalue and proceeds forward until a match occurs or the end of the payload is encountered</td>
</tr>
<tr>
<td>Svalue &lt; dvalue</td>
<td>Scan starts at offset svalue and proceeds forward until a match occurs, or the end of the payload is encountered, or the match is attempted at dvalue+1.</td>
</tr>
<tr>
<td>Svalue = dvalue</td>
<td>Scan is anchored at offset svalue – either the match occurs there or not.</td>
</tr>
<tr>
<td>Svalue &gt; dvalue</td>
<td>Scan is false.</td>
</tr>
</tbody>
</table>

Like other expressions, SCAN evaluates to true or false. However, two special forms – SCANB and SCANE – also return a value, and thus these two expressions are always interpreted as deferred expressions. SCANB sets Rr0 to the offset in the packet content of the first character of the completely matching string. SCANE sets Rr0 to the offset in the packet content of one byte beyond the end of the completely matching string. If they evaluate to false, Rr0 is not changed.

Assuming a 10-character content of AbcAbcAbcA

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCANB(&quot;cA&quot;)</td>
<td>returns true and Rr0=2</td>
</tr>
<tr>
<td>SCANB(&quot;cA&quot;,0)</td>
<td>returns true and Rr0=2</td>
</tr>
<tr>
<td>SCANB(&quot;cA&quot;,0,0)</td>
<td>returns false</td>
</tr>
<tr>
<td>SCANB(&quot;cA&quot;,3,5)</td>
<td>returns true and Rr0=5</td>
</tr>
<tr>
<td>SCANB(&quot;cA&quot;,999,999)</td>
<td>returns false</td>
</tr>
<tr>
<td>SCANB(ul&quot;abcabc&quot;)</td>
<td>returns true and Rr0=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCANE(&quot;cA&quot;)</td>
<td>returns true and Rr0=4</td>
</tr>
<tr>
<td>SCANE(&quot;cA&quot;,0)</td>
<td>returns true and Rr0=4</td>
</tr>
<tr>
<td>SCANE(&quot;cA&quot;,0,0)</td>
<td>returns false</td>
</tr>
<tr>
<td>SCANE(&quot;cA&quot;,3,5)</td>
<td>returns true and Rr0=7</td>
</tr>
<tr>
<td>SCANE(&quot;cA&quot;,999,999)</td>
<td>returns false</td>
</tr>
<tr>
<td>SCANE(ul&quot;abcabc&quot;)</td>
<td>returns true and Rr0=6</td>
</tr>
</tbody>
</table>

**Time Nature of Values**

Because PPL separates the time occurrence of evaluating a rule’s expressions and performing its actions if true, it is important to understand the time nature of values used.

- Values used in (non-deferred) expressions are those values that existed upon entry to the event.
- Values used in deferred expressions, actions, and in “APPLY’ed” policies are the current values. That is, if the value was changed by a prior action in the event, the up-to-date value is used.

---

8 One might assume the result of SCANB or SCANE are easily computed from the other. This is the case for a fixed-length string but not the case in general for regular expressions.
Values

Values appear in expressions, in some actions, and in some policies. The following table summarizes the types of values.

<table>
<thead>
<tr>
<th>Type</th>
<th>Can be destination?</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>No</td>
<td>10.10.0.0/16, FuF</td>
</tr>
<tr>
<td>Predefined constant</td>
<td>No</td>
<td>IPV4, TCP, AH</td>
</tr>
<tr>
<td>Packet field</td>
<td>Yes</td>
<td>IP_DEST, TCP_SEQNUM</td>
</tr>
<tr>
<td>Dynamic packet field</td>
<td>Yes</td>
<td>CONTENT(RR0), PFIELD(12)</td>
</tr>
<tr>
<td>Packet state</td>
<td>No</td>
<td>PS_PACKETHANDLE</td>
</tr>
<tr>
<td>Array</td>
<td>Yes</td>
<td>TRF_CLASS(8)</td>
</tr>
<tr>
<td>Current connection</td>
<td>Yes</td>
<td>CX_STATE</td>
</tr>
<tr>
<td>Register</td>
<td>Yes</td>
<td>RR0, RE7, RG100, RR0.q</td>
</tr>
<tr>
<td>Statement label</td>
<td>No</td>
<td>group1</td>
</tr>
</tbody>
</table>

Any of the above followed by /nn, where nn is a number from 1 to 127 indicating how many higher-order bits are not to be masked off. A mask applies to values, not to destinations. So the action SET(Rr0, Re1/16 + Re2/8) means: the value of Re1 anded with FFFF0000 and the value of Re2 anded with FF000000 are added together and the sum stored in Rr0).

Constants

Constants can be one of the following

<table>
<thead>
<tr>
<th>d[d…]</th>
<th>One of more digits 0..9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xhh[hh…]</td>
<td>0x followed by one or more pairs of digits 0..F</td>
</tr>
<tr>
<td>num.num.num.num</td>
<td>A 32-bit value where num is 0..255</td>
</tr>
<tr>
<td>hn:hn:hn:hn:hn:hn:hn:hn</td>
<td>A 128-bit value where hn is 0..FFFF. IPv6 double-colon notation for zero compression is also permitted (e.g., FF02::2)</td>
</tr>
<tr>
<td>’c[c..]’</td>
<td>One or more characters, each of which represented an 8-bit encoding</td>
</tr>
<tr>
<td>FuF</td>
<td>The 32-bit value 0xFFFFFFFF</td>
</tr>
<tr>
<td>Pi</td>
<td>The 32-bit value 3.14159⁹</td>
</tr>
</tbody>
</table>

A number of other predefined constants listed later

There are named values of some fields within packets and tables. E.g., the name TCP has the value 6 because it is intended to be used as a value in the IP_PROT field.

Where the size isn’t given above, a constant is represented as a 32-bit number if it fits, or as a 128-bit number otherwise.

Named Packet Fields

The packet field names are shown in the table below. Note that these names refer to fields within the current packet. If there is no current packet, these fields have no defined value and any references to them create the no-current-packet exception. Fields that begin as “IP_” in general apply to both IPv4 or IPv6 unless otherwise noted. Also, unless otherwise noted, when

---

⁹ Just kidding; Pi is not a name defined in the language.
something in the specification is stated to be associated with the current packet being IPv4 or IPv6, it refers to that state of the packet at the instant it became the current packet (i.e., its arrival state).

For ease of use and to avoid creating programs with basic security holes, almost every named field, when used in an expression, is associated with a set of implicit expressions that are also evaluated for the rule to be true. These implicit expressions are referred to in the third column and defined in a following table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Definition</th>
<th>Implicit expressions</th>
<th>Built-in values</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_VERSION</td>
<td></td>
<td>0</td>
<td>IPv4, IPv6</td>
</tr>
<tr>
<td>IP_HDRLEN</td>
<td>IPv4 only</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IP_TOS_TC_DS</td>
<td>8-bit value in first word of IP header. If IPv4, bits 8-15. If IPv6, bits 4-11.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IP_TOSPREC</td>
<td>Bits 0-2 of IP_TOS_TC_DS (meaningful in IPv4 only)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_TOSTOS</td>
<td>Bits 3-6 of IP_TOS_TC_DS (per RFC 1349, meaningful in IPv4 only)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_DSCP</td>
<td>Bits 0-5 of IP_TOS_TC_DS (per RFC 2474)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IP_DSECN</td>
<td>Bits 6-7 of IP_TOS_TC_DS (per RFC 3168)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IP_PACKETLEN</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_IDENTIFIER</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_DF</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_MF</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_FRAGOFFSET</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_TTL</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_HDRCSUM</td>
<td>IPv4 only</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>IP_FLOWLABEL</td>
<td>IPv6 only</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IP_PAYLOADLEN</td>
<td>IPv6 only</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IP_HOPLIMIT</td>
<td>IPv6 only</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IP_PROT</td>
<td>IPv4 and IPv6. In IPv6, this is not in a fixed place. It is identical to IP_NEXTHDR unless IP_NEXTHDR has one of the values 0, 43, 44, or 60, in which case IP_PROT is the next-header field of the first extension whose value is other than 0, 43, 44, 60.</td>
<td>2</td>
<td>ICMP, IP, TCP, UDP, ESP, AH, ICMPv6</td>
</tr>
<tr>
<td>IP_NEXTHDR</td>
<td>IPv6 only. Always the 6th byte of the IP header.</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IP_SOURCE</td>
<td>Source IP address</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>IP_DEST</td>
<td>Destination IP address</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L4_SPORT</td>
<td>Source port in the IP payload assuming it is a layer 4 protocol containing this field</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>L4_DPORT</td>
<td>Destination port in the IP payload assuming it is a layer 4 protocol containing this field</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ICMP_TYPE</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ICMP_CODE</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>UDP_LENGTH</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>UDP_CSUM</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>TCP_SEQNUM</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>TCP_ACKNUM</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Field name</td>
<td>Definition</td>
<td>Implicit expressions</td>
<td>Built-in values</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>TCP_DATAOFFSET</td>
<td>Bits 8-15 in 4th word of TCP header</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_FLAGS</td>
<td>Bits 8-15 in 4th word of TCP header</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_CWR</td>
<td>Bit 0 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_ECE</td>
<td>Bit 1 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_URG</td>
<td>Bit 2 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_ACK</td>
<td>Bit 3 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_ACKONLY</td>
<td>Means ACK without FIN or SYN</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_PSH</td>
<td>Bit 4 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_RST</td>
<td>Bit 5 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_SYN</td>
<td>Bit 6 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_SYNONLY</td>
<td>Means SYN without ACK or FIN</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_SYNACK</td>
<td>Means SYN and ACK</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_FIN</td>
<td>Bit 7 in flags</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_WINDOW</td>
<td></td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_CSUM</td>
<td></td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>TCP_URGPTR</td>
<td></td>
<td>(8)</td>
<td></td>
</tr>
</tbody>
</table>

Note that changing these fields of the current packet has no semantics other than literally changing the field value.\(^{10}\)

When these field names are used in a non-deferred rule expression, a number of other tests are also added as defined by the table below. So, for instance, saying

Rule $\text{EQ(TCP\_SYN,1)} \text{ APPLY(x)}$

is identical to saying

Rule $\text{EQ(TCP\_SYN,1) EQ(IP\_VERSION,IPv4) GE(IP\_HDRLEN,5) EQ(PS\_NOORFIRSTFRAG,1)}$

$\text{GE(PS\_IPDATASIZE,20) EQ(IP\_PROT,TCP)}$

$\text{APPLY(x)}$

Rule $\text{EQ(TCP\_SYN,1) EQ(IP\_VERSION,IPv6) EQ(PS\_NOORFIRSTFRAG,1)}$

$\text{GE(PS\_IPDATASIZE,20) EQ(IP\_PROT,TCP)}$

$\text{APPLY(x)}$

(PS\_NOORFIRSTFRAG and PS\_IPDATASIZE, defined subsequently in this chapter, are packet-state values computed from the current packet.)\(^{11}\)

---

\(^{10}\) E.g., changing IP\_VERSION to IPv6 does not convert the packet to an IPv6 form. Changing IP\_PACKETLEN does not implicitly shrink or enlarge the packet.

\(^{11}\) One could technically create a first fragment where its size is such that the layer-4 header appeared in a subsequent fragment. No sensible program we know of does this, and thus such a packet is probably an evil one. If one wants to handle this case, one could do so by using the field names that don’t do implicit tests – PFIELD – discussed shortly.
Implicit expression added

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EQ(PS_IP,1)</td>
</tr>
<tr>
<td>1</td>
<td>EQ(IP_VERSION,IPv4)</td>
</tr>
<tr>
<td>2</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) or EQ(IP_VERSION,IPv6)</td>
</tr>
<tr>
<td>3</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5)</td>
</tr>
<tr>
<td>4</td>
<td>EQ(IP_VERSION,IPv6)</td>
</tr>
<tr>
<td>5</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8) or EQ(IP_VERSION,IPv6) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8)</td>
</tr>
<tr>
<td>6</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8) EQ(IP_PROT,ICMP) or EQ(IP_VERSION,IPv6) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8) EQ(IP_PROT,ICMP)</td>
</tr>
<tr>
<td>7</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8) EQ(IP_PROT,UDP) or EQ(IP_VERSION,IPv6) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,8) EQ(IP_PROT,UDP)</td>
</tr>
<tr>
<td>8</td>
<td>EQ(IP_VERSION,IPv4) GE(IP_HDRLEN,5) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,20) EQ(IP_PROT,TCP) or EQ(IP_VERSION,IPv6) EQ(PS_NOORFIRSTFRAG,1) GE(PS_IPDATASIZE,20) EQ(IP_PROT,TCP)</td>
</tr>
</tbody>
</table>

One needs to be careful to understand the semantics of the implied expressions. For instance, it might appear that the following two rules represent an “if-than-else” situation.

<table>
<thead>
<tr>
<th>Rule</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ(TCP_SYN,1) FORWARD(20)</td>
<td></td>
</tr>
<tr>
<td>NE(TCP_SYN,1) FORWARD(21)</td>
<td></td>
</tr>
</tbody>
</table>

However, this is not the case; both rules could evaluate to false because of the implied expressions (e.g., if this is not a TCP packet).

Aside from the implicit checks in non-deferred expressions, the named packet fields are just a shorthand for referring to a location in the current packet. For instance, TCP_FLAGS is one byte at PS_IPDATAOFFSET+13. IP_HOPLIMIT is the same as PFIELD(7). The implementation will always detect an attempt to reference or store beyond the extent of the actual packet and generate an exception upon such an occurrence.

**Dynamic Packet Fields**

Another way to refer to a field in a packet is provided by the following indexed values:

- **PFIELD**, which generally refers to something relative to the start of the IP packet
- **CONTENT**, which generally refers to something relative to the start of the packet payload
- **L2FIELD**, which generally refers to something relative to the start of the layer 2 header
- MFIELD, which refers to something in metadata associated with the packet

The specific notation is

- \( \text{PFIELD}(\text{index})[\text{width}] \)
- \( \text{CONTENT}(\text{index})[\text{width}] \)
- \( \text{L2FIELD}(\text{index})[\text{width}] \)
- \( \text{MFIELD}(\text{index}) \)

Index can be either a constant or a register. It specifies the byte offset of the first byte of the field. Accessing or attempting to store a value beyond the defined bounds of the current packet or its metadata generates an exception.

Width, described in a subsequent section, allows one to denote an 8, 16, 32, or 128-bit field.

The base of L2FIELD is the start of the outermost header associated with the current packet. For instance, if the packet was received over Ethernet, L2FIELD(0) is the first byte of the Ethernet header.

The base of PFIELD is the start of the outermost IP header, or, if not an IP packet, where the IP header would start.

The base of CONTENT is the first data byte (sometimes called segment or chunk) beyond the TCP, UDP, or ICMP header if IP_PROT is one of these. Otherwise, it is the first byte after the IP header if the packet is an IP packet. Otherwise, it is the same as PFIELD.

Note that one can control these positions with the NEWPACKET policy, and one can manipulate the L2FIELD and PFIELD bases in the PACKET policy.

MFIELD is used to address metadata. Metadata exists while a packet is within the PPL virtual machine. It is created when the packet is received (or created), and it disappears when the packet is forwarded outside of the PPL virtual machine or dropped. The PPL virtual machine uses the first 32 bytes of metadata, and its contents are implementation dependent. An implementation may choose to allow one to specify in DeviceMap a larger metadata size per packet, in which case the metadata may be used by the PPL program. MFIELD always implies a field width of 32 bits.

**Packet State Values**

There are several values that specify the state of arrival of the current packet. These are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS_IP</td>
<td>Boolean specifying that the packet is an IP packet.</td>
</tr>
<tr>
<td>PS_TCPUDP</td>
<td>Boolean specifying that the packet is either TCP or UDP protocol. Shorthand for testing if IP_PROT is equal to TCP or UDP.</td>
</tr>
<tr>
<td>PS_FRAGMENT</td>
<td>Boolean specifying that the packet is a fragment. For an IPv4 packet this indicates that either IP_FRAGOFFSET is nonzero or that IP_MF is set. For an IPv6 packet this indicates that a fragment extension header is present.</td>
</tr>
<tr>
<td>PS_NOORFIRSTFRAG</td>
<td>Boolean specifying that the packet is not a fragment or is the first fragment. For an IPv4 packet this indicates that IP_FRAGOFFSET is zero. For an IPv6 packet this indicates that either there is no fragment extension header, or there is one and its fragment offset value is zero.</td>
</tr>
<tr>
<td>PS_EXTENDEDHDR</td>
<td>Boolean specifying that the packet has extended headers. For an IPv4 packet, this indicates that IP_HDRLEN &gt; 5. For an IPv6 packet, this indicates that IP_NEXTHDR has one of the values: 0, 43, 44, 50, 51, or 60.</td>
</tr>
<tr>
<td>PS_IPDATASIZE</td>
<td>Value specifying the number of bytes of data in the packet after the outermost IP with options and extensions. For IPv4 this is IP_PACKETLEN minus 4 times IP_HDRLEN. For IPv6 this is the value IP_PAYLOADLEN minus the size of all extension headers present.</td>
</tr>
<tr>
<td>PS_IPDATAOFFSET</td>
<td>Value specifying the byte offset of the first data byte after the IP header(s) (measured from the start of the IP header).</td>
</tr>
<tr>
<td>PS_CONTENTSIZE</td>
<td>Value currently defined only if a UDP or TCP packet – specifies the content length in bytes not included the TCP or UDP header. This is identical to PS_IPDATASIZE minus the length of the TCP or UDP header.</td>
</tr>
<tr>
<td>PS_CONTENTOFFSET</td>
<td>Value defining the CONTENT base – the byte offset (relative to the start of the IP header) of the first byte beyond the TCP, UDP, or ICMP header, or of the first byte beyond the outermost IP header if IP_PROT is not one of these.</td>
</tr>
<tr>
<td>PS_PACKETHANDLE</td>
<td>Value uniquely identifying the current packet. The language does not define the bit representation.</td>
</tr>
</tbody>
</table>
| PS_LPN          | Value specifying the logical port number on which the packet arrived.  
| PS_VLAN         | Value specifying to which VLAN the packet belongs; FuF if no VLAN. How this is obtained is implementation dependent; an obvious implementation is to assign this the value of the VLAN tag in the layer 2 frame if present. |
| PS_TYPE         | Value specifying the type of the outermost header (that to which L2FIELD refers). See section on NEWPACKET and PACKET policies for encodings. |

If there is no packet, PS_PACKETHANDLE has the value FuF, the Booleans have the value zero, and the other PS_ values have undefined values.

If there is a packet but it is not an IP packet, PS_IP is zero, PS_PACKETHANDLE, PS_LPN, PS_VLAN, and PS_TYPE have their defined values, and the other PS_ values have undefined values.

The PS_ values cannot be destinations.

#### Connection State Values

Another set of values are fields of the current connections-table entry in a CONNECTIONS policy.

---

12 If the packet was forwarded to an event, PS_LPN would represent the logical port number to which the packet was forwarded.
CX_DATA is a 32-bit value. CX_STATE has an implementation-dependent size, although it must at least accommodate values of 0..255. If there is no current connection, CX_STATE has the value 0 and CX_DATA is undefined, and any attempt to use either as a destination causes an invalid-operation exception. See the CONNECTIONS policy for further discussion of CX_STATE.

Array Values

PPL provides a way to define one-dimensional lists of equal-size values, called arrays, and the ability to index into an array to select a value. An array is defined by an ARRAY statement having the syntax:

```
label: ARRAY(size)[width] [INITIAL(list of constants)]
```

The label is the array name. A value is selected by specifying `label(index)`, where index is a constant or register. For instance for the array:

```
Rate_limits: ARRAY(4) INITIAL(10000,20000,30000,40000)
```

the expression `EQ(Re7,Rate_limits(Rr1))` would compare register Re7 to the “Rr1’th” element (starting at 0) of the array. The array above has valid index values 0..3. Attempting to exceed the extent of an array results in an exception.

Size must be a numeric constant of value 1 or greater. Optional width specifies the element size as .b (byte), .h (16 bits), .w (32 bits), or .q (128 bits). If not specified, the default is .w. Therefore the following describes an array of what are perhaps IPv6 addresses:

```
alternate_servers: ARRAY(32).q
```

An indexed value in an array (an array element) takes on implicitly the width of the array. Thus from the above `Rate_limits(4)` is a 32-bit value and `alternate_servers(4)` is a 128-bit value.

An array is initialized in the following ways:

- If INITIAL is not specified, each element is initialized to all 1’s (for 32-bit quantities, FuF)
- If INITIAL is specified and only a single constant is given, each element is initialized to that value
- If INITIAL is specified and N constants are given where N is two or more, those constants initialize the first N elements of the array, and all others are initialized to all 1’s.

The device-dependent DeviceMap statement in PPL provides several optional special controls on arrays, such as allowing one to specify that an array must be located in a specific memory and/or at a specific address (e.g., useful for memory-mapped devices and for data sharing with a control-plane processor).
## Registers

PPL defines 256 long-lifetime registers, multiple sets of 32 short-lifetime registers, each associated with an instance of an event, and multiple sets of four “very-short-lifetime” registers, each associated with an instance of a run group. All registers are 32 bits wide with a provision to refer to consecutive registers as a 128-bit register.

<table>
<thead>
<tr>
<th>Category</th>
<th>Registers</th>
<th>Lifetime</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run group registers</td>
<td>Rr0..Rr3</td>
<td>An event instance or a run group within an event instance</td>
<td>To FuF at start of each event and each run group</td>
</tr>
<tr>
<td>Event registers</td>
<td>Re0..Re31</td>
<td>An event instance</td>
<td>To FuF at start of each event</td>
</tr>
<tr>
<td>Global registers</td>
<td>Rg0..Rg255</td>
<td>Permanent</td>
<td>To FuF upon startup (event 999)</td>
</tr>
</tbody>
</table>

The implementation is free to define the atomicity of operations on registers (behavior if multiple run groups or events simultaneously operate on a register), but all implementations are required to guarantee correct behavior for at least the unmasked ADD action on 32-bit event and global registers.

## Labels

Another value is the set of optional statement labels on policies and rules. Policies within a PPL program are consecutively numbered, with the starting number being arbitrary. Also, all rules have unique numbers, and rules within an event are consecutively numbered. Therefore a label on a policy or rule statement takes on the number of that policy or rule. This arrangement has several uses, such as allowing one to index into policies or rules. E.g.,

```plaintext
Rule SET(Re14, Routetable) SET(Re14, Re14 + IP_TOSPREC) APPLY(Re14)
-----------------------------
Rule SET(Re14, Actionrules) SET(Re14, Re14 + IP_PROT) JUMP(Re14)
```

Array labels have no value.

## Indexed Packet Values and Numerically Named Registers, and Their Widths

Each of these conceptually have the form `name(index)[width]`. For all but registers, `index` must be expressed as a constant value or register value. For registers, the index is built into the name. Referring or attempting to store to something with a negative index or an offset beyond the bounds of the entity generates a compile error (if detectable) or an exception.

For registers, the index is units of 32 bits. For the others, the index is a byte offset, and if multiple bytes are referenced, the lowest-address byte is at the offset.

The manner of explicitly expressing widths is shown below.
PFIELD allows one to refer to data at numeric byte offsets anywhere within the packet. The form of this reference is PFIELD(offset)[width]. Thus PFIELD(0) represents the first byte of the packet; PFIELD(0).w represents the first word of the packet.

CONTENT is similar to PFIELDs except that the offset is relative to the start of the L7 content, not the beginning of the packet. Typically the offset is expressed in a register because layer-7 things are generally not at fixed places. For instance, CONTENT(RE22) refers to a byte at an offset contained in register RE22. Like PFIELD, the offsets are always in units of bytes.

L2FIELD has the same behavior.

The use of .q on a register name allows one to use that register and the next three consecutively numbered registers as a 128-bit value. One cannot use the .q notation on the last three run-group, event, and global registers (e.g., Rg255.q is illegal).

### Operations on Values of Differing Widths

Data widths in PPL range from 1 to 128 bits (e.g., TCP_SYN is one bit in size and Rr0.q is 128 bits). Every value has a defined width; as explained earlier in the discussion of constants, constants are 32 or 128 bits.

When used in an operation, a value conceptually becomes 32 or 128 bits; that is. If smaller than 32 bits it is conceptually expanded to 32 bits by inserting high-order zeros. When an operation is performed (the combining of two values for a comparison, arithmetic, or logical operation), the operation is a 32-bit operation unless one or both values are 128 bits, in which case any 32-bit value is conceptually expanded (with high-order zeros) to 128 bits. When a result is stored, the value is truncated (high-order bits eliminated) if wider than the destination or expanded (with high-order zeros) if shorter than the destination.

```
Rule SET(PFIELD(18),Re2)   # Stores least significant byte of Re2
Rule SET(TCP_FIN,5)        # Sets TCP_FIN bit to 1
Rule . . . EQ(RE4.q,Rr0)   # Does 128-bit comparison using zero-extended Rr0
Rule EQ(IP_SOURCE,X)       # IP_SOURCE width depends on whether packet is
                           # IPv4 or IPv6 - dynamically determined
```

Data is treated as big endian. For instance, if CONTENT(0).w is used as a numerical value, the first byte (CONTENT(0).b) is the most significant byte. Similarly, if Rg4.q is used as a 128-bit numerical value, Rg4 is the most-significant 32 bits and Rg7 the least-significant 32 bits.

---

13 Technically, to the data part after a TCP, UDP, or ICMP layer 4 header.
Actions

The remaining part of a rule is an action(s). The action(s) must appear after (to the right of) the expression(s). Multiple actions are processed in order.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP</td>
<td>The current packet is discarded.</td>
</tr>
<tr>
<td>FORWARD[value, value]</td>
<td>The current packet is forwarded.</td>
</tr>
<tr>
<td>STOP</td>
<td>Processing of the event terminates.</td>
</tr>
<tr>
<td>LOG[value]</td>
<td>The current packet is logged. If a value is present, it is stored with the log entry.</td>
</tr>
<tr>
<td>SET[value, value_exp]</td>
<td>The value_exp, which is a value or simple valued expression, is stored into the container of the first value.</td>
</tr>
<tr>
<td>COMPUTE[function, value, value]</td>
<td>Performs one of a set of computational functions.</td>
</tr>
<tr>
<td>APPLY[value]</td>
<td>The policy whose number is value is applied.</td>
</tr>
<tr>
<td>JUMP[value]</td>
<td>Instead of continuing to the actions of the next rule, transfer to the rule whose number is value.</td>
</tr>
<tr>
<td>UNLOCK[value, value]</td>
<td>Locks a lock, waiting if necessary</td>
</tr>
<tr>
<td>ACT</td>
<td>Null action</td>
</tr>
</tbody>
</table>

DROP

DROP frees up any resources associated with the current packet, and the event no longer possesses a current packet. If there is no current packet, an exception is generated.

FORWARD

FORWARD hands the current packet off to some other entity. Subsequent to performing the action, there is no longer a current packet in the event, and PS_PACKETHANDLE has the value FuF.

FORWARD with no values expressed means to perform IP layer-3 forwarding based on the destination IP address.

When values are present, FORWARD has the form FORWARD(lpn, id). Lpn is a logical port number. Id is an identifier value whose meaning depends on the mapping of lpn. If id is not included, it is assumed 0. Here the semantics of FORWARD are controlled by the implementation-dependent DeviceMap statement. However, if there is no explicit mapping of this lpn in DeviceMap, the packet will be forwarded to a PPL event of the same number if one exists.

An exception is generated if the logical port number cannot be resolved or if it has a value 990..999.

---

14 For clarity here, the Intel IXP2xxx DeviceMap allows one to map a forwarded packet to an Ethernet port, to a destination port and class in a CSIX fabric, to a PCI bus, and to an external program.
STOP

The explicit STOP action, or the implicit act of processing actions up to the next Event statement or to the physical end of the PPL program, all cause the same action – processing of this occurrence of the event is terminated.

Typically, an event that has a current packet should terminate by processing either a DROP or a FORWARD followed by a STOP, and an event with no packet should terminate by processing a STOP. One needs to be cautious when using STOP when there is a current packet because STOP does not free any resources associated with the packet; thus to avoid permanent loss of the resources (memory) one should have forwarded the packet or retained the packet handle for later processing.

LOG

LOG creates a system log entry.\(^\text{15}\)

SET

SET takes a value, or determines the value of a simple valued expression, and stores this into the container of the leftmost value. SET has the form SET(value,value_exp) where value_exp is one of the following

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value is assigned</td>
</tr>
<tr>
<td>value + value</td>
<td>The sum of the two values is assigned</td>
</tr>
<tr>
<td>value - value</td>
<td>The difference of the two values is assigned</td>
</tr>
<tr>
<td>value &gt;&gt; value</td>
<td>The first value is shifted N bits to the right, with inserted bits being 0. N is the second value modulo 32 if the first value is 32 bits or smaller, modulo 128 otherwise.</td>
</tr>
<tr>
<td>value &lt;&lt; value</td>
<td>The first value is shifted N bits to the left, with inserted bits being 0. N is the second value modulo 32 if the first value is 32 bits or smaller, modulo 128 otherwise.</td>
</tr>
<tr>
<td>value &amp; value</td>
<td>The bitwise AND of the two values is assigned</td>
</tr>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>value ^ value</td>
<td>The bitwise XOR of the two values is assigned</td>
</tr>
</tbody>
</table>

Arithmetic is unsigned and modulo the destination size. The destination cannot be a constant or the packet state.

Note that the mask capability on values gives one some ability to do multiple logical operations in one action (e.g., SET(A,B/4 | C/6)). Also note that the NOT function can be obtained by XORing all 1’s to a value.

COMPUTE

COMPUTE has the form COMPUTE(function,value1[,value2]), where function specifies the function to be performed. Depending on the function, one or two values are given. When a

\(^{15}\) The information in log entries and how log entries are acquired for examination is not part of the language definition. However, it is expected that log entries contain part or all of the packet, the event number, the time, and the optional value expressed in LOG. The assumption is that logging is done into a circular buffer.
value is listed as a result (destination), it must follow the rules of destinations ala the SET action.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Other values used</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBN</td>
<td>Convert character numerical value to binary</td>
<td>Result</td>
<td>Content offset</td>
<td></td>
</tr>
<tr>
<td>CEND</td>
<td>Convert to other endian</td>
<td>Result</td>
<td>Value to be converted</td>
<td></td>
</tr>
<tr>
<td>HASH</td>
<td>Hash</td>
<td>Result</td>
<td>Value to be hashed</td>
<td></td>
</tr>
<tr>
<td>RAND</td>
<td>Random number</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>SEED</td>
<td>Seed random number</td>
<td>Seed value</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>Current time</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>UTIM</td>
<td>Microsecond time</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>ICSM</td>
<td>IPv4 checksum</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>TCSM</td>
<td>TCP/UDP checksum</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
<tr>
<td>SCSM</td>
<td>Step checksum</td>
<td>Old checksum and result</td>
<td>Old value</td>
<td>Rr0 is new value</td>
</tr>
<tr>
<td>CC32</td>
<td>Stream CRC-32</td>
<td>Result</td>
<td>Pfield offset</td>
<td></td>
</tr>
<tr>
<td>SC32</td>
<td>Step CRC-32</td>
<td>Initial CRC and result</td>
<td>Value to be incorporated</td>
<td></td>
</tr>
<tr>
<td>CC3C</td>
<td>Stream CRC-32C</td>
<td>Result</td>
<td>Pfield offset</td>
<td></td>
</tr>
<tr>
<td>SC3C</td>
<td>Step CRC-32C</td>
<td>Initial CRC and result</td>
<td>Value to be incorporated</td>
<td></td>
</tr>
<tr>
<td>CONN</td>
<td>Real-time connection state</td>
<td>Result</td>
<td>Not used</td>
<td></td>
</tr>
</tbody>
</table>

CBN (convert to binary) expects a string of one or more digits in the range 0..9 to begin at the offset in the packet content specified by value2, and it converts the value to a binary value and stores it in the destination expressed by value1. The end of the series of one or more digits is defined by a character not “0” .. “9”. The implementation is permitted to put an upper bound (but no less than 10) on the maximum number of digits that will be converted.  

CBN produces a 128-bit result, which follows the normal rules if the destination is 32 bits. If the string at the specified offset is not one or more digits, or if the number of digits exceeds the implementation’s upper bound, CBN produces the 128-bit value FuF00..00, meaning FuF in the most-significant 32 bits and 0 in the least-significant 96 bits.

CEND (convert endian) creates a value by reversing the byte order in the 32- or 128-bit value, and then stores this converted value in the destination expressed by value1.

HASH takes a 32- or 128-bit value and computes a 32- or 128-bit hash, where the algorithm distributes the computed hash uniformly across all the bits of the result. For instance, COMPUTE(HASH,Rr0.q,Rr0.q) takes the 128-bit value in Rr0.q (Rr0-Rr4) and replaces it with a 128-bit result where any subset of the 128 bits would be a high-quality hash-table index.

RAND places a 32-bit pseudo random number in the destination.

SEED stores the 32-bit value as the seed of the pseudo random number generator. The exact meaning of this relative to a guaranteed repeatable random sequence is implementation dependent. However, if only one serial event is seeding and getting random numbers, the

---

16 Current implementation is 16.
implementation shall guarantee that, given a specific seed, the pattern of random numbers is repeatable.

**TIME** returns a 32-bit value representing the approximate number of milliseconds that have transpired.\(^{17}\) This value wraps around when it overflows.

**UTIM** returns a 32-bit value representing a microsecond timer. This value wraps around when it overflows. The value is normally an approximation to microseconds, and implementations should document an adjustment factor.

**ICSM** (IPv4 checksum) returns a value expressing the correct IPv4 checksum. If there is no current packet or the current packet is not IPv4, an invalid-operation exception occurs.\(^{18}\)

**TCSM** (TCP or UDP checksum) returns a value expressing the correct TCP or UDP checksum. If there is no current packet or the current packet is not TCP or UDP protocol, an invalid-operation exception occurs.\(^{19}\) If the packet is IPv6, the destination address in the outer header is used for the pseudo-header calculation (as opposed to an address from a routing header extension).

**SCSM** (step checksum) returns an incremental checksum using the 1’s complement formula expressed in RFC 1624. This function expects a value in Rr0. Given an “old” checksum \((\text{value1})\), an “old” 32-bit value \((\text{value2})\) that is assumed to be aligned on a 16-bit boundary, and a “new” 32-bit value (Rr0), it computes a “new” checksum and stores it in \(\text{value1}\).

**CC32** returns the CRC computed over a specified part of the current packet. The CRC is the CRC-32 polynomial, also known by the code 0x82608EDB. \(\text{Value2}\) specifies the offset of the first byte relative to the start of the IP packet header. The calculation is done 32-bit word by 32-bit word from this point to the end of the packet, logically adding 1-3 bytes of zero to form the last word as needed.

**SC32** returns an incremental CRC. It updates the incremental CRC \(\text{value1}\) using \(\text{value2}\). The CRC is the CRC-32 polynomial (code 0x82608EDB).

**CC3C** is the same as CC32, except that the CRC-32C polynomial is used (code 0x8F6E37A0).\(^{20}\)

**SC3C** is the same as SC32, except that the CRC-32C polynomial is used (code 0x8F6E37A0).

**CONN** returns the real-time connection state value. See the CONNECTIONS policy for a description of this.

---

\(^{17}\) When “ticking” starts is implementation dependent but can generally be assumed to be when the system is started.

\(^{18}\) One can validate the checksum in the packet by using this function and then comparing the result with IP_HDRCSUM.

\(^{19}\) Note that this operation is typically time consuming because it needs to read the whole packet payload. If one wishes to recalculate a checksum as a result of changing specific fields, such as changing a L4 port address and an IP address for NAT, one is advised to compute the revised checksum as a delta in the PPL program explicitly.

\(^{20}\) This is the polynomial suggested for SCTP and iSCSI.
**APPLY**

The value specifies a number of a policy, which is applied. If there is no policy with this number, the invalid-operation exception occurs.

**JUMP**

JUMP is a way of altering the strict sequential processing of actions of the true rules within an event. JUMP specifies the number (often a statement label) of a rule to which processing is moved; the actions of true rules from the target rule onward thus become the next actions processed.

If the value expressed is a nonexistent rule, a rule outside of the event, a rule outside of the run group, or a rule inside a different run group, the invalid-operation exception occurs.\(^{21}\)

**LOCK and UNLOCK**

LOCK\((lock\_number, time\_limit)\) locks the specified lock if it is currently unlocked. If the lock is currently locked, processing of the event is delayed until the lock becomes unlocked, at which time LOCK locks it. The \textit{time\_limit} value, in units of milliseconds, expresses the handling of the delay. If the time limit is exceeded, the time-exceeded exception occurs. If the value 0 is used, it expresses a time limit of approximately 0.25 milliseconds. If the value is FuF, there is no time limit.

UNLOCK\((lock\_number)\) unlocks the specified lock (independent of whether currently locked or unlocked).

Locks are numbered 0..N, where N is implementation dependent, but must be at least 31. Specifying a lock number beyond what locks exist results in the invalid-operation exception.

**ACT**

ACT is a null action; processing it has no effect. It is intended for one circumstance – where one needs to designate an expression as a deferred expression but needs this to precede all other actions on the rule. For instance

```
Rule ACT EQ(Rr0,FuF) APPLY(x)
```

---

\(^{21}\) Unless, of course, the compiler can detect the error and report it.
## Policies

Policies are algorithms or state tracking that is built into the implementation but controllable from the PPL program. Some policies are specific to the underlying implementation.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOCIATE</td>
<td>Defines a table and how entries are created in the table</td>
<td>NUMBER(constant) SEARCHKEYS(list of values) [TIMEOUT(value)]</td>
</tr>
<tr>
<td>RECALL</td>
<td>Looks up an entry in a table</td>
<td>SEARCHKEYS(list of values) LINKED(constant) [ROW(value)] [TIMEOUT(value)]</td>
</tr>
<tr>
<td>DISASSOCIATE</td>
<td>Removes a specific entry in a table or all entries</td>
<td>[SEARCHKEYS(list of values) or ALL or ROW(value)] LINKED(constant)</td>
</tr>
<tr>
<td>CIPHER</td>
<td>Converts a portion of the current packet between cleartext and ciphertext, including the computation of a message digest, authentication code, and checksum</td>
<td>[RELEASE] [ENCRYPT(alg_cnst[,alg_cnst]) LOCATION(base_cnst,ioff_val[,len_val[,ooff_val][]) KEY(value[,value]) [IV(value)] [PAD(type_cnst[,constant])] ] [DECRYPT(alg_cnst[,alg_cnst]) LOCATION(base_cnst,ioff_val[,len_val[,doff_value]) KEY(value[,value]) [IV(value)] ] [HASH(alg_cnst) [DIGEST(use_cnst[,size_cnst[,value[,value]]]) [HMAC(value[,value])] [LOCATION(base_cnst,ioff_val[,len_val])] or [RESIDUE(value,size_val]) or [IPAD(value[,value])] or [OPAD(value[,value])] ]</td>
</tr>
<tr>
<td>CLASSIFY</td>
<td>Searches a multi-field database</td>
<td>[NUMBER(constant) [MODE(1)]] or [DATABASE(external name)] or [LINKED(constant)] DATA(list of values) [SELECT(value)] [ROW(value)] [QUERY] or [SET] or [SETCOMP] or [GET] or [GETCOMP]</td>
</tr>
<tr>
<td>CONNECTIONS</td>
<td>Defines information used to track connection state</td>
<td>NUMBER(constant) EVENTS(list of constants) [PROTOCOL(constant)] [VIRTUALNETWORK(value)] [SOURCEALIAS(value[,value])] [DESTALIAS(value[,value])] [STATE(constant,constant,constant,constant,constant)…]</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Parameters</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>CONTROL</strong></td>
<td>Controls event processing</td>
<td>ID(value), SIZE(value), DATA(list of values)</td>
</tr>
<tr>
<td><strong>DEFRAG</strong></td>
<td>Assembles fragments into nonfragmented packets</td>
<td>NUMBER(constant), SIZE(constant), MAXPACKETS(constant), MODE(constant[,constant]), TIMEOUT(value[,value]), VIRTUALNETWORK(value)</td>
</tr>
<tr>
<td><strong>NEW PACKET</strong></td>
<td>Creates a new packet</td>
<td>SIZE(value), HEADERS(poff_value,l4off_value,coff_value), INITIALIZE(string), ENCAPSULATE, PREPEND</td>
</tr>
<tr>
<td><strong>PACKAGE</strong></td>
<td>Performs functions on the current packet and/or a specified packet</td>
<td>HANDLE(value) DROP, HANDLE(value) FORWARD[value[,value]], HANDLE(value) CURRENT, HANDLE(value) TYPE(value), HANDLE(value) COPY(toff_value,soff_value,size_value), INSERT(method,size_value,ip_pos_value), STRIP(method,size_value,ip_pos_value)</td>
</tr>
<tr>
<td><strong>PATTERNS</strong></td>
<td>Does a directed search through a patterns database</td>
<td>DATABASE(external name), SCAN(base_cnst[,destination[,svalue[,dvalue]]]), MODE(mode_val,tag_val) or LPM(value[,qual_val]), LINKED(constant) ID(value)</td>
</tr>
<tr>
<td><strong>PROGRAM</strong></td>
<td>Signals or invokes a function outside of the scope of PPL</td>
<td>FUNCTION(external name), DATA(list of values)</td>
</tr>
<tr>
<td><strong>QUEUES</strong></td>
<td>Defines a set of queues</td>
<td>NUMBER(constant), WEIGHT, EVENT(constant), MODE(mode_cnst,lpn_val,time_val[,tspread_val])</td>
</tr>
<tr>
<td><strong>QUEUE</strong></td>
<td>Performs an operation on a queue</td>
<td>LINKED(constant), NUMBER(value), ADD(constant), HANDLE(value), DATA(value), REMOVE(constant), HANDLE(value), DATA(value), QUERY(value)</td>
</tr>
<tr>
<td><strong>RATE</strong></td>
<td>Maintains a time-based rate</td>
<td>TIMEBASE(value), COUNTING(value), RESETTIME(value), COUNTED(value) STARTED(value)</td>
</tr>
<tr>
<td><strong>NEWSUPERPACKET</strong></td>
<td>Defines a database of superpackets and creates a superpacket</td>
<td>NUMBER(constant), MAXPACKETS(constant), ID(value), TIMEOUT(value[,value])</td>
</tr>
<tr>
<td>SUPER-PACKET</td>
<td>Performs an operation on a superpacket</td>
<td>LINKED(constant)</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ID(value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ADD(value,value,value,value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DELETE((value))]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[QUERY(value,value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SCANB(string,[svalue,[dvalue]])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SCANE(string,[svalue,[dvalue]])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[FORWARD([value,[value]])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DROP]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SETFIELD(value,(value[[w][w]],value))]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[WHICH PACKET(dest,offsetvalue)]</td>
</tr>
</tbody>
</table>

All times are expressed in milliseconds. This gives one a range of approximately 0.001 seconds to 45 days.

Individual policies are discussed in subsequent sections as part of explaining major functions of PPL programs.
Association Policies

Associations are associative table structures and entail the ASSOCIATE, RECALL, and DISASSOCIATE policies and applying them in rules. The ASSOCIATE policy allows one to create a table and store associatively searched information in the table. The RECALL policy allows one to make an association, that is, look up a table entry. So, the ASSOCIATE policy defines the table, applying the ASSOCIATE policy creates a new table entry if one doesn’t already exist, and applying the RECALL policy retrieves data from a table entry. DISASSOCIATE destroys a table entry and marks it as free for use.

The policies function with both content-based lookups as well as specific row numbers. The latter allows associations to be coupled with indexed arrays to create arbitrarily large and complex databases.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOCIATE</td>
<td>Defines a table and how entries are created in the table</td>
<td>NUMBER(constant) SEARCHKEYS(list of values) [TIMEOUT(value)]</td>
</tr>
<tr>
<td>RECALL</td>
<td>Looks up an entry in a table</td>
<td>SEARCHKEYS(list of values) LINKED(constant) [ROW(value)] [TIMEOUT(value)]</td>
</tr>
<tr>
<td>DISASSOCIATE</td>
<td>Removes a specific entry in a table or all entries</td>
<td>[SEARCHKEYS(list of values) or ALL or ROW(value)] LINKED(constant)</td>
</tr>
</tbody>
</table>

The ASSOCIATE policy has the form

```
Label: Policy ASSOCIATE NUMBER(size) SEARCHKEYS(list of values) TIMEOUT(type,time_interval)
```

This creates a table of at least size entries, each having the width of the set of search keys. When one applies this policy, if no entry exists matching the search keys, it creates one at an arbitrary free spot and returns its row number; if the entry exists, it just returns its row number. If no entry exists, it creates one and stores the data values. RR0 is given the value FuF if the apply of an ASSOCIATE policy failed because no space is available; otherwise it is the row number assigned.

As an example, if we wanted to create a table containing a unique entry for every incoming IP address / TCP port and destination IP address / TCP port we could write

```
trafficpairs: Policy ASSOCIATE NUMBER(10000) SEARCHKEYS(IP_SOURCE,L4_SPORT,IP_DEST,L4_DPORT)

Rule EQ(TCP_SYN,1) APPLY(trafficpairs)
```

---

22 The implementation is permitted to round this up – e.g., to a power-of-two value.
23 It is presumed that there is some way outside of PPL (e.g., via SNMP and/or the control plane) to get at this data.
If we wanted to count the packets associated with these traffic pairs we would write

```
Define max_track = "10000"

trafficpairs: Policy ASSOCIATE NUMBER(max_track)
SEARCHKEYS(IP_SOURCE,L4_SPORT,IP_DEST,L4_DPORT)

Trafficpairlookup: Policy RECALL LINKED(trafficpairs)
SEARCHKEYS(IP_SOURCE,L4_SPORT,IP_DEST,L4_DPORT)

Trafficcount: ARRAY(max_track)

Rule APPLY(trafficpairlookup) NE(RR0,FuF)
  SET(Trafficcount(Rr0),Trafficcount(Rr0) + 1)
  # This searches the table, setting RR0 not FuF if an entry exists

Rule ACT EQ(RR0,FuF) APPLY(trafficpairs) SET(Trafficcount(Rr0),1)
```

As another example, up at layer 7, suppose we wish to watch for all SIP SUBSCRIBE requests (methods) to SIP request URI's and record the URI and the number of times we have seen it. If the count of times exceeds 100, we will drop the packet. To keep things simple here, we will assume well-behaved SUBSCRIBE requests that start the "SUBSCRIBE sip:" and that it is sufficient to capture the first 8 characters of the URI.

```
Define ac="RR0"
Define goturi="RE0"
Define scanresult="RE1"
Define URIFOUND="NE(goturi,FuF)"
Define URINOTFOUND="EQ(goturi,Fu)"
Define GOTSUBSCRIBE="NE(scanresult,FuF)"

uri_instances: ARRAY(5000).

subscribes: POLICY ASSOCIATE NUMBER(5000) SEARCHKEYS(RE20,RE21)
TIMEOUT(50000)

subscribed: POLICY RECALL LINKED(subscribes) SEARCHKEYS(RE20,RE21)
TIMEOUT(50000)

dropsub: POLICY DISASSOCIATE LINKED(subscribes) SEARCHKEYS(RE20,RE21)

Rule SCANE("SUBSCRIBE sip:",0,0) SET(scanresult,ac) SET(RE20,CONTENT(ac).w)
  SET(ac,ac + 4) SET(RE21,CONTENT(ac).w)
  APPLY(subscribed) SET(goturi,ac)
  #Above first see if the entry exists. If not, goturi will be FuF.

Rule ACT GOTSUBSCRIBE URIFOUND GE(uri_instances(goturi),99) DROP STOP
Rule ACT GOTSUBSCRIBE URIFOUND LE(uri_instances(goturi),99)
  SET(uri_instances(goturi),uri_instances(goturi) + 1)

Rule ACT GOTSUBSCRIBE URINOTFOUND APPLY(subscribes) SET(uri_instances(ac),1)
```

The relationships between ASSOCIATE and RECALL are shown in the table below. RECALL looks up an association. If ROW is not specified, a search occurs and the matching row number or FuF is returned. If ROW is specified, the behavior is quite different; if a valid entry (created by ASSOCIATE) exists in that row, the searchkey fields are retrieved from the row. Thus if ROW is specified, the SEARCHKEYS list must be a list of valid destinations. If the specified row contains no association, Rr0 is returned with the value FuF. If the row is beyond the defined size of the association, an exception occurs.

---

24 We could use a regular expression to scan for a less-well-behaved request.
DISASSOCIATE operates similarly. One has three options: remove an association by search keys, remove an association in a specific row, or remove all associations. If DISASSOCIATE is done by row, it returns FuF in RR0 if the row was already free. Similarly, if it is done by search keys, it returns FuF if the entry was not found.

How these policies are implemented, specifically the searching, is not specified by PPL, but one does have to be aware of some likely implementations and the consequences. A likely implementation is a hash table, and most hash techniques either get extremely slow as the table fills up, and/or can’t utilize every entry in the table. Therefore the usual advice is to allow considerably more space than the maximum number of entries that will exist; for instance, if one feels the maximum number of valid entries will be N, perhaps allocate 2N entries as the table size. However, there are some cases where the number of entries must be specific, as in the example below.

Let’s illustrate this capability to implement the form of NAT called NAPT (Network Address Port Translation). NAPT (called “hide mode” by some) uses a pool of dynamically assignable port numbers to hide the internal agents behind one external IP address. Thus on outbound traffic, the “real” source IP address and source port is remembered in a table, and the source IP address is changed to VIP and the source port to a dynamically assigned one from which we can recall the original real addresses. On the inbound traffic, if we have a packet destined to VIP and one of our dynamically assigned port, we performed the reverse translation. As one firewall vendor does, in this example we will reserve ports 600-1023 and 10000-60000 as our pool. This means a table of size 50425 (or two tables totaling this, which is what we have done). A potential problem in NAPT is knowing when to release an allocated port. We will follow the common convention of releasing entries that go unused for about 15 minutes.

```plaintext
Define fifteen_minutes = “900000”
Define IDLE = “0”
Define SVIP = “IPLIST(0)”
Define lower_NAPT_port_low = “600”
Define lower_NAPT_port_top = “1023”
Define upper_NAPT_port_low = “10000”
Define upper_NAPT_port_top = “60000”
Define ac = “RR0”
LowNAPTtable: Policy ASSOCIATE NUMBER(424) TIMEOUT(fifteen_minutes)
                     SEARCHKEYS(IP_SOURCE, L4_SPORT)
LowNAPTxlate: Policy RECALL LINKED(LowNAPTtable) ROW(Re10)
```
Event (outgoing)

Rule NE(PS_TCPUDP, 1)                   FORWARD (outside) STOP # if not TCP/UDP
Rule GE(L4_SPORT, lower_NAPT_port_top)  JUMP (Use_high_NAPT)
Rule ACT EQ(ac, FuF)                      JUMP (trouble_tables_full)
    # If not there, we create it; if there found it. Either case, RR0(ac) is row
Rule SET(ac, ac + lower_NABT_port_low)  SET(IP_SOURCE, SVIP)
    SET(L4_SPORT, ac)             FORWARD STOP

Use_high_NABT:
Rule ACT EQ(ac, FuF)                      JUMP (trouble_tables_full)
Rule SET(ac, ac + upper_NABT_port_low)  SET(IP_SOURCE, SVIP)
    SET(L4_SPORT, ac)             FORWARD STOP

Event (incoming)

Rule NE(PS_TCPUDP)                     FORWARD STOP # if not TCP or UDP
Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, lower_NABT_port_low)
    LE(L4_DPORT, lower_NABT_port_top)
        SET(RE10, L4_DPORT - lower_NABT_port_low)
        APPLY (LowNAPTxlate)

Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, lower_NABT_port_low)
    LE(L4_DPORT, lower_NABT_port_top)
    ACT EQ(ac, FuF)                      DROP STOP # Using a reserved port
Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, lower_NABT_port_low)
    LE(L4_DPORT, lower_NABT_port_top)  SET(IP_DEST, RE11)  SET(L4_DPORT, RE12)
        FORWARD STOP

Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, upper_NABT_port_low)
    LE(L4_DPORT, upper_NABT_port_top)
        SET(RE10, L4_DPORT - upper_NABT_port_low)
        APPLY (HighNAPTxlate)

Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, upper_NABT_port_low)
    LE(L4_DPORT, upper_NABT_port_top)
    ACT EQ(c, FuF)                      DROP STOP # Using a reserved port
Rule EQ(IP_DEST, SVIP)
    GE(L4_DPORT, upper_NABT_port_low)
    LE(L4_DPORT, upper_NABT_port_top)  SET(IP_DEST, RE11)  SET(L4_DPORT, RE12)
        FORWARD STOP

Timeout
ASSOCIATE and RECALL have optional timeout parameters. The expressed timeout can be a value (i.e., not limited to a constant) and the value FuF has special significance.

If no timeouts are expressed on an ASSOCIATE policy and linked recall policies, there is no timeout. If one or more timeouts are expressed, they apply to the specific entry (i.e., entries stored or recalled can have independent timeouts). If a TIMEOUT parameter is expressed on a RECALL policy, a TIMEOUT parameter must also be expressed on the linked ASSOCIATE policy. Timeout values specify the approximate number of milliseconds until a timeout should occur (i.e., meaning that the entry should be implicitly freed).

Usage of the timeout value is explained in the table below. One way to use the timeouts is for the timeout on the ASSOCIATE policy to become the absolute timeout and the timeout on a linked RECALL to become the idle timeout.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Entry already in table?</th>
<th>TIMEOUT parameter present?</th>
<th>Timeout value FuF?</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOCIATE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No timeouts at all for this association table</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Set specified timeout for new entry</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Set maximum timeout for new entry</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Set new timeout for this entry</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Don’t change current timeout for this entry</td>
</tr>
<tr>
<td>RECALL</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Don’t change timeout for this entry</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Set new timeout for this entry</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Don’t change timeout for this entry</td>
</tr>
</tbody>
</table>
CIPHER Policy

The CIPHER policy allows one to encrypt and decrypt a portion of the current packet in a manner independent of any specific protocol, such as IPSec, SSL, TLS, RTP, HAIPE, etc. Using the encryption and digest algorithms provided, one can implement arbitrary protocols in PPL by making use of the CIPHER policy.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIPHER</td>
<td>Converts a portion of the current packet between cleartext and ciphertext, including the computation of a message digest, authentication code, and checksum</td>
<td>[RELEASE] [ENCRYPT(alg_cnst[,alg_cnst]) LOCATION(base_cnst,ioff_val[,len_val[,ooff_val]])] KEY(value[,value]) [IV(value)] [PAD(type_cnst[,constant])] ] [DECRYPT(alg_cnst[,alg_cnst]) LOCATION(base_cnst,ioff_val[,len_val[,doff_value]])] KEY(value[,value]) [IV(value)] ] [HASH(alg_cnst) DIGEST(use_cnst[,size_cnst[,value[,value]]]) [HMAC(value[,value])] [LOCATION(base_cnst,ioff_val[,len_val]) or RESIDUE(value,size_val) or IPAD(value[,value]) or OPAD(value[,value])] ]</td>
</tr>
</tbody>
</table>

When used for hashing, CIPHER is a stateful policy, but stateful only within the instance of an event using it, meaning that the state is associated with the combination of the policy and the event instance. When the event ends, the state disappears. The state maintained by CIPHER is (1) a message digest that may evolve over successive APPLY’s of the CIPHER policy and (2) the number of bytes of data represented in the digest.

CIPHER defines three functions. One CIPHER policy can express one and only one function. The functions, denoted by the names of their first parameters, are

- **ENCRYPT** Converts a portion of the current packet from cleartext to ciphertext and deposits the ciphertext at the same or a different location in the current packet. Also accumulates the cleartext or ciphertext in a message digest.
- **DECRYPT** Converts a portion of the current packet from ciphertext to cleartext and deposits the cleartext at the same or a different location in the current packet. Also accumulates the cleartext or ciphertext in a message digest.
- **HASH** Accumulates data into a message digest.

ENCRYPT and DECRYPT also return a checksum of the ciphered text.

The RELEASE keyword is an optional optimization that can be used with any of the functions.
ENCRIPT and DECRYPT

The encrypt and decrypt functions are nearly identical and will be described together. They take a specified section of the current packet and apply a specified cipher algorithm and a key to it, returning the result to the same or different section of the packet. They can also accumulate the original or converted text into a message digest using a specified hash.

The keywords are defined below.

ENCRIPT names one or two algorithms, with the order being significant. One must be a cipher algorithms denoted as 3DES, AES, AES192, and AES256\(^{25}\). The other value, which is optional, can denote a hash algorithm as MD5 and SHA1 or can denote AES counter mode as AESCTR. If cipher precedes hash (e.g., ENCRYPT(AES,SHA1)), it means that the cipher occurs first and the hash is on the cipher output. If reversed, it means the hash occurs first. If only a cipher is given, no data is accumulated in the hash. When AESCTR is used, it can only be used as ENCRYPT(AES,AESCTR), ENCRYPT(AES196,AESCTR), or ENCRYPT(AES256,AESCTR).

DECRYPT same as ENCRYPT except that AESCTR is not permitted.

KEY specifies the key to be used in the cipher algorithm. The first value is the first 128 bits of the key. The second value, if present, expresses up to 128 additional bits of the key (refer to table below).

IV optionally specifies an initialization vector or a counter block. When AESCTR is not present, the presence of IV denotes that the cipher is performed with CBC (cipher block chaining) instead of ECB (electronic codebook). The value is interpreted as a 128-bit value; the number of bits actually used depend on the algorithm as defined in the table below. When AESCTR is present, IV must be present and is interpreted as a 128-bit value used as the initial counter block.

LOCATION describes the location of the input and output text, in the form:

\[
\text{LOCATION}(\text{CONTENT}, \text{input_offset}[\text{, length[\text{, output_offset}]]})
\]

\[
\text{LOCATION}(\text{PFIELD}, \text{input_offset}[\text{, length[\text{, output_offset}]]})
\]

The first value is a constant named CONTENT or PFIELD; it denotes whether the other values are relative to the base of CONTENT or PFIELD of the current packet. The second value is the offset of the input text (cleartext in the case of ENCRYPT, ciphertext in the case of DECRYPT). The third value, if present, is the number of bytes of input text. If omitted or FuF, its value is interpreted as the length from the start of the input text to the end of the packet. For DECRYPT, the number of bytes must be an integral multiple of the cipher algorithm’s block size. The last value, if present, denotes the offset of the output text; if not present or FuF, the output text begins at the offset of the input text. \(^{26}\)

PAD (with ENCRYPT only; required if not AESCTR; must be absent with AESCTR.) Specifies how the text should be padded as processed, either when its length is not an integral multiple of the cipher algorithm’s block size, or always. The first value defines the type of pad values created, valid values being

- RAND random byte values
- SEQ sequential byte values starting at 1 (i.e., 1, 2, 3, …)

---

\(^{25}\) DES can be done by repeating the 64-bit DES key three times and using 3DES.

\(^{26}\) Overlap of the input text space and output text space is guaranteed to behave as expected only if the output space starts at or before the input space.
LEN  N values of the form 01 or 0202 or 030303, ...
The second value describes in a broader way how padding is added, and we
will return to this shortly.

The number of bits of significance is dependent on the cipher specified, as shown in the table
below.

<table>
<thead>
<tr>
<th></th>
<th>KEY 1\textsuperscript{st} Value</th>
<th>KEY 2\textsuperscript{nd} Value</th>
<th>IV</th>
<th>Block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DES</td>
<td>Used as high-order 128 bits of 3DES key</td>
<td>High-order 64 bits used as last 64-bits of 192-bit 3DES key</td>
<td>Low-order 64 bits used as 3DES CBC IV</td>
<td>8 bytes</td>
</tr>
<tr>
<td>AES</td>
<td>Used as 128-bit AES key</td>
<td>Not used</td>
<td>128-bit value used as AES CBC IV or AES CTR counter block</td>
<td>16 bytes</td>
</tr>
<tr>
<td>AES192</td>
<td>Used as high-order 128 bits of AES key</td>
<td>High-order 64 bits used as last 64 bits of 192-bit AES key</td>
<td>128-bit value used as AES CBC IV or AES CTR counter block</td>
<td>16 bytes</td>
</tr>
<tr>
<td>AES256</td>
<td>Used as high-order 128 bits of AES key</td>
<td>Used as last 128 bits of 256-bit AES key</td>
<td>128-bit value used as AES CBC IV or AES CTR counter block</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>

For encryption, the PAD parameter can be used to logically add 0 to 9 (for 3DES) or 0 to 17 (for all AES) bytes to the end of the input text. This is done automatically by the encryption function in the manner of the table below. Note that N is calculated such that the last byte added makes the total number of bytes being encrypted be a multiple of the block size.

<table>
<thead>
<tr>
<th></th>
<th>3DES</th>
<th>All AES</th>
<th>Hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD(type)</td>
<td>Append N = 0-7 bytes of type</td>
<td>Append N = 0-15 bytes of type</td>
<td>Used for SSL</td>
</tr>
<tr>
<td>PAD(type,255)</td>
<td>Append N = 0-7 bytes of type, then one byte of value N</td>
<td>Append N = 0-15 bytes of type, then one byte of value N</td>
<td>Used for SSL</td>
</tr>
<tr>
<td>PAD(type,254)</td>
<td>Append N = 0-7 bytes of type, then one byte of value N, then one byte of value IP_PROT</td>
<td>Append N = 0-15 bytes of type, then one byte of value N, then one byte of value IP_PROT</td>
<td>Used for IPSec transport mode</td>
</tr>
<tr>
<td>PAD(type,253)</td>
<td>Reserved</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>PAD(type,0)</td>
<td>Appendix N = 0-7 bytes of type, then one byte of value 0.252</td>
<td>Appendix N = 0-15 bytes of type, then one byte of value 0.252</td>
<td>Used for IPSec tunnel mode</td>
</tr>
</tbody>
</table>

To illustrate, suppose we are using 3DES, we specify PAD(SEQ,254), and our input text is seven bytes of FF. CIPHER would logically insert nine bytes at the end, such that the value being ciphered is

```
FF FF FF FF
FF FF FF 01
02 03 04 05
06 07 07 ZZ
```

where ZZ is the value of IP_PROT

This means that the output text could be up to 17 bytes (for AES) larger than the input text. If the output text could extend past the end of the packet, either for this reason (encryption only) or for the reason that the destination location is beyond the source location, CIPHER will extend the end of the packet in the same manner as the INSERT(END function of the PACKET policy, if there is memory space available. If there isn’t sufficient space, an extent exception occurs.

For decryption, the input text (whose length is expressed explicitly or implicitly) must be an integral multiple of the cipher’s block size. If it isn’t, an invalid-policy-value exception occurs.
For 3DES, the same key is used for encryption and description. For AES, the decryption key must be obtained by an expansion of the encryption key. The relationship between the expanded key and the key values provided for DECRYPT are shown in the table below. For instance, for the 128-bit AES, the values used are the last four words of the expanded key, which are words 40-43 (or labeled EK40..EK43 below). These need to be provided as the decryption key, with the words ordered as shown in the table.\(^{27}\)

<table>
<thead>
<tr>
<th></th>
<th>KEY 1 st value</th>
<th>KEY 2 nd value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>EK3</td>
<td>Word 3</td>
</tr>
<tr>
<td>AES</td>
<td>EK2</td>
<td>Word 2</td>
</tr>
<tr>
<td>AES</td>
<td>EK1</td>
<td>Word 1</td>
</tr>
<tr>
<td>AES</td>
<td>EK0</td>
<td>Word 0</td>
</tr>
</tbody>
</table>

Upon completion of ENCRYPT and DECRYPT, two values are returned in Rr0:

- Bits 0-15 contain a TCP/UDP-compatible checksum of the output text (including any padding added).
- Bits 16-23 contain a value denoting the number of bytes added based on the PAD keyword. Depending on the cipher type, the PAD keyword, and the text length, this can be a value from 0 to 17.

**HASH**

The hash function allows one to accumulate additional data into the hash, or to compute a hash without encrypting or decrypting. For instance, different security protocols require that the message digest include additional data, sometimes from the packet, sometimes from other sources. The first keyword is

**HASH** names the hash algorithm (MD5 or SHA1).

The combinations of keywords allowed and their interpretation is largely dictated by the DIGEST keyword, as shown in the table below. In all situations, zero or one of LOCATION, RESIDUE, IPAD, or OPAD can be specified, and if specified represent the data to be accumulated.

\(^{27}\) Typically, when one obtains an AES key, one would also derive the decryption key, save it in the order described in the table, and then have both keys available for use.
<table>
<thead>
<tr>
<th>DIGEST keyword</th>
<th>HMAC present?</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not present</td>
<td>Cannot be present</td>
<td>Data specified is accumulated in the message digest</td>
</tr>
<tr>
<td>DIGEST(IN,size_cnst,value[,value])</td>
<td>Cannot be present</td>
<td>Message digest is initialized to DIGEST values. The constant denotes how many bytes this digest represents. Data specified is accumulated in the message digest</td>
</tr>
<tr>
<td>DIGEST(OUT,value[,value])</td>
<td>No</td>
<td>Data specified is accumulated in the message digest. The hash is completed and the hash digest is stored in the DIGEST values.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Yes</td>
<td>Data specified is accumulated in the message digest. The hash is completed and the HMAC digest is stored in the DIGEST values.</td>
</tr>
<tr>
<td>DIGEST(INOUT,size_cnst,value[,value])</td>
<td>No</td>
<td>Message digest is initialized to DIGEST values. The constant denotes how many bytes this digest represents. Data specified is accumulated in the message digest. The hash is completed and the hash digest is stored in the DIGEST values.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Yes</td>
<td>Message digest is initialized to DIGEST values. Data specified is accumulated in the message digest. The hash is completed and the HMAC digest is stored in the DIGEST values.</td>
</tr>
<tr>
<td>DIGEST(RESET)</td>
<td>Cannot be present</td>
<td>Message digest is reset[20]. Data specified is accumulated in the message digest.</td>
</tr>
<tr>
<td>DIGEST(RESET,value[,value])</td>
<td>Cannot be present</td>
<td>Message digest is reset. Data specified is accumulated in the message digest. The hash is completed and the hash digest is stored in the DIGEST values.</td>
</tr>
</tbody>
</table>

DIGEST describes how the digest is initialized (if at all), how many bytes of data are represented by the initial digest, whether the final digest should be computed, and to where the final digest is to be returned. The constant, when present, is the number of bytes of hashed text the digest represents; this value is used for only the calculation of the length value in the padding. The constant must be 0..64. For MD5, the next value expresses a 128-bit digest value. For SHA1, the value expresses the high-order 128-bit value of the digest and the last value expresses the low-order 32 bits of the 160-bit digest. When a digest value is returned, MD5 produces a 128-bit digest value, which is stored in the location of the first value. SHA1 produces a 160-bit digest. The value of the high-order 128 bits are stored in the location of the first value, and the value of the lower 32 bits are stored in the location of the second value.

HMAC specifies that an HMAC instead of a hash digest is to be returned. The values of the keyword express the precomputed 128-bit (MD5) or 160-bit (SHA1) opad digest.

At most one of the following keywords can be expressed, and if present it specifies data to be accumulated in the message digest.

LOCATION describes the location of the input and output text, in the form:

LOCATION(CONTENT,input_offset[,length])
LOCATION(PFIELD,input_offset[,length])

The first value is a constant named CONTENT or PFIELD; it denotes whether the other values are relative to the base of CONTENT or PFIELD of the current packet. The second value is the offset of the input text. The third value, if

\[20\] The reset value is specific to the algorithm.
present, is the number of bytes of input text. If omitted or FuF, its value is interpreted as the length from the start of the input text to the end of the packet.

**RESIDUE** specifies a value and its length. The first value is one to 16 bytes to be accumulated in the hash. The second value expresses the number of least-significant bytes of the first value (0-16). If the second value is greater than 16, an invalid policy value exception occurs.

**IPAD** specifies that an inner HMAC calculation is to be performed, using the value(s) representing the key, and accumulated in the hash. For MD5, the first value is the 128-bit key. For SHA1, the first value is the most-significant 128 bits and the second value is the remaining 32 bits of the 160-bit key. The function exclusive-or’s the key with 0x36 byte values (ipad), expands the key to 64 bytes by appending sufficient 0x36 (ipad) bytes, and accumulates the 64 bytes into the hash.\(^{29}\)

**OPAD** specifies that an outer HMAC calculation is to be performed.\(^{30}\) The operation is identical to IPAD expect that the value 0x5C is used.

When the hash or HMAC is to complete, padding (not to be confused with IPAD and OPAD) is accumulated, except if IPAD or OPAD are used. Both hash algorithms are defined over integral multiples of 64 bytes. For SHA1, the hash function automatically accumulates a byte valued 0x80, followed by 4..N bytes of zero, followed by a 4-byte value denoting the length of the text (without padding) in bits. MD5 padding is similar except the 0x80 is followed by 0..N bytes of zero, followed by a 4-byte value denoting the length of the text (without padding) in bits, followed by 4 bytes of zero.

If IPAD or OPAD are used, the assumption is that they are being used for precalculation of text-independent parts of the HMAC calculation, so the digest is produced without the SHA1 or MD5 padding.

**RELEASE**

Depending on the implementation, the CIPHER policy may need to compete for and allocate cryptographic hardware. If this is the case, the first CIPHER policy applied during an event will allocate the resource, and the event will retain this resource until completion of the event. If the PPL program wishes to free such resource before the completion of the event, it can specify the RELEASE keyword, which causes the policy to release the resource at its completion.

**AES Counter Mode Encryption**

AES counter mode has a number of differences deserving special note. Rather than ciphering the cleartext with AES, it ciphers an incrementing counter block with AES and then exclusive-OR’s this into the cleartext. Thus it converts the AES block cipher into a stream cipher, which is why the PAD keyword is not used with AES-CTR; padding is not required by the cipher itself, and thus any padding needed must be handled explicitly by the PPL program.

\(^{29}\) Refer to the chapter on implementation restrictions for a minor restriction on IPAD usage in the IXP2850.

\(^{30}\) And, technically, accumulated in the hash, although it is likely an error to be thinking this way. OPAD is intended for precomputing a digest for the outer HMAC for later use.
The input text and output text must be at the same position in the packet, or an integral multiple of eight bytes apart. No checksum is returned when using AES counter mode.

The 128-bit IV value is used as the initial value of the counter block. To use AES-CTR with high confidentiality, the initial counter block value must be carefully selected.\(^3\)

Because the cleartext payload does not pass through the ciphering mechanism, the policy does not provide a way to concurrently cipher and hash. For AES-CTR, one must do hashing of the cleartext or ciphertext as a separate step.

Decryption is not provided because one decrypts AES-CTR ciphertext by encrypting it again.

**Examples**

As an example, let’s assume we are using a proprietary protocol where we provide encryption and authentication of the payload of a TCP/IP packet. The MAC is computed over the cleartext. The TCP destination port value is used to look up the keys in an array. Using AES and SHA-1, we want to decrypt a packet in place and do an authentication test.

The packet format is below, where the IV and MAC is the last 20 bytes of the packet (we use a truncated 128-bit MAC).

<table>
<thead>
<tr>
<th>IP header</th>
<th>TCP header</th>
<th>Ciphertext</th>
<th>MAC</th>
<th>IV</th>
</tr>
</thead>
</table>

createopad: Policy CIPHER HASH(SHA1) OPAD(Re0.q,Re4) DIGEST(RESET,Re8.q,Re7)  
innermac: Policy CIPHER HASH(SHA1) IPAD(Re0.q,Re4) DIGEST(RESET)  
decipher: Policy CIPHER DECRYPT(AES,SHA1) KEY(Re0.q) IV(Rr0) LOCATION(CONTENT,0,Rr1)  
dehmac: Policy CIPHER HASH(SHA1) HMAC(Re8.q,Re7) DIGEST(OUT,Re0.q,Re4) RELEASE  
keys: ARRAY(65536).q  
keyexts: ARRAY(65536).w  

Rule  
SET(Re0.q,keys(L4_DPORT))  
APPLY(createopad)  
SET(Rr1,PS_CONTENT_SIZE - 4)  
SET(Rr1,Rr1 - 16)  
APPLY(decipher)  
Rule EQ(Re0.q,CONTENT(Rr1).q) . . .  
# HMAC authenticates

Another example is RTP encryption per RFC 3550. This is even simpler because there is no authentication.

---

\(^3\) With the high-order 32 bits being a nonce, the middle 64 bits being an initialization vector, and the low-order bits being an initial counter value. E.g., see RFC3686 and NIST Special Publication 800-38A.
The following is an example of using the CIPHER policy to encode the content of the current packet as an encrypted SSL record. It assumes we are using the SSL or TLS cipher suite TLS_RSA_WITH_3DES_EDE_CBC_SHA. We assume we have determined the key and IV values outside of this example, that we have precomputed the OPAD digest, and that we have an SSL sequence number counter to deter replay and reordering attacks.

```plaintext
Define ssl3_rt_application_data = “23030100”
Define tls1_version = “0x03010000”
Define ssl_rec_hdr_len = “5”
Define sslseqnum = “Rg0.q”  # SSL record sequence number
# Key is in Re12-16, IV in Re8.q, precomputed ipad digest in Re20.q,Re19
# precomputed opad digest in Re24-28

Enlarge_packet: Policy PACKET INSERT(END,25,0)
SSL_ipad:       Policy CIPHER HASH(SHA1) DIGEST(IN,64,Re20.q,Re19)
SHA_adders:     Policy CIPHER HASH(SHA1) RESIDUE(Re4.q,13)
Content_hash:   Policy CIPHER HASH(SHA1) LOCATION(CONTENT,0)
Hash_HMAC:      Policy CIPHER HASH(SHA1) HMAC(Re24.q,Re28)
                 DIGEST(OUT,Re0.q,Re4)
Ssl_encrypt:    Policy CIPHER ENCRYPT(3DES) IV(Re7) KEY(Re0.q,Re4)
                 LOCATION(CONTENT,0,Rr1,ssl_rec_hdr_len)
                 PAD(LEN,255) RELEASE

# Make packet bigger for 5-byte SSL header and 20-byte MAC
APPLY(enlarge_packet)
# Start hash with the precomputed inner hmac digest
APPLY(SSL_ipad)
# Build residue for SHA according to TLS spec (build in Re4.q - Re4-Re7)
SET(sslseqnum,sslseqnum + 1)
SET(Re4.q,sslseqnum <= 40)       # Shift by 5*8
SET(Re7,tls1_version | PS_CONTENTSIZE)
APPLY(SHA_adders)
# Build the SHA-1 HMAC and store in 20 bytes of past original end
APPLY(content_hash)
APPLY(hash_HMAC)
SET(Rr1,PS_CONTENTSIZE)
SET(CONTENT(Rr1).q,Re0.q)
SET(Rr1,Rr1 + 16)
SET(CONTENT(Rr1).w,Re4)
# Now encrypt the content plus hmac, moving all 5 bytes to “right”
SET(Rr1,Rr1 + 4)           # content size plus HMAC
APPLY(Ssl_encrypt)
# Create the SSL record header and update IP_PACKETLEN
SET(CONTENT(0).w,ssl3_rt_application_data)
SET(Rr1,Rr1 + ssl_rec_hdr_len)
SET(Rr0,Rr0 >> 16)         # Padding added by encrypt
SET(Rr1,Rr1 + Rr0)         # New total content size
SET(CONTENT(3).h,Rr1)
SET(Rr1,Rr1 - PS_CONTENTSIZE)  # delta increase in packet size
SET(IP_PACKETLEN,IP_PACKETLEN + Rr1)
```
CLASSIFY Policy

The CLASSIFY policy is a general multi-field, multi-comparator searching mechanism to look up a set of values in a database. The program can define the number of values and the lookup criteria. As an illustration, consider the following database.

<table>
<thead>
<tr>
<th>mc</th>
<th>value</th>
<th>mc</th>
<th>value</th>
<th>mc</th>
<th>value</th>
<th>mc</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
<td>mc</td>
<td>value</td>
</tr>
</tbody>
</table>

This database consists of some number of rows, each with four values and their match comparators (mc). CLASSIFY can take four values we present to it and determine which of the 10 rows, if any, match.

There are several other basic options. One is where all of the rows have the same match criteria, as conceptually shown below.

<table>
<thead>
<tr>
<th>mc</th>
<th>mc</th>
<th>mc</th>
<th>mc</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
</tbody>
</table>

Another basic option allows us to “turn off” any of the columns of values for a specific search.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSIFY</td>
<td>Searches a multi-field database</td>
<td>[NUMBER(constant) [MODE(1)] ] or [DATABASE(external name)] or [LINKED(constant)] DATA(list of values) [SELECT(value)] [ROW(value)] [ [QUERY] or [SET] or [GETCOMP] or [GET] or [GETCOMP] ]</td>
</tr>
</tbody>
</table>

The CLASSIFY policy must refer to a classification database in one of three ways:

- NUMBER is specified, which creates an empty database of the specified size
- DATABASE is specified, in which case the policy uses a database created external to the PPL program
- LINK is specified, which refers to another CLASSIFY policy that must define the database in one of the two above ways
NUMBER defines the number of rows in the database. One and only one of NUMBER and LINKED must be present.

MODE(1) defines the structure of the database. If not present, each row has match criteria. If MODE(1) is present, there is one set of match criteria that apply to all rows.

LINKED refers to the CLASSIFY policy defining the database (containing the NUMBER or DATABASE keyword). LINKED allows one to create multiple CLASSIFY policies for performing different operations or specifying different DATA variables.

DATABASE specifies the external name (external to PPL) of the database. The database is constructed by programs separate from the PPL compiler. The format of the database is very specific to the algorithms used by the CLASSIFY policy.

The following set of keywords provide information to the CLASSIFY function.

DATA defines a set of values associated with a row. For the policy in which NUMBER is specified, the values in the DATA keyword determine the number of elements in each row and their width. When doing a lookup, the values expressed are the values to be looked up. DATA is defined in more detail in a table later in the section.

SELECT an optional keyword that can be used with a search to ignore certain elements (columns). If SELECT is not present, all elements are used. If present, if bit \( n \) in its value is 0, element \( n \) is ignored and thus DATA value \( n \) is not used. For instance, SELECT(3) means ignore all but the first two elements and DATA values.

ROW specifies a row number. On a search, ROW is optional; if present, it denotes the ROW at which the search starts. How ROW is used in other operations is defined in a table shortly.

If none of the remaining keywords are present, an APPLY of the policy is a search. If there is a match in the database, the row number is placed in Rr0; otherwise Rr0 is set to FuF. In many situations this is sufficient – cases where the database is constructed by a program outside of the PPL program. The remaining keywords are defined below. At most one can appear on a CLASSIFY policy, and in almost all cases they would appear in conjunction with the LINKED keyword.

QUERY looks for an empty row, beginning at the specified row (ROW) or 0. It stores the row number of an empty row in Rr0, or stores FuF if no empty row can be found.\(^{32}\)

SET stores the values of DATA in the row specified by ROW.

SETCOMP stores encoded values of DATA (see description later) as the match comparators in the row specified by ROW. If the database is MODE(1) (one set of comparators for all rows), the ROW value is ignored.

GET obtains the values of the elements in the row specified by ROW and stores these in the values of DATA.

---

\(^{32}\) Note that in a mode-1 database, there is an internal indicator per row of whether the row is empty. For a mode-0 database, empty is denoted by the first comparator being the empty comparator.
GETCOMP obtains the criteria of the elements in the row specified by ROW and stores these as encoded values in the values of DATA. If the database is MODE(1), (one set of criteria for all rows), the ROW value is ignored.

Assuming a database exists, the following example does a search in the database.

```
Label: Policy CLASSIFY NUMBER(2500)
DATA(IP_DEST,IP_SOURCE,IP_PROT,IP_DSCP,PS_VLAN,Re7)
```

In this case we are searching using six values: four values from the current packet, the virtual network number, and the value in register Re7. Returned in Rr0 is the number of the first matching row, or FuF if no match. If we wanted to start our search at other than the beginning, we would specify the start point with the ROW keyword.

The data width of the database is defined by the number and size of values in the DATA keyword in the CLASSIFY policy containing the NUMBER keyword, or, in the case of a CLASSIFY database containing the DATABASE keyword, is self describing in the database. Implementations will likely place restrictions on the number and size of DATA values, but it is recommended that at least eight values and a total width of at least 64 bytes be supported.

The following table shows how some of the keywords and result value relate to the function selected.

<table>
<thead>
<tr>
<th>Function</th>
<th>DATA</th>
<th>SELECT</th>
<th>ROW</th>
<th>Returned in Rr0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search (none of the below)</td>
<td>Values to be looked up</td>
<td>Optional. Can be used to eliminate certain columns and data values from lookup</td>
<td>Optional. If present, specifies the first row in which the lookup should start</td>
<td>Matching row or FuF</td>
</tr>
<tr>
<td>QUERY</td>
<td>Not present</td>
<td>Not present</td>
<td>Optional. If present, specifies the first row in which the search should start</td>
<td>Empty row or FuF</td>
</tr>
<tr>
<td>SET</td>
<td>Values to be stored</td>
<td>Not present</td>
<td>Typically present; the row in which the values are stored. Row 0 if not present. For mode 1, also marks the row as not empty.</td>
<td>Not changed</td>
</tr>
<tr>
<td>SETCOMP</td>
<td>Encoded comparators to be stored</td>
<td>Not present</td>
<td>Typically present; the row in which the comparators are stored. Row 0 if not present. For mode 1, if row is not present, the single set of comparators are stored. In mode 1, if row is present, that row is marked as empty.</td>
<td>Not changed</td>
</tr>
<tr>
<td>GET</td>
<td>Values must be valid destinations.</td>
<td>Not present</td>
<td>Typically present; the row from which the values are obtained. Row 0 if not present.</td>
<td>Not changed</td>
</tr>
<tr>
<td>GETCOMP</td>
<td>Values must be valid destinations.</td>
<td>Not present</td>
<td>Typically present except in mode 1; the row from which the comparators are obtained. Row 0 if not present. For mode 1, row is ignored, and the single set of comparators are obtained.</td>
<td>Not changed</td>
</tr>
</tbody>
</table>
Match Comparators

A row of the database has the following conceptual form:

<table>
<thead>
<tr>
<th>Match comparator</th>
<th>Value</th>
<th>...</th>
<th>Match comparator</th>
<th>Value</th>
</tr>
</thead>
</table>

If MODE(1) is used, the above still holds, except that every row has the same match comparators.

A match occurs if, using the match comparator for each element, all of the corresponding DATA values match the corresponding element values in a row (except for any that are omitted using the SELECT keyword). The match criteria available are listed below.

- **EQ**  Equals
- **NE**  Does not equal
- **LT**  Less than
- **LE**  Less than or equal
- **GT**  Greater than
- **GE**  Greater than or equal
- **Em**  Equals, after masking from the left. $m$ is 001 to 127
- **Nm**  Does not equal, after masking from the left. $m$ is 001 to 127
- **Rm**  Equals, after masking from the right. $m$ is 001 to 127
- **Wm**  Does not equal, after masking from the right. $m$ is 001 to 127
- **Fm**  Equals, after masking low-order byte with $m$. $m$ is 8-bit mask.
- **Mm**  Does not equal, after masking low-order byte with $m$. $m$ is 8-bit mask.
- **X**  Always match (i.e., don’t care, ignore value)
- **ZZ**  Empty (never matches)

GETCOMP and SETCOMP use encoded comparators. They consist of the four-byte values STxx, where xx are two don’t care bytes and S and T are the symbols above. For instance, if a comparator is equality, its value as returned from GETCOMP is 0x45510000, where the first two bytes are the characters “EQ”.

When a mask is used, the mask byte (T) is a binary value. For instance, if we want to express equals when masking with 8 bits from the left, the value used would be 0x45080000. When masking other than quad (128-bit) values from left or right, $m$ is used mod 32. When masking byte and halfword values from the left, the value is treated as a word value relative to masking; for instance, to compare only the first 4 bits of a byte value, the mask value should be 28. For the F and M comparators, the mask is used directly as an 8-bit value that is ANDed to the low-order 8 bits of the DATA value before comparison with the value in the row. F and M are not defined for quad (128-bit) values.

Database Construction

A classification database can be constructed in one of two ways:

- By an external program outside of the scope of PPL
- By the PPL program
A database constructed by an external program can be modified by the PPL program, if needed. Unless constructed by an external program, a database initially has all elements marked as empty.

If an external database builder is not used, or if the PPL program needs to alter the database, the QUERY, SET, SETCOMP, GET, and GETCOMP functions can be used to manipulate the database. To create a row, one must APPLY two CLASSIFY policies, one with SET and the other with SETCOMP. With SET, the DATA keyword contains the values to be stored. SETCOMP uses the DATA keyword to specify the match comparators. The match comparators are specified as values shown in the table above, with the leftmost character being the leftmost (most significant) byte of the value.

As an example, if we have a two-element (two-column) database and we wish to create a row 100 containing the values 1024 and 1099 with the comparators being greater than and less than respectively, we would apply the following two policies:

```
mnadd: Policy CLASSIFY LINKED(main) ROW(100) DATA(1024,1099) SET
mncre: Policy CLASSIFY LINKED(main) ROW(100) DATA('GT','LT') SETCOMP
```

For simplicity we have presented all the values as constants for simplicity, but in practice they would be references to values other than constants.

If invalid comparators are expressed as DATA with SETCOMP, the bad-value exception is generated.

If one wishes to mark a row as empty, one would set the criteria ‘ZZ’ into the first element of the row.

Means exist for the PPL program to "read out" a row from the database – the GET and GETCOMP functions. GET reads the row's values and GETCOMP reads the row's match criteria in the character form used above.

**Cautions When Using SET and SETCOMP**

Because a classification database is assumed to be a relatively static thing, PPL does not guarantee correct searching behavior if the PPL program itself is changing the database using SET and SETCOMP, and other concurrent PPL events are either searching or making changes. If this is the case, the PPL program must worry about synchronization explicitly. There are a number of techniques that can be used, such as only changing the database in one serial event, switching between two duplicate databases, and using the LOCK action.

**Relationship to ASSOCIATE and PATTERNS Policies**

CLASSIFY differs from ASSOCIATE and PATTERNS in the following ways:
- Association tables provide for an exact match only and have timeout provision by entry. They are thus typically used for dynamic state information, such as tracking traffic, tracking connections such as TCP and SIP connections, and performing NAT.
- The multi-pattern aspect of the PATTERNS policy searches the payload of the current packet for any substring that matches a typically large database of signatures or patterns.
- The longest-prefix aspect of the PATTERNS policy searches a “one wide” database for a value having the longest prefix that matches a specified value.
## Connection Tracking Policy

The CONNECTIONS policy provides the means to track multidirectional flows of related packets, and as such can be used to track connections and sessions. Although the examples we use herein are primarily TCP, one can use this mechanism for tracking any type of connection or session defined by a protocol.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECTIONS</td>
<td>Defines information used to track connection state</td>
<td>NUMBER(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EVENTS[[list of constants]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[PROTOCOL(constant)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[VIRTUALNETWORK(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SOURCEALIAS(value[,value])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DESTALIAS(value[,value])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[STATE(constant,constant,constant,constant)…]</td>
</tr>
</tbody>
</table>

Typically a PPL program would have one CONNECTIONS policy, although more than one can exist.\(^{33}\) A CONNECTIONS policy creates a connections table prior to the processing of the PPL program. The keywords have the following meaning:

**NUMBER**
- Specifies the maximum number of entries in the table.\(^{34}\)

**EVENTS**
- Specifies the events associated with the connections. An automatic lookup is performed upon entry to any of these events. If an event’s EVENT statement lists more than one logical port number, the first number needs to be used here.

**PROTOCOL**
- If present, connection lookups are only made for the IP protocol specified.\(^{35}\)

**VIRTUALNETWORK**
- If present and if its value is not FuF, specifies that connections shall also be qualified with a virtual-network value. The value defines the virtual network associated with a specific connection when created by applying the policy.

**SOURCEALIAS**
- If present and if its value is not FuF, specifies an IP address to be used as an alias for the source IP address in the current packet when applying the policy. If the second value is present, it specifies an L4 port that shall also be used as an alias.

**DESTALIAS**
- If present and if its value is not FuF, specifies an IP address to be used as an alias for the destination IP address in the current packet when applying the policy. If the second value is present, it specifies an L4 port that shall also be used as an alias.

**STATE**
- Zero or more descriptions of each state and its timeout behavior. If a state isn’t described, there is no timeout from it. The parameters are STATE(state_num,timeout_time,timeout_type,new_state_num)

There are four basic operations dealing with connections:

---

\(^{33}\) It would be an error to have two or more connections policies that list the same event number.

\(^{34}\) Because the most likely implementation is a hash table, this number should be significantly higher than the maximum anticipated entries. Also, plan for the fact that usage of SOURCEALIAS or DESTALIAS doubles the number of entries used.

\(^{35}\) PROTOCOL has no functional effect; it only serves as a performance optimization. For instance, if PROTOCOL(TCP) is specified, a UDP packet never initiates a lookup or search.
1. A PPL program creates a connection (more appropriately, a description of a connection) by APPLYing a CONNECTIONS policy.
2. A PPL program destroys a connection (by setting CX_STATE to 0).
3. For any event listed in a CONNECTIONS policy, a connection (if any) matching the current packet is located automatically at the start of the event.
4. Connections may time out and change state or be destroyed implicitly.

When the CONNECTIONS policy is APPLY'ed, a connection is created in the table. At the minimum, the IP protocol, and IP and port source and destination addresses are captured. If any of virtual network and IP and port source and destination addresses are specified, these values are captured also as part of the connection entry. A unique number representing the entry is created. CX_STATE and CX_DATA values of the entry are set to 1 and the unique value respectively. Rr0 is assigned the unique value if successful or FuF if insufficient space or connection already exists.

Connections are typically bidirectional (two-directional flows), so the IP addresses and ports are used in pairs, and each pair is interchangeable (can match source or destination address). Optionally, a CONNECTIONS policy can require a virtual network value, meaning that for a packet to be considered part of a created connection, it must also be of the same virtual network value. Also, the CONNECTIONS policy provides the option of specifying, connection by connection, alternative or alias IP address/port values. To illustrate the combinations, the following table shows packets that match different examples of connections. We will assume the current packet at time of creation had source address (IP and port) of 10.10.10.10/200, destination address of 20.20.20.20/80 and protocol of TCP.

<table>
<thead>
<tr>
<th>CONNECTIONS policy values at time of creation of entry</th>
<th>Packets that match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>Network</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>4</td>
</tr>
<tr>
<td>TCP</td>
<td>4</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
<tr>
<td>TCP</td>
<td>6</td>
</tr>
</tbody>
</table>

36 For IP_PROTOCOL, the value used is the value the packet had at the time it became the current packet.
37 This has many uses, basically any circumstance where you need to track a connection but different IP addresses might appear. This can occur when load balancing and when using a variety of forms of NAT.
When an event named in the CONNECTIONS policy occurs, the connection (if any) is located automatically and its CX_STATE and CX_DATA values are taken from the entry. If CX_STATE has the value 0 upon event entry, no connection was found and CX_DATA is undefined.

One can destroy a connection entry by setting CX_STATE to 0. In addition, connection entries can change implicitly via timeouts, if specified. The optional STATE keyword allows one to describe a timeout value to be associated with each state. STATE has the following form:

```
STATE(state_num,timeout_time,timeout_type,new_state_num).
```

- **state_num** is the state being described. state_num cannot be 0.
- **timeout_time** is the approximate number of milliseconds of the timeout.
- **timeout_type** if 0 means idle timeout, if 1 means absolute timeout.
- **new_state_num** is the definition of what happens upon a timeout; the connection’s state is changed to this value.

For instance STATE(1,10000,0,0) says that if a connection is in state 1 for approximately 10 seconds with no packets matching this entry, the connection’s state should be changed to 0 (and 0 has a special meaning, specifically destroy the connection entry). An idle timeout means that the timeout timer is reset each time a packet matches the entry. An absolute timeout means that the state transition should happen in approximately the timeout time independent of the arrival of matching packets.

The following shows a bump-in-the-wire PPL program for tracking TCP connections between outside agents and inside agents, where both are allowed to initiate a connection. We will deal here with packets that are not addressed with the alias addresses; we will return to this later. We will also assume that PPL code exists to check for anomalous combinations of TCP flags and will not show this code here.

```ppl
Define NONE = "0"
Define CONNECTING = "1"
Define ESTABLISHED = "2"
Define CLOSING = "3"
Define FINALCLOSE = "4"
Define IDLE = "0"
Define ABSOLUTE = "1"
Define one_sec = "1000"
Define one_min = "60000"

Cxtable: Policy CONNECTIONS NUMBER(100000) EVENTS(incoming,outgoing)
  PROTOCOL(TCP)
  STATE(CONNECTING,one_sec,ABSOLUTE,NONE)
  STATE(ESTABLISHED,one_min,IDLE,NONE)
  STATE(CLOSING,one_sec,IDLE,NONE)
  STATE(FINALWAIT,one_sec,ABSOLUTE,NONE)

Event(incoming)
  Rule EQ(TCP_SYNONLY,1) EQ(CX_STATE,NONE) APPLY(Cxtable)
  Rule EQ(TCP_SYNONLY,1) NE(CX_STATE,NONE) NE(CX_STATE,CONNECTING) DROP
    # A SYN by itself with no connection creates one in the connecting
    # state. A SYN when we’re already in the connecting state means a
    # simultaneous connect and is just forwarded. A SYN in other than the
    # none or connecting state is wrong and is dropped.
  Rule EQ(TCP_SYNACK,1) NE(CX_STATE,CONNECTING) DROP
  Rule EQ(TCP_SYNACK,1) EQ(CX_STATE,CONNECTING) SET(CX_STATE,ESTABLISHED)
```
# ACKONLY’s in the ESTABLISHED or CLOSING states are fine and
# just fall through
Rule  EQ(TCP_ACKONLY,1) EQ(CX_STATE,FINALCLOSE)  SET(CX_STATE,NONE)
Rule  EQ(TCP_ACKONLY,1) EQ(CX_STATE,CONNECTING)  DROP
Rule  EQ(TCP_ACKONLY,1) EQ(CX_STATE,NONE)        DROP
Rule  EQ(TCP_RST,1)                              SET(CX_STATE,NONE)
Rule  EQ(TCP_FIN,1) EQ(CX_STATE,CLOSING)         SET(CX_STATE,FINALCLOSE)
Rule  EQ(TCP_FIN,1) EQ(CX_STATE,ESTABLISHED)     SET(CX_STATE,CLOSING)

...
Event(outgoing)
  # place exactly the same code here

The next case we will look at is straightforward TCP splicing. Straightforward splicing can be
done when incoming TCP connections are requested to a gateway or proxy VIP address (herein
called the VIP), and we determine at connection setup time a real inside agent to handle the
connection and then forward the remainder of the flow to that agent. This straightforward
splicing assumes we aren’t making the load balancing decision on layer-7 information; if we
want to do this, we need to use a TCP proxy, which is discussed later.

We can accomplish this load balancing and splicing by adding six rules to the incoming event of
the previous example and one rule to the outgoing event. An example of handling the incoming
event would be

Define SVIP="IPLIST(0)"
Define SIMPLE = "0"
Define SPLICE = "1"
Define PROXY  = "2"
Agents:  ARRAY(4) INITIAL(192.168.1.1,192.168.1.2,192.168.1.3,192.168.1.4)
Cxtable: Policy CONNECTIONS NUMBER(100000) EVENTS(incoming,outgoing)
            DESTALIAS(SVIP)
            STATE(CONNECTING,one_sec,ABSOLUTE,NONE)
            STATE(ESTABLISHED,one_min,IDLE,NONE)
            STATE(CLOSING,one_sec,IDLE,NONE)
            STATE(FINALWAIT,one_sec,ABSOLUTE,NONE)
Event(incoming)
Rule  EQ(IP_DEST,SVIP) EQ(TCP_SYNONLY,1) COMPUTE(RAND,Rr0) SET(Rr0,Rr0 & 3)
      SET(IP_DEST,Agents(Rr0))
Rule  EQ(IP_DEST,SVIP) NE(IP_PROT,TCP)   COMPUTE(RAND,Rr0) SET(Rr0,Rr0 & 3)
      SET(IP_DEST,Agents(Rr0))
Rule  EQ(IP_DEST,SVIP) EQ(IP_PROT,TCP)
      NE(TCP_SYNONLY,1) NE(CX_STATE,NONE)
      EQ(CX_DATA,SPLICE)                       SET(IP_DEST,CX_BIP)
Rule  EQ(IP_DEST,SVIP) EQ(IP_PROT,TCP)
      NE(TCP_SYNONLY,1) NE(CX_STATE,NONE)
      NE(CX_DATA,SPLICE)                       DROP

Insert other input section code from prior example on previous page here

Rule  EQ(IP_DEST,SVIP) EQ(TCP_SYNONLY,1)    SET(CX_DATA,SPLICE)
Rule  EQ(IP_DEST,SVIP) NE(IP_PROT,TCP)     SET(CX_DATA,SPLICE)
The first two rules have us pick an agent (which will change IP_DEST at program end). We do these actions if the destination address is SVIP and if the packet is a TCP packet with just SYN set. Also, we do the same actions for a packet addressed to VIP that is not a TCP packet.

The next (third) rule is the one that does the actual splicing. This occurs for a SVIP-addressed packet that is TCP but other than SYN only where the mode is splice. In this case we change the destination address to that which was stored in the table as the inside agent IP address. And the next rule drops the packet if it is a non-SYN TCP packet to SVIP and the had not been set to splice. The last two rules repeat the expressions of the first two rules and thus set the mode to splice if we picked an agent at the beginning.

We need to add one rule to the output section

```
Rule NE(CX_STATE,NONE) EQ(CX_DATA,SPLICE)    SET(IP_SOURCE,SVIP)
```

This rule may or may not be needed, depending on whether the agents reply to connections with their own source IP addresses or VIP, and whether we are implementing NAT in our PPL program.

Connection tracking can also be used to restrict the use of “connectionless” UDP traffic. Suppose we wish to exclude all incoming UDP traffic except for responses to internally generated network time services (NTP) requests. We will assume we wish to limit this to server-to-server NTP requests, where both use port 123. The following code segments in the input and output sections accomplish this.

```
Event(incoming)
    . . .
Rule EQ(IP_PROT,UDP) EQ(L4_SPORT,123) EQ(L4_DPORT,123)
    EQ(CX_STATE,CONNECTING)                   SET(CX_STATE,NONE) JUMP(AAA)
Rule EQ(IP_PROT,UDP)                           DROP
AAA:
    . . .
Event(outgoing)
    . . .
Rule EQ(IP_PROT,UDP) EQ(L4_SPORT,123) EQ(L4_DPORT,123)
    EQ(CX_STATE,NONE)                         APPLY(Cxtable)
    . . .
```

Reading the output section first, we create a connection entry if we see an outgoing UDP request for ports 123 and there is no connection existing. In the input section, the only UDP message we accept is one for ports 123 with the state being connecting, at which time we destroy the connection entry. The output section presumes we allow other outgoing UDP messages; if we wished to preclude all others, we could jump around a DROP action as in the input section.

The last case we will consider here is the situation where we wish to load-balance TCP connections based on the layer-7 data. This approach requires a proxy because we need to do the three-way handshake with the requester in order to see the layer-7 data, at which time we can decide whom we wish to have handle the connection and go through the three-way handshake with this agent. We assume this proxy function is handled by control-plane software outside of the scope of the PPL program, but that once both handshakes are done, we revert to
splicing for better speed. We also assume that the control-plane proxy code controls the TCP sequence and acknowledgement numbers of the second handshake such that we don’t need to translate these during the later splicing.

We will also assume in this example for simplicity that we’re handling only inbound TCP initiations to the single VIP and there are no outgoing initiations. To remove these assumptions, one needs to merge in some of the PPL code described earlier in this section.

```plaintext
Define local_network = "1"
Define outside_network = "0"
Define SVIP="IFLIST(0)"
Servers: ARRAY(16)
Cpproxystart: PROGRAM FUNCTION(controlTCPproxy.ex) DATA(1,PS_PACKETHANDLE)
Cpproxysynack: PROGRAM FUNCTION(controlTCPproxy.ex) DATA(2,PS_PACKETHANDLE)
Cpproxycontinue: PROGRAM FUNCTION(controlTCPproxy.ex) DATA(3,PS_PACKETHANDLE)
Cxtable: Policy CONNECTIONS NUMBER(100000) EVENTS(incoming,outgoing)
STATE(CONNECTING,one_sec,ABSOLUTE,NONE)
STATE(ESTABLISHED,one_min,IDLE,NONE)
STATE(CLOSING,one_sec,IDLE,NONE)
STATE(FINALWAIT,one_sec,ABSOLUTE,NONE)

Event(incoming)
Rule NE(IP_DEST,SVIP) JUMP(AAA) #Bypass if not VIP
Rule NE(IP_PROT,TCP) JUMP(AAA) #or not TCP
Rule EQ(TCP_SYNONLY,1) EQ(CX_STATE,NONE) APPLY(Cxtable)
    SET(CX_DATA,PROXY)
    APPLY(Cpproxystart) STOP
Rule EQ(TCP_SYNONLY,1) NE(CX_STATE,NONE) DROP STOP
Rule EQ(CX_DATA,SPLICE) SET(IP_DEST,CX_BIP)
    # The above prepares to splice anything once in splice mode
Rule EQ(TCP_ACKONLY,1) EQ(CX_DATA,PROXY)
    EQ(CX_STATE,CONNECTING) JUMP(FIND_A_SERVER) # code shown later
Rule EQ(TCP_ACKONLY,1) EQ(CX_DATA,PROXY)
    EQ(CX_STATE,FINALCLOSE) SET(CX_STATE,NONE) DROP STOP
Rule EQ(TCP_ACKONLY,1) EQ(CX_DATA,PROXY) APPLY(cpproxycontinue) STOP
    # The above is the case where the connector is sending data faster than
    # the control plane can switch us into splice mode
Rule EQ(TCP_ACKONLY,1) EQ(CX_DATA,SPLICE)
    EQ(CX_STATE,FINALCLOSE) DROP STOP    # Should not occur
Rule EQ(TCP_ACKONLY,1)
    EQ(CX_STATE,FINALCLOSE) SET(CX_STATE,NONE)
Rule EQ(TCP_ACKONLY,1)
    FORWARD(local_network)
    STOP    #main spliced case
Rule EQ(TCP_RST,1)
Rule EQ(TCP_RST,1) EQ(CX_DATA,SPLICE) FORWARD(local_network) STOP
Rule EQ(TCP_RST,1) EQ(CX_DATA,PROXY) DROP
Rule EQ(TCP_FIN,1) EQ(CX_STATE,ESTABLISHED) SET(CX_STATE,CLOSING)
Rule EQ(TCP_FIN,1) EQ(CX_STATE,CLOSING) SET(CX_STATE,FINALCLOSE)
Rule EQ(TCP_FIN,1) EQ(CX_DATA,PROXY) DROP
Rule EQ(TCP_FIN,1) EQ(CX_DATA,SPLICE) FORWARD(local_network) STOP
Rule EQ(TCP_FIN,1) EQ(CX_STATE,ESTABLISHED) SET(CX_STATE,CLOSING)

Event(outgoing)
Rule EQ(IP_DEST,SVIP) EQ(TCP_SYNACK,1)
```

EQ(CX_DATA, PROXY) EQ(CX_STATE, CONNECTING) SET(CX_STATE, ESTABLISHED)
SET(CX_DATA, SPLICE)
APPLY(cpproxsynack) STOP

SET(CX_STATE, NONE) DROP STOP

EQ(IP_DEST, SVIP)
# all other cases to VIP are just a RST or anomalies

EQ(CX_DATA, SPLICE)
# May not be needed

EQ(TCP_FIN, 1) EQ(CX_STATE, CLOSING) SET(CX_STATE, FINALCLOSE)
EQ(TCP_FIN, 1) EQ(CX_STATE, ESTABLISHED) SET(CX_STATE, CLOSING)
EQ(TCP_RST, 1)
# ACKONLY’s in the ESTABLISHED or CLOSING states are fine and
# just fall through
EQ(TCP_ACKONLY, 1) EQ(CX_STATE, FINALCLOSE) SET(CX_STATE, NONE)
EQ(TCP_ACKONLY, 1) EQ(CX_STATE, CONNECTING) DROP
EQ(TCP_ACKONLY, 1) EQ(CX_STATE, NONE) DROP

The only thing missing from the above is the load-balancing algorithm at the jump to
find_a_server. This should be some code that ends by changing the IP_DEST and then says
FORWARD(CPPROXYCONTINUE). Since this code is independent from the rest of the
example, it was excluded above. However, to show a specific example, we write this code
below for one specific type of load balancing on a layer-7 protocol.

In this example, we will do a hashed load balance based on the call_ID in a SIP INVITE
message.

FIND_A_SERVER:
Rule SCAN(str"Call_ID: ")
SET(Re30, CONTENT(Rr0).w) SET(Rr0, Rr0 + 4)
SET(Re30, Re30 + CONTENT(Rr0).w)
COMPUTE(HASH, Rr0, Re30)
SET(Rr0, Rr0 & 15) SET(IP_DEST, Servers(Rr0))
APPLY(cpproxycontinue) STOP

# The above takes the first 8 characters of the call ID and adds each half to
# form a 32-bit value that is used as the hash value to select a server
# If get to the below, not an INVITE or INVITE with call_ID not found
Rule APPLY(group2) APPLY(cpproxycontinue) #send to some other group

Unique Connection Value

When a connection is created, a unique value is created and stored in the connection’s
CX_DATA field and returned in Rr0. It could be used, for instance, if one wanted to store in an
array other information about a connection. Therefore, the unique value will be in the range
0..N-1, where N is the least power of 2 that is equal to or greater than the value of the NUMBER
parameter.

Real-Time Connection State

For any specific connection, the implementation is required to process the associated packets
serially (as opposed to concurrently), although the implementation is permitted to be pipelined.
If pipelined, there is a possibility, albeit generally very small, that the connection state of a
packet is being looked up while a packet just ahead of it is changing the connection state. In this case, the value of CX_STATE could be obsolete.

If this can occur in a particular implementation and application, there is a mechanism to obtain the absolutely correct value of CX_STATE; it is the CONN function of the COMPUTE action.
CONTROL Policy

The CONTROL policy allows one to control certain aspects of event processing. Typically, one might apply the CONTROL policy in the startup event.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL</td>
<td>Controls event processing</td>
<td>ID(value) EVENT(value) SIZE(value) [DATA(list of values)]</td>
</tr>
</tbody>
</table>

The ID value expresses a control function to be performed, and the interpretation of the other keywords and their values depends on the ID value.

<table>
<thead>
<tr>
<th>ID value</th>
<th>Function</th>
<th>EVENT value</th>
<th>SIZE value</th>
<th>DATA values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Enable or disable timed invocation of an event</td>
<td>Event number</td>
<td>Time interval</td>
<td>Not used</td>
</tr>
<tr>
<td>1-FuF</td>
<td>Reserved. Causes invalid policy value exception</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ID value 0 enables or disables the processing of a specified event on a periodic basis. The EVENT value denotes the number of the event; if there is no event of this number, the invalid-policy-value exception occurs. The SIZE value denotes the period, in milliseconds. A SIZE value of 0 denotes a period of approximately 0.25 milliseconds. If the SIZE value is FuF, timed invocation of the specified event is turned disabled. When enabled, the event is invoked as a packetless event with no parameters approximately every SIZE milliseconds. \(^{38}\)

---

\(^{38}\) One would typically denote the event as SERIAL. If not serial, it is possible that multiple instances of the timed event might occur.
DEFRAG Policy

The DEFRAG policy allows one to collect fragmented packets and reassemble them into a nonfragmented packet.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFRAG</td>
<td>Assembles fragments into nonfragmented packets</td>
<td>NUMBER(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIZE(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAXPACKETS(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MODE(constant[,constant])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMEOUT(value[,value]))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[VIRTUALNETWORK(value)]</td>
</tr>
</tbody>
</table>

DEFRAG takes the current packet from the PPL program; the program then has no current packet. DEFRAG stores the (fragmented) packets internally and attempts to create nonfragmented packets. The nonfragmented packet can be returned to the PPL program or forwarded to a PPL event. A nonfragmented packet is built from related fragments. Two fragments are related if they have the same:

- Source IP address
- Destination IP address
- IP protocol
- IP identification (16 bits in IPv4, 32 bits in IPv6)
- Virtual network, only if the VIRTUALNETWORK parameter is specified

The parameters of the DEFRAG policy are:

- **NUMBER** specifies the maximum number of fragment relationships the policy will track, meaning the maximum number of nonfragmented packets the policy will attempt to build concurrently.
- **SIZE** specifies the maximum size (in bytes) of a nonfragmented packet. This is a recommendation, meaning that the implementation is allowed to round it up (e.g., to a power of 2).
- **MAXPACKETS** specifies the maximum number of fragments that are allowed in a relationship (i.e., for assembly into an unfragmented packet).
- **MODE** specifies whether the policy will operate synchronously or asynchronously. If MODE(INOUT) is specified, nonfragmented packets are returned back to the PPL program; if this application of the policy completes a nonfragmented packet, the nonfragmented packet is returned from the policy. MODE(IN,53), for example, specifies that the policy should run asynchronously, and anytime a nonfragmented packet is complete, it should be forwarded to event 53.
- **TIMEOUT** specifies the maximum reassembly time as one of the following:
  - `TIMEOUT(time_value)`
  - `TIMEOUT(time_value,event_number)`

  The first value is the time limit in milliseconds. If zero is specified, the time limit is approximately 0.25 milliseconds. The optional second value is the number of an event that should be notified when the timeout occurs.
VIRTUALNETWORK if present and if its value is not FuF, specifies this fragment has a virtual-network value as part of its relationship.

As an example, assume we wish to use DEFRAG to reassemble fragments as part of an IPSec implementation.

| Reassem: Policy DEFRAG NUMBER(1000) SIZE(2048) MODE(IN,10) TIMEOUT(one_sec) |
| Rule EQ(IP_PROT,ESP) EQ(PS_FRAGMENT,1) APPLY(REASSEM) |

Here if we have an IPSec ESP packet but it is a fragment, we dispose of it by handing it to the DEFRAG policy, requesting that the policy forward unfragmented packets, as they are reassembled, to event 10. Note that this example works for both IPv4 and IPv6.

The behavior of the policy, especially regarding the handling of errors, depends on the MODE keyword.

**Synchronous MODE(INOUT)**

DEFRAG consumes the current packet. If it is a fragment that is acceptable, it returns RR0 as FuF (meaning this fragment has completed an unfragmented packet), zero (meaning this fragment is the first fragment of a new relationship), or nonzero (meaning this fragment is other than the first fragment encountered). If RR0 is FuF, the new unfragmented packet becomes the current packet.

If the fragment causes NUMBER to be exceeded, the insufficient storage exception occurs. If the fragment causes MAXPACKETS to be exceeded, the invalid-packet exception occurs. In both cases, the fragment is made available to the exception handler. If the fragment’s extent would exceed the lesser of the rounded-up-by-implementation SIZE or the extent of the unfragmented packet (only known if the “MF=0” last fragment has already been given to DEFRAG), the extent exception occurs. If the fragment’s size causes the total number of bytes of the collected fragments to exceed the lesser of SIZE and size of the unfragmented packet, the invalid-packet exception occurs.\(^{39}\)

**Asynchronous MODE(IN,event)**

DEFRAG consumes the current packet and processing of the PPL event continues. At some point when an unfragmented packet is reassembled, it is forwarded to the specified event. If at any point an error is detected, such as NUMBER or MAXPACKETS exceeded or extent or size exceeded, the related packets and incomplete unfragmented packet are implicitly dropped.

**Timeout**

Timing starts when the DEFRAG policy is given a fragment that is unrelated to any other fragment being tracked by the policy. If the nonfragmented packet is not completed within the time limit, the reassembly is abandoned. If TIMEOUT is expressed with a single value – the

\(^{39}\) This could occur instead of the extent error in the case of overlapping fragments, which almost always is an error or attack.
time limit – the related fragments and incomplete unfragmented packet are implicitly dropped. If expressed with two values, the second value is the number of an event that is invoked as a packetless event. In this situation, all but the first fragment received, and the incomplete unfragmented packet, are implicitly dropped. The handle of the first fragment received is passed as the single parameter to the packetless event.

Other Considerations

If DEFRAG is applied when there is no current packet, the no-current-packet exception occurs. If DEFRAG is applied when the current packet is not a fragment (PS_FRAGMENT is 0) or not an IP packet (PS_IP = 0) the invalid-packet exception occurs. If a TIMEOUT event is expressed but there is no such event matching the number, the invalid-policy-value exception occurs.

There are three requirements for successful reassembly:

1. DEFRAG has the first fragment (i.e., where PS_NOORFIRSTFRAGMENT = 1).
2. DEFRAG has the last fragment (where MF flag from the IPv4 header or IPv6 fragment header is 0).
3. The sum of the sizes implied by each fragment is equal to the unfragmented packet size implied by the last fragment.
Packet Management Policies

For many applications described by PPL, there will be a need for only the notion of the current packet – the packet whose arrival triggered an event leading to rule processing. Other applications will need to save packets (e.g., for traffic management) or create additional packets. The NEWPACKET and PACKET policies provide the means to do this and more.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEWPACKET</td>
<td>Creates a new packet</td>
<td>[SIZE(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HEADERS(poff_value,l4off_value,coff_value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[INITIALIZE(string)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ENCAPSULATE]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[PREPEND]</td>
</tr>
<tr>
<td>PACKET</td>
<td>Performs functions on the current packet and/or a specified packet</td>
<td>[HANDLE(value) DROP]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value) FORWARD[(value[,value])]]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value) CURRENT]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value) TYPE(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value)] COPY(toff_value,soff_value,size_value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[INSERT(method,size_value,ip_pos_value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[STRIP(method,size_value,ip_pos_value)]</td>
</tr>
</tbody>
</table>

The NEWPACKET policy creates a new packet and returns its handle in Rr0 (or creates an exception if an error occurs). At least SIZE/HEADERS or ENCAPSULATE must be specified.

| SIZE     | Specifies the number of bytes to be allocated for the packet. |
| HEADERS | Specifies the offsets of headers within the packet. The form is HEADERS(poff,l4off,coff) where poff is the offset of where the IP header will start, l4off is the offset of the byte beyond the IP header, or where a layer-4 header would start if present, and coff is the offset of the byte beyond a layer-4 header, or where the content would start if present. |
| INITIALIZE | Specifies an initialization string. If present, the packet is initialized with the string (with as many bytes of the string that will fit in the packet). Initialization starts at offset 0. |
| ENCAPSULATE | Specifies that the new packet shall contain the current IP packet. If SIZE and HEADERS are omitted, the new packet is identical to the current packet. If SIZE and HEADERS are specified, the bytes of the current packet starting at PFIELD(0) are copied to the new packet starting at offset pfield as specified in the HEADERS parameter. If both INITIALIZE and ENCAPSULATE are specified, ENCAPSULATE is performed first, then INITIALIZE. |
| PREPEND | Can be specified optionally if SIZE and HEADERS are present. If present, specifies that an implementation-defined number of bytes are to be present in front of the packet. If absent, the packet starts at the beginning of the memory allocated. |

If we wanted to create a 1040-byte raw packet without a layer-2 header to be used as a TCP/IPv4 packet, we could write

---

40 The intention is that the DeviceMap statement provide a way of stating the prepend size.
Xyz: Policy NEWPACKET SIZE(1040) HEADERS(0,20,40)

The ENCAPSULATE parameter causes a new packet to be built containing a replica of the current packet. In the first statement below, an exact copy of the current packet is created. The second statement does the same but might redefine the locations of the headers.

Xyz: Policy NEWPACKET ENCAPSULATE
Xyz: Policy NEWPACKET SIZE(IP_PACKETLEN) HEADERS(0,13len,14len) ENCAPSULATE

The PACKET policy operates on a previously created packet. The keywords are:

- **HANDLE** specifies a packet handle.
- **DROP** discards the specified packet. If the handle happens to specify that of the current packet, the action is the same as specifying the DROP action.
- **FORWARD** forwards the specified packet. The values are identical to those of the FORWARD action. If the handle specified is that of the current packet, the action is identical to using the FORWARD action.
- **CURRENT** makes the specified packet the current packet. See the explanation below concerning state changes.
- **TYPE** changes the type definition of the specified packet.
- **COPY** copies an area from the current packet into the specified packet. The three values are (1) starting offset in the target packet, (2) starting offset in the current packet, and (3) number of bytes to be copied. The offsets are relative to the PFIELD base.\(^{41}\)
- **STRIP** removes the specified number of bytes from the front or rear of the current packet.
- **INSERT** adds the specified number of bytes to the front or rear of the current packet.

All seven functions (excluding HANDLE parameter) are mutually exclusive; one and only one must be specified per PACKET policy. The invalid-packet-handle exception occurs if the packet handle is invalid\(^{42}\). The extent exception occurs in conjunction with COPY, STRIP, and INSERT if the values are improper relative to the current packet.

Note that CURRENT, STRIP, and INSERT change the nature of the current packet. Packet field names can be used, as they are relative to the PFIELD base. For CURRENT, packet-state (PS_) values represent the state of the new current packet. However, connection-state (CX_STATE) has the value 0.

As a simple example, let’s suppose we wish to examine all incoming UDP packets to see if they are SIP SUBSCRIBE messages, and whenever we find one, in addition to forwarding this packet on, we create a duplicate packet and forward it to a special IP address.

Define sip_subscribe_monitor = “26.14.30.2”
Get_duplicate: POLICY NEWPACKET ENCAPSULATE
Make_current: POLICY PACKET HANDLE(Rr0) CURRENT

\(^{41}\) Which unless one explicitly changes it, is the start of the IP header.

\(^{42}\) The current implementation can detect most instances of a value not being a valid packet handle.
Rule EQ(IP_PROT, UDP) SCAN(ul"SUBSCRIBE sip:\", 0, 0)
  APPLY(Get_duplicate)
  SET(RR1, PS_PACKETHANDLE)
  APPLY(Make_current)
  SET(IP_DEST, sip_subscribe_monitor)
  FORWARD(internal_network)
  SET(RR0, RR1)
  APPLY(Make_current)
Rule FORWARD # Forward the original packet

ENCAPSULATE with the SIZE/HEADERs parameters omitted creates an exact copy of the current packet. If we had wished to encapsulate a copy of the current packet in a new IPv4 header, we could have written

Encap_duplicate: POLICY NEWPACKET SIZE(Rr0) HEADERS(20, 40, 40) ENCAPSULATE PREPEND
Make_current: POLICY PACKET HANDLE(Rr0) CURRENT
Adjust_base: POLICY PACKET INSERT(PREP, 20, 20)
Rule SET(Rr0, IP_PACKETLEN + 20) APPLY(Encap_duplicate)
  APPLY(make_current) APPLY(Adjust_base)

which would put 20 bytes of additional space at the front of the new packet. Here we used the second PACKET policy to adjust the PFIELD base.

The following is another example of packet creation. Suppose the problem is that we wish to construct an ICMP destination unreachable message and send it back to the source of the current packet.\footnote{We might alternatively alter the current packet and use it as the ICMP packet, or possibly encapsulate the current packet, but we create a new empty packet here to show some of the packet field operations. Also, for simplicity, we assume the current packet was a standard length IP packet header (five words).}

We can initialize a new packet using the optional strings, as shown below.

X: POLICY NEWPACKET SIZE(64) HEADERS(0, 20, 40) PREPEND
INITIALIZE("\"|000000000000040002200000000000000000000010050000000000000000000000000000000000000\")

Applying this policy creates an IPv4 packet of 62 bytes. The IP header is first initialized by the specified string, which in this case sets the DF flag and the TTL field. Then in the IP header the version would be set to IPv4, the header length to 5 (words), the packet length to 62, and the protocol to TCP. The TCP header is set to source port 1, destination port 80, and the rest zeros. The content field is initialized as shown.

ICMP_unreachable: POLICY NEWPACKET SIZE(56) HEADERS(0, 20, 40) PREPEND
INITIALIZE("\"|45000002E00000002001\")
Make_current: POLICY PACKET HANDLE(Rr0) CURRENT
... Rule SET(RR0.q, PFIELD(0).q)
  SET(RE0.q, PFIELD(16).q) # Save first 8 words of current packet
  SET(RE13, IP_SOURCE)
SET(RE14,IP_DEST) APPLY(ICMP_unreachable) SET(RE14,PS_PACKETHANDLE) # Save handle of current packet APPLY(Make_current) SET(CONTENT(0).q,RR0.q) SET(CONTENT(16).q,RE0.q) # Store ICMP data SET(ICMP_TYPE,3) # Destination unreachable SET(ICMP_CODE,0) # Net unreachable SET(IP_TTL,25) SET(IP_DEST,RE13) SET(IP_SOURCE,RE14) FORWARD(0)

**INSERT and STRIP**

The form of INSERT is INSERT(method, bytes, ip_pos), where method is one of PRE2, PREP, TOP2, and END, and bytes and ip_pos are values. Insert adds space to the front or rear of the current packet and possibly repositions the bases of L2FIELD and PFIELD. Depending on the implementation and on other factors discussed later, the use of Insert could require some or all of the current packet to be moved in memory.

PRE2, PREP, and TOP2 are defined by the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>L2FIELD base</th>
<th>PFIELD base</th>
<th>Possible memory move</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRE2</td>
<td>L2FIELD – bytes</td>
<td>PFIELD – ip_pos</td>
<td>None to the whole frame</td>
</tr>
<tr>
<td>PREP</td>
<td>Unchanged</td>
<td>PFIELD – ip_pos</td>
<td>None to L2 header to packet to whole frame</td>
</tr>
<tr>
<td>TOP2</td>
<td>PFIELD – bytes</td>
<td>PFIELD – ip_pos</td>
<td>None to the whole frame</td>
</tr>
<tr>
<td>END</td>
<td>Unchanged</td>
<td>PFIELD – ip_pos</td>
<td>None to the whole frame</td>
</tr>
</tbody>
</table>

Packet with L2 header

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
```

PRE2

PREP

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
```

TOP2

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
```

END

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
```

Packet with no header

```
+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+----------------+
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
|                  |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |                |
```

PRE2

```
The hatched area represents the bytes we are inserting. The light gray area is the L2 header, and the darker area represents the packet. The adjustment to L2FIELD base is such that the base is the leftmost entity; L2FIELD points to the original L2 header or to the space inserted, depending on which method is used. It is important to note that this is the case even if the implementation needed to move part or all of the packet and/or L2 header.

Insert also allows one to perform arithmetic on the PFIELD base. For instance, INSERT(PREP,20,0) would insert a 20-byte header in front of the IP header and not change the PFIELD base, where INSERT(PREP,20,20) would insert a 20-byte header and subtract 20 from the PFIELD base, making the PFIELD base be the inserted header. In the latter case, we may be doing this to insert an IP header in front of the existing header.

The implementation is expected to be intelligent about avoiding moving the packet to do an insert, for instance by studying the values to determine the minimal movement, and by providing flexibility via the DeviceMap statement to provide for reserved space in packet buffers. Also, implementations will have restrictions on the maximum amount of space that can be inserted.

Moving the PFIELD base an amount greater than the inserted space yields an extent-error exception.

The form of STRIP is STRIP(method,bytes,ip_pos), where method is one of PRE2, PREP and END, and bytes and ip_pos are values. Strip removes space from the front or rear of the current packet and possibly repositions the bases of L2FIELD and PFIELD.

PRE2 and PREP are defined by the table below.\(^{44}\)

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>L2FIELD base</th>
<th>PFIELD base</th>
<th>Possible memory move</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRE2</td>
<td>Remove space in front of the L2 header if present or before the packet otherwise</td>
<td>L2FIELD + bytes</td>
<td>PFIELD + ip_pos</td>
<td>None</td>
</tr>
<tr>
<td>PREP</td>
<td>Remove space in front of the packet, preserving the L2 header if present</td>
<td>Unchanged</td>
<td>PFIELD + ip_pos</td>
<td>None to L2 header</td>
</tr>
<tr>
<td>END</td>
<td>Remove space from the rear of the packet (i.e., reducing the maximum PFIELD offset)</td>
<td>Unchanged</td>
<td>PFIELD + ip_pos</td>
<td>None</td>
</tr>
</tbody>
</table>

PRE2 simply does arithmetic on the L2FIELD and PFIELD bases by adding values to both. For instance, STRIP(PRE2,20,0) moves the L2FIELD base 20 bytes forward. PREP removes space in front of the current packet header and possibly repositions the packet base. One could

\(^{44}\) TOP2 is not present in STRIP because it would have the same definition as PRE2.
remove a standard IPv4 header by using STRIP(PREP,0,20), which removes 0 bytes but repositions the PFIELD base 20 bytes forward. END shrinks the packet from the rear.

If the specified number of bytes do not exist in front of the L2 header or packet, or if one attempts to reposition L2FIELD or PFIELD beyond the end of the packet, or if one attempts to move the end of the packet in front of its start, an extent-error exception occurs.

**TYPE Encodings**

The values of the outer header type, which are present in the PS_TYPE value and the TYPE parameter of the PACKET policy, are listed in the table below. These values are not represented symbolically in the language because their presence and encodings are allowed to vary by implementation. The minimum set expected (but not required) in an implementation is given by the table below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Unknown</td>
</tr>
<tr>
<td>0x01</td>
<td>Ethernet</td>
</tr>
<tr>
<td>0x02</td>
<td>PPP</td>
</tr>
<tr>
<td>0x06</td>
<td>PPP over Ethernet, discovery stage</td>
</tr>
<tr>
<td>0x07</td>
<td>PPP over Ethernet, session stage</td>
</tr>
<tr>
<td>0x08</td>
<td>MPLS unicast</td>
</tr>
<tr>
<td>0x09</td>
<td>MPLS multicast</td>
</tr>
<tr>
<td>0x21</td>
<td>IPv4</td>
</tr>
<tr>
<td>0x57</td>
<td>IPv6</td>
</tr>
<tr>
<td>0xC0</td>
<td>IP over custom, custom header length 2</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>0xDF</td>
<td>IP over custom, custom header length 64</td>
</tr>
<tr>
<td>0xFE</td>
<td>Data</td>
</tr>
</tbody>
</table>

The NEWPACKET policy sets the type of the packet in the following way:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Type assumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE and HEADERS not specified</td>
<td>Type of current packet</td>
</tr>
<tr>
<td>First HEADERS value (pfield) not zero</td>
<td>Ethernet</td>
</tr>
<tr>
<td>First HEADERS value zero, second value 20</td>
<td>IPv4</td>
</tr>
<tr>
<td>First HEADERS value zero, second value &gt; 20</td>
<td>IPv6</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Data</td>
</tr>
</tbody>
</table>

One can always revise the type of a new packet by using the PACKET policy.
PATTERNS Policy

The PATTERNS policy represents a set of algorithms that perform heavy-duty multiple-pattern searches. PATTERNS does one of two types of searches:

- A search for all, or a designated section of, the current packet contents against a potentially large database of patterns. The database and the search can be tagged or not. Tagged means an additional qualifier beyond the patterns themselves.\(^{45}\)
- A search for a value (typically an IPv4 or IPv6 address) against a database, looking for the longest prefix match.\(^{46}\)

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATTERNS</td>
<td>Does a directed search through a patterns database</td>
<td>DATABASE(external name) [SCAN(base_cnst[,destination[,svalue[,dvalue]]]) [MODE(mode_val,tag_val] or [LPM(value[,qual_val])]] [LINKED(constant) ID(value)]</td>
</tr>
</tbody>
</table>

DATABASE Specifies the external name (external to PPL) of the set of patterns to be matched against. The pattern databases are constructed by programs separate from the PPL compiler. The format of the databases is very specific to the algorithms used by the PATTERNS policy.

SCAN Specifies a multipattern search. SCAN can express 1-4 values in the form SCAN(base,destination,svalue,dvalue). Base must be PFIELD or CONTENT, and thus SCAN(CONTENT) scans relative to the base of CONTENT (typically the payload) and SCAN(PFIELD) scans relative to the base of PFIELD (typically the start of the IP packet). When the policy completes, Rr0 contains the identifier of the matching pattern, or FuF if none was found.

The other three values are optional, and if used are similar to the controls and results of the PPL SCANB action. Destination is where the offset of the matching pattern in the packet relative to the base is stored. Svalue denotes an offset relative to the base where the search starts. Dvalue denotes denoting an offset relative to the base beyond which the start of the matching pattern will not be searched for. If svalue is omitted, it defaults to 0. If dvalue is omitted, it defaults to the end of the packet.

MODE can optionally be used with SCAN to performed tagged searches. MODE has the form MODE(modevalue,tagvalue) where tagvalue is a 32-bit value and modevalue describes how the tag is used. These are discussed in a subsequent section.

LPM Specifies a longest-prefix match. The first value is compared to the database. The second value, if present, is an optional qualifier.\(^{47}\)

LINKED Specifies the number of a NEWSUPERPACKET policy in the situation where one wants to search a superpacket.

ID Specifies the ID of the superpacket.

---

\(^{45}\) Although the language does not dictate the algorithm, the first implementation uses a further-optimized form of the Wu-Manber algorithm.

\(^{46}\) And the first implementation of this uses a further-optimized form of the Eatherton-Dittia Treee Bitmap algorithm.

\(^{47}\) For instance, if one said LPM(IP_DEST,PS_VLAN) one could qualify the lookup as being in one of many tables, one per virtual network.
When the PATTERNS policy is applied, a result is returned in register Rr0. If a match was found, Rr0 contains the number of the matching pattern from the database. If no match occurred, Rr0 has the value FuF. If multiple matches are valid, which one is indicated is implementation dependent. For SCAN, if a pattern was found, its offset is also stored in the destination (if expressed) in SCAN.

The following example would examine a packet's entire content

```
spam_signatures: Policy PATTERNS DATABASE(i$sigs002) SCAN(CONTENT)
```

The next example would examines a packet's content for a pattern starting at the offset in Re4 or beyond, but not beyond the offset in Re5. In addition to the pattern number being returned in Rr0, the offset of the matching pattern is returned in Rr1.

```
Inside_urls: Policy PATTERNS DATABASE(i$inside_urls) SCAN(CONTENT,Rr1,Re4,Re5)
```

The example below specifies that the pattern matching is to be done with the destination IP address of the current packet. In this situation, we are using the PATTERNS policy to determine if the IP address is a member of a set we want to handle, as opposed for instance to writing a large number of PPL rules to compare the IP address.

```
Service_clients: Policy PATTERNS DATABASE(i$clientips) LPM(IP_DEST)
```

Another usage of the PATTERNS policy with IP addresses is to find a longest-prefix match.

Note that LPM could be used to translate any 32- or 128-bit value to a 32-bit value. If one wishes to translate to a 128-bit value, this could be done in two steps: use the PATTERNS policy to translate into a 32-bit value, and then use this as an index to an array of 128-bit values.

**Tagged Searches**

We shall illustrate this first with an example

```
intrusions: Policy PATTERNS DATABASE(arachnids) SCAN(CONTENT)  
           MODE(1,Rr1) 
           . . .  
           Rule SET(Rr1,IP_PROT << 16) SET(Rr1,Rr1 | L4_DPORT) APPLY(intrusions)
```

In this program we are creating in Rr1 a tag consisting of IP_PROT and the destination port number. We express this in the PATTERNS policy in the MODE keyword, with mode 1 meaning “find any matching pattern provided its tag is this value.”

The modes currently provided are defined below.

<table>
<thead>
<tr>
<th>mode</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None. Same as omitting MODE keyword.</td>
</tr>
<tr>
<td>1</td>
<td>Consider only patterns whose tag, with an optional mask, matches the tag</td>
</tr>
</tbody>
</table>
Tagged searches are very useful when a pattern only has significance in the presence or absence of something else. Note that if the 32-bit tag is insufficient, one could hash a larger number of bits into a 32-bit tag.

**Scan Database Format**

The format of the database(s), the programs that build the databases, and the format of the inputs to these programs is outside the definition of PPL. However the database is expected to contain the following:

- Some representation of the patterns
- A designation by pattern of whether to match with case sensitivity or insensitivity
- An ID per pattern (an arbitrary value returned upon a match)
- A notation for wild cards (bytes of the pattern that match anything)
- A 32-bit tag
- A mask for the tag comparison, at least at byte granularity

**Superpackets**

The LINKED and ID parameters are only valid if SCAN is also specified. In this case, the search is done of a superpacket, and done relative to the piece_offset of the first packet in the superpacket.

---

48 The current implementation supports patterns of lengths 2-4095.
PROGRAM Policy

Some PPL programs will need the ability to communicate directly with other functions (typically software, but not limited to software) outside of the scope of the PPL program. This capability is provided by the PROGRAM policy. Examples would be signaling alerts to management software, invoking control-plane software functions, and interfacing to special customer-written proprietary data-plane code.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM</td>
<td>Signals or invokes a function outside of the scope of PPL</td>
<td>FUNCTION(external name) [DATA(list of values)]</td>
</tr>
</tbody>
</table>

The PROGRAM policy names an external function. The mechanism for resolving external names is outside of the scope of the language definition. It also provides the ability to pass a set of values to the function being invoked. It is anticipated that the implementation will consider means to bind the external name to:

- Programs on control-plane and/or management-plane processors
- Programs outside of PPL but in the data plane
- This or a different PPL program via the packetless event

The specific semantics of APPLYing a PROGRAM policy depend on the function invoked. The preferred model is that the parameters provided be copied, the function signaled, and then immediately the processing of further actions in the PPL program continues (i.e., asynchronous). A function invoked can communicate back to the PPL program by invoking a specific event.
**Queue Management Policies**

The QUEUES and QUEUE policies maintain a set of packet queues having like attributes.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUEUES</td>
<td>Defines a set of queues</td>
<td>NUMBER(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[WEIGHT]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[EVENT(constant)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[MODE(mode_cnst,lpn_val,time_val[,tspread_val])]</td>
</tr>
<tr>
<td>QUEUE</td>
<td>Performs an operation on a queue</td>
<td>LINKED(constant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUMBER(value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ADD(constant)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DATA(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[REMOVE(constant)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[HANDLE(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DATA(value)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[QUERY(value)]</td>
</tr>
</tbody>
</table>

THE QUEUES policy defines a set of queues and their common properties. If the EVENT and MODE keywords are not present, the queues are described as static queues, meaning all operations on them are explicit. For instance, the following defines 16 static queues.

**Label: Policy QUEUES NUMBER(16)**

The presence of the QUEUES policy creates the queues. APPLYING it subsequently changes the mode if specified.

NUMBER specifies the number of queues. Each queue is then known as number 0..NUMBER-1.

WEIGHT if present, denotes that the queues have dynamic weights. Weighted queues track the sum of the weights of the packets on the queue.

EVENT optionally specifies a PPL event to be invoked as a packetless event with one parameter whenever a queue goes from empty to nonempty. The parameter passed is the queue number.

MODE optionally specifies an active implicit management of the queue. MODE is described in a subsequent section.

The QUEUE policy performs an enqueue, dequeue, or query on a specified queue or a queue determined by a selection process. One and only one of ADD, REMOVE, and QUERY must appear on a QUEUE policy. The keywords are

LINKED specifies the QUEUES policy defining the queues.

NUMBER specifies either the specific queue to be operated upon, or the starting queue for the selection process.

ADD enqueues a packet on the selected queue. The constant determines how the queue is selected.

0: Use the specific queue from NUMBER
1: Use the first empty queue found, starting with NUMBER queue. If none are empty, use the NUMBER queue.
2: Use the shortest or lightest (least-weighted) queue found. If all the same size, use the NUMBER queue. If the queues are not weighted, the shortest queue is the one with fewest entries.

If HANDLE is specified with ADD, it specifies the handle of the packet to be enqueued. If HANDLE is not specified, the current packet is enqueued and the event ceases to have a current packet. If DATA is specified with ADD, it is carried along with the packet as an attribute while the packet is on the queue. If the queue is weighted, the data value is the packet weight. If the queue is not weighted, the data value could be used, for instance, as a time stamp or a code to the dequeuer.

REMOVE dequeues a packet from the selected queue. The constant determines how the queue is selected:

0: Use the specific queue from NUMBER
1: Use the first nonempty queue found, starting with NUMBER queue and proceeding “downward.”
2: Use the longest or heaviest queue found. If all the same size, use the NUMBER queue.

If HANDLE is specified with REMOVE, it denotes where the handle of the dequeued packet should be placed. If HANDLE is not specified, the dequeued packet becomes the current packet. If DATA is specified with REMOVE, it denotes where the DATA value carried with the packet should be placed. Rr0 is set to the queue number if a packet is dequeued and to FuF if no packet could be found.

QUERY returns information about the queue. Rr0 is set to the number of packets currently enqueued on the specified queue. The value specified with QUERY is set to the current weight of the queue if the queue is weighted; otherwise it is unchanged.

HANDLE Optionally specifies the handle of the packet to be enqueued (ADD) or where the handle of the dequeued packet is placed (REMOVE).

DATA Optionally specifies an attribute or weight of the packet being enqueued or where the attribute or weight is placed of the packet being dequeued.

Queue Weights

If a set of queues is weighted, each packet on a queue is considered to have a weight, which is the value expressed in the DATA keyword when the packet was enqueued. The queue weight is the sum of the packet weights. The interpretation of “weight” is up to the PPL program. The only built-in dependencies are an option of enqueuing upon the lightest queue and dequeuing from the heaviest queue.

Queue Modes

As an option on the QUEUES policy, one can define active management of the queues. The active-management options pertain to implicit de dequeuing. They are specified by the MODE keyword, whose form is

MODE(mode,logical_port_number,time[,time_spread])

The modes currently provided are defined below.
<table>
<thead>
<tr>
<th>mode</th>
<th>Process</th>
<th>Time interval given by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None. Active management is disabled.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Every specified time interval, dequeue from the next queue (rotating through the queues) and forward the packet to the specified logical port number.</td>
<td>Time</td>
</tr>
<tr>
<td>2</td>
<td>Every specified time interval, dequeue from the longest or heaviest queue and forward to the specified logical port number.</td>
<td>Time</td>
</tr>
<tr>
<td>3</td>
<td>Every specified time interval, dequeue from the next queue and forward to the lpn and id given by the DATA value of the packet (see below).</td>
<td>Time</td>
</tr>
<tr>
<td>4</td>
<td>Every specified time interval, dequeue from the longest queue and forward to the lpn and id given by the DATA value of the packet.</td>
<td>Time</td>
</tr>
<tr>
<td>5</td>
<td>Every time interval by queue, dequeue and forward the packet to the specified logical port number.</td>
<td>Time + queue_number * time_spread</td>
</tr>
<tr>
<td>6</td>
<td>Every time interval by queue, dequeue and forward the packet to the lpn and id given by the DATA value of the packet.</td>
<td>Time + queue_number * time_spread</td>
</tr>
</tbody>
</table>

The time_value and time_spread, if used, are in units of approximate microseconds.

In modes 3, 4, and 6, the logical port number in the MODE keyword is not used, and instead the forwarding information comes from the packet. The upper 16 bits of the DATA value of the packet are used as the logical port number, and the lower 16 bits are used as the id (see description of the FORWARD action of PPL).
Rate Measurement Policy

The RATE policy creates a time-based rate counter. Each APPLY of the policy causes a count to be accumulated and returns in register RR0 the approximate count per timebase.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>RATE</td>
<td>Maintains a time-based rate</td>
<td>TIMEBASE(value) COUNTING(value) RESETTIME(value) [COUNTED(value) STARTED(value)]</td>
</tr>
</tbody>
</table>

- **TIMEBASE** Specifies the quantum of time (in milliseconds) used for the calculation.
- **COUNTING** Specifies the number of units to be added to the measurement.
- **RESETTIME** Specifies the approximate number of milliseconds between times when the counter is reset. If omitted or less than timebase, the counter is not reset.
- **COUNTED** Optionally specifies a place where count is maintained.
- **STARTED** Optionally specifies the time that this rate was reset.

The following is an example of using a rate counter to inhibit more than 1000 TCP connection attempts per 30 seconds over a time period of approximately one day. To count occurrences, parameter COUNTING is set to one.

```plaintext
Define day = "86400000"
Define sec30 = "30000"
TCPconnection_rate: Policy RATE RESETTIME(day) TIMEBASE(sec30) COUNTING(1)
. . .
Rule EQ(TCP_SYNONLY,1) APPLY(TCPconnectionrate) GE(Rr0,1000) LOG DROP
```

COUNTED and STARTED are optional and intended for use when maintaining a set of rates. Their use is illustrated later. If not present as explicit parameters, they exist conceptually as part of the policy.

Using the parameter names as their values and Tod to denote the current time, RATE performs the following function:

\[
\text{Counted} = \text{Counted} + \text{Counting} \\
\text{If } (\text{Tod} - \text{Started}) < 1000 \\
\text{Then } \text{Rr0} = \text{Counted} \times \text{Timebase} / 1000 \\
\text{Else } \text{Rr0} = \text{Counted} \times \text{Timebase} / (\text{Tod} - \text{Started}) \\
\text{If Resettime > Timebase And Resettime < (Tod - Started) } \\
\text{Then Started = Tod, Counted = 0}
\]

RATE is well behaved right after first use or reset. Until approximately one second has transpired, it always returns the rate calculated as if one second has transpired. If at any time the returned rate cannot be expressed as a 32-bit value, RR0 is returned as FuF.

The example below is a fragment of a PPL program tracking traffic bit rates for a set of traffic classes. Note that it shows several interesting things: it is indexing into a set of policies, it is counting not by one but by the layer-3 packet bit size, and it is using pairs of RATE policies, one for a sustained rate (over 30 minutes) and the other for a peak rate (over 30 seconds).
Rules:

```
Rule ACT NE(Tclass,FuF) SET(Rr0,IP_PACKETLEN + 18) SET(Rr0,Rr0 << 8)
  SET(Tclass,Tclass + Rates) APPLY(Tclass)
  SET(susrate,Rr0) SET(Tclass,Tclass + 1) APPLY(TClass)
Rule ACT LE(susrate,suslim) LE(Rr0,peaklim) FORWARD STOP
```

Unless COUNTED and STARTED are specified, each RATE policy maintains one internal “Counted” state value having the effect of a 128-bit value and one internal “Started” state 32-bit value. By specifying COUNTED and STARTED explicitly, one can hold these outside of the RATE policy, which is useful when there is a need to maintain a large set of rates having the same characteristics. For instance, if we wanted to maintain a set of rates for each element in an association or connections table, we might code:

```
Define hosts=1000
Tracker: Policy ASSOCIATE NUMBER(hosts) SEARCHKEYS(IP_SOURCE)
Trates: Policy RATE TIMEBASE(hour) RESETTIME(day) COUNTING(Re2)
  COUNTED(Tcounts(Rr0))STARTED(Tstimes(Rr0))
Tcounts: Array(hosts).q
Tstimes: Array(hosts)
```

When we APPLY the RATE policy, it will use an element of array Tcounts as the Counted value and an element of array Tstimes as the Started value, which gives us the ability to maintain a rate associated all 1000 elements of the association table.

All parameters are interpreted as 32-bit values except the optional COUNTED parameter, which can be 32 or 128 bits.

Like all other policies, RATE behaves correctly in a concurrent implementation (e.g., parallel events can use the same RATE policy).

RATE will not behave correctly if more than $2^{32}$ one-millisecond intervals occur between applying the policy (if COUNTED/STARTED are not specified) or between applying the policy to a specific COUNTED/STARTED location. This is approximately 50 days.

The implementation is not required to perform an exact division; a value that is guaranteed to be within ±3% of the mathematically correct quotient is acceptable.
**Superpacket Policies**

A superpacket is a sequence of packets that can be dynamically constructed by a PPL program to behave as a single packet for a limited set of operations. The NEWSUPERPACKET policy defines a database used to hold superpackets and creates a superpacket when applied. The SUPERPACKET policy performs operations on a superpacket.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Purpose</th>
<th>Keywords</th>
</tr>
</thead>
</table>
| NEWSUPERPACKET | Defines a database of superpackets and creates a superpacket | NUMBER(constant)  
|                |                                      | MAXPACKETS(constant)  
|                |                                      | ID(value)  
|                |                                      | [TIMEOUT(value,[value])]  |
| SUPERPACKET    | Performs an operation on a superpacket | LINKED(constant)  
|                |                                      | ID(value)  
|                |                                      | [ADD(value,value,value,value)]  
|                |                                      | [DELETE([value])]  
|                |                                      | [QUERY(value,value)]  
|                |                                      | [SCANB(string,[svalue,[dvalue]]])  
|                |                                      | [SCANE(string,[svalue,[dvalue]]])  
|                |                                      | [FORWARD([value,[value]])]  
|                |                                      | [DROP]  
|                |                                      | [SETFIELD([value,(value)[w],[value][w],value])]  
|                |                                      | [WHICHPACKET(dest,offsetvalue)]  |

The parameters of the NEWSUPERPACKET policy are

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>Specifies the maximum number of superpackets that can exist at any point in time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPACKETS</td>
<td>Specifies the maximum number of packets that can exist in any superpacket.</td>
</tr>
<tr>
<td>ID</td>
<td>Specifies the unique identifier of the superpacket being created. Valid values are 0..NUMBER-1.</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>Specifies what should be done if the specified time elapses after the superpacket is created and the superpacket still exists. The time is in units of milliseconds, and the value 0 denotes approximately 0.25 milliseconds. TIMEOUT(time_limit) results in the deletion of the superpacket and dropping of any packets within it.</td>
</tr>
<tr>
<td></td>
<td>TIMEOUT(time_limit,event_number) results in an event being invoked. E.g., TIMEOUT(10000,42) denotes that event 42 should run if 10 seconds have expired.</td>
</tr>
</tbody>
</table>

A superpacket consists of one or more packets that logically have the appearance to the right. Packets are concatenated at a certain offset, which may be specified dynamically. A superpacket is constructed piecemeal by inserting a packet into a superpacket. Packets can be inserted in any order; the order is specified by presenting a relative first-byte-number when a packet is inserted. One can also have the “first” packet automatically removed from a
superpacket when a new one is added, allowing for instance the superpacket to represent a moving window in a packet flow.

Each packet in a superpacket has three additional attributes:

- **Piece_offset**: the offset of the first byte to be considered logically in the superpacket. The offset is from the first byte in the packet (PFIELD base).
- **Piece_length**: the number of bytes in the packet to be considered logically in the superpacket.
- **First_byte_number**: the logical (and arbitrary) number of the first byte at piece_offset. When a packet is added to a superpacket, it is inserted based on its first_byte_number (modulo $2^{32}$). That is, the order of the packets is by first_byte_number.

We say there is no “hole between two packets” in a superpacket when the first_byte_number of packet N+1 is equal to first_byte_number + piece_length of packet N.

Applying the NEWSUPERPACKET policy creates an empty superpacket with the specified ID. If a superpacket already exists with that ID, it is deleted (and any packets within it are dropped).

The SUPERPACKET policy has the following parameters. One and only one of the optional parameters must be included per policy.

- **LINKED**: Specifies the number of the NEWSUPERPACKET policy to which this applies.
- **ID**: Specifies the ID of the superpacket on which we will operate.
- **ADD**: Inserts a packet into the superpacket. The form is ADD(packet_handle, piece_length, piece_offset, first_byte_number). Rr0 is returned as FuF if the packet cannot logically exist in the superpacket because its first byte number and length conflicts with (overlaps) other packets, as 0 if the operation was successful and no packet was pushed out of the superpacket.
  
  Note that ADD can push a packet out of a superpacket if the superpacket contains MAXPACKETS packets and the added packet has a first byte number greater than some other packet. In this case, the handle of the pushed-out packet is returned in Rr0.

- **DELETE**: Deletes a packet from the superpacket. The value specifies a packet handle. The packet is not dropped (i.e., continues to exist).

- **QUERY**: Returns information about the superpacket. Rr0 is set to 0 if the superpacket is incomplete (has holes), or to the length of the complete superpacket otherwise. The length of a superpacket is the packet length of the first packet plus the piece_length’s of all subsequent packets. The two values expressed must be valid destinations. The first value is set to the first byte number of the first packet in the superpacket and the second value is set to the number of packets in the superpacket.

- **SCANB/SCANE**: Scan the superpacket for a matching string or regular expression. The values are the same as those for the SCANB/SCANE actions. The superpacket is scanned as if it were a contiguous packet. Note that the scan starts at the piece_offset of the first packet, which is not necessarily the start of the content as defined for the SCANB/E actions.
FORWARD  Forwards any and all packets in the superpacket. The values are the same as for the FORWARD action. The packets are also deleted from the superpacket (i.e., the superpacket continues to exist, but is empty).

DROP   Drops any and all packets in the specified superpacket. The packets are also deleted from the superpacket.

SETFIELD  Moves a value between a field in the superpacket and another value. E.g., SETFIELD(Re2,(Re1).b) moves the byte at the offset in Re1 to Re2. The offset is relative to the piece_offset of the first packet in the superpacket. Note that in the form where the first value is not indexed, it must be a valid destination.

WHICHPACKET  Returns the packet handle of the packet in which the specified offset falls in the superpacket. The offset is relative to the piece_offset of the first packet. The packet handle is stored in the first value of the keyword and the offset is the second value. Rr0 contains the equivalent offset value within the packet whose handle is returned.

Note that operations that expect a complete SUPERPACKET generate the incomplete-superpacket exception if incomplete. These are SCANB, SCANE, SETFIELD, and WHICHPACKET.

Timeout

Timing starts when a superpacket is created. If it still exists after the time limit has been exceeded, one of two things happens depending on whether an event is specified in the TIMEOUT keyword.

1. If TIMEOUT is expressed with a single value – the time limit – the superpacket is deleted and any packets within it are dropped.

2. If TIMEOUT is expressed with two values, the second value is the number of an event that is invoked as a packetless event. Nothing is implicitly deleted or dropped. Two parameters are passed to the event: Re0 will contain the policy number of the NEWSUPERPACKETS policy and Re1 will contain the ID of the superpacket. If there is no event of this number, the invalid-policy-value exception occurs instead.
Exception Handling

Exceptions are “should not occur” situations and are typically associated with errors in the PPL program. For all known detectable exceptions, the following occurs:

1. Rule processing for the current event ends.
2. If an event 998 exception handler exists in the PPL program, certain information about the exception is retained and the event is signaled.
3. If no exception handler exists, the current packet (if there is one) is dropped.

One can use exception handling to take some explicit action after the occurrence of an exception. Event 998, if specified, is the point in the PPL program that invoked after an exception occurs. When the rules associated with event 998 are processed, there is no current packet and no packet state. Information about the exception is stored in the following registers just prior to the processing of the rules.

- **Re0**: number defining the nature of the exception, as shown in the table below.
- **Re1**: the handle of the current packet at the time of the exception. If there was no current packet, it has the value FuF.
- **Re2**: number of the rule in the PPL program causing the exception.

<table>
<thead>
<tr>
<th>Exception number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invalid operation (e.g., see description of COMPUTE action, jumping to a non-existing rule, applying a non-existing policy, storing into a CX_ value when there is no current connection)</td>
</tr>
<tr>
<td>1</td>
<td>Extent error – attempting to access a non-existing indexed value location (e.g., accessing beyond the bounds of a packet, beyond the extent of an array, to a non-existing explicit row in an association)</td>
</tr>
<tr>
<td>2</td>
<td>No current packet - attempting to reference or use the current packet (e.g., reference a packet field, DROP, FORWARD) when there is no current packet</td>
</tr>
<tr>
<td>3</td>
<td>Invalid policy value (specifically values in a policy that cannot be determined as invalid beforehand)</td>
</tr>
<tr>
<td>4</td>
<td>Insufficient storage</td>
</tr>
<tr>
<td>5</td>
<td>Invalid packet (e.g., not IPv4 or IPv6, invalid header length) expressed in a policy application or a forward action.</td>
</tr>
<tr>
<td>6</td>
<td>Unresolvable name - cannot resolve external (e.g., program) name</td>
</tr>
<tr>
<td>7</td>
<td>Unresolvable logical port number specified in a FORWARD action</td>
</tr>
<tr>
<td>8</td>
<td>Unknown route. Specifically, the underlying system does not understand how to transmit a packet with its destination IP address out a specific network port.</td>
</tr>
<tr>
<td>9</td>
<td>Exception event exception - exception of some type occurred in an event 998 exception handler</td>
</tr>
<tr>
<td>10</td>
<td>Time exceeded – a lock could not be obtained by the LOCK action</td>
</tr>
<tr>
<td>11</td>
<td>Invalid packet handle.(^\text{49})</td>
</tr>
<tr>
<td>12</td>
<td>Incomplete superpacket – the operation cannot be performed because it is one requiring the superpacket to be devoid of holes.</td>
</tr>
</tbody>
</table>

\(^{49}\)Implementations are expected, at the least, to detect common invalid handles like all zero and FuF. See implementation restrictions section.
What one does in an exception handler depends to some degree on whether one is debugging a PPL program or handling exceptions in a “final” program. General advice is that the exception handler should log or otherwise notify a control-plane program and drop the packet expressed in Re1. Note that the exception event is the same as a packetless event and thus one can invoke it explicitly if wanted.
Checksum Recalculation

For IPv4 packets\textsuperscript{50}, the underlying machine will compute the IP header checksum when the packet is forwarded. Layer 4 checksums (e.g., such as in TCP) need to be changed explicitly.

\textsuperscript{50} IPv6 has no checksum.
Compilation and Dynamic Compilation

The intended implementation of PPL is that it be “compiled” to a low-level set of tables that can be processed very efficiently by one or more concurrent microcode interpreters. The compiler is expected to produce the level of error analysis (syntactic and otherwise) expected by typical compliers.

The implementation is expected to support dynamic compilation, by which we mean the modification of the PPL program during execution. The suggested means of implementation is a dynamic compilation program that given a rule or policy number (or label), will replace that statement with a specified PPL statement.

For instance, if the dynamic compiler is presented with:

```
Check44: Rule EQ(IP_SOURCE,172.19.19.55) DROP
```

It needs to substitute this rule for the existing rule “Check44” in the system.\textsuperscript{51}

\textsuperscript{51} A replace mechanism is simple and sufficient. Its one weakness is that it seemingly does not allow one to insert or append multiple rules. This is easily avoided by “padding” one’s program with a sufficient set of blank rules and blank policies.
Regular Expressions

The PPL SCAN expression can take as its search argument a regular expression of the form expressed in the PERL language, UNIX GNU, IEEE POSIX 1003.2 specification and elsewhere. Because some features of regular expressions are not useful in examining layer 7 content or prohibit high-speed implementations, PPL specifies a large subset of regular expressions but not every feature present in every definition of regular expressions. One specific difference is that PPL does not provide for "greedy" regular expressions. In greedy regular expressions one tries to locate the maximum amount of matching data, and in non-greedy regular expressions, one tries to locate the minimal amount of matching data.

Each character, metacharacter, and sequence of metacharacters in the PPL use of regular expressions are defined in the table below.

<table>
<thead>
<tr>
<th>Group</th>
<th>Symbol(s)</th>
<th>Usage</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A...Z a...z 0...9 _ etc</td>
<td>Matches the exact character. Can be letters, numerals, space, underscore, etc., anything not a metacharacter. Metacharacters are . [] ? * + {} $ ( )</td>
<td></td>
</tr>
<tr>
<td>matching</td>
<td>\p</td>
<td>Matches p if p is a metacharacter. If p is other than a metacharacter, it represents one of the later cases in the table, or an error.</td>
<td>$10.00 matches $10.00</td>
</tr>
<tr>
<td></td>
<td>\t \n \r \f \a \e \b</td>
<td>Match, respectively, tab, newline, return, form feed, alarm (bell), escape, backspace</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\xp</td>
<td>Matches ASCII character coding given by pp where pp is in range of 00 to FF</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>[br_el...]</td>
<td>One of. Matches any of the set of characters defined by br_el.</td>
<td>[ab] matches a and b.</td>
</tr>
<tr>
<td>classes</td>
<td>[^br_el...]</td>
<td>None of. Matches anything not in the set of characters defined by br_el.</td>
<td>[^0-9] matches anything not a number</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>Wildcard. Matches anything (including newline)</td>
<td>a.b matches aab, abb, abc, adb, ...</td>
</tr>
<tr>
<td>br_el</td>
<td>p...p</td>
<td>One of. Matches any of the set of characters.</td>
<td>Character class [ab] matches a and b.</td>
</tr>
<tr>
<td>bracket</td>
<td>p-p</td>
<td>Range. Matches any of the set of characters in the range.</td>
<td>[^0-9] matches any character not a digit.</td>
</tr>
<tr>
<td>elements</td>
<td>[:digit:]</td>
<td>Same as 0-9</td>
<td>[0123456789], [0-9], [[:digit:]] are equivalent</td>
</tr>
<tr>
<td></td>
<td>[:alnum:]</td>
<td>Same as A-Za-z0-9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:alpha:]</td>
<td>Same as A-Za-z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:lower:]</td>
<td>Same as a-z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:upper:]</td>
<td>Same as A-Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:xdigit:]</td>
<td>Same as 0-9A-Fa-f</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:blank:]</td>
<td>Space and tab</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[:space:]</td>
<td>Whitespace characters, namely [:blank:] and \r, \n, \f</td>
<td></td>
</tr>
<tr>
<td>Alternation</td>
<td></td>
<td>Either or</td>
<td>grey</td>
</tr>
<tr>
<td>Repeaters</td>
<td>+?</td>
<td>Match 1 or more times, nongreedily.</td>
<td>0+? matches one or more consecutive zeros</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>Match 0 or more times, nongreedily.</td>
<td>.? CALL matches 0 or more &quot;any&quot; characters followed by a space and then CALL</td>
</tr>
<tr>
<td></td>
<td>??</td>
<td>Match 0 or 1 time, nongreedily.</td>
<td>Bee?? matches Bee and Bee</td>
</tr>
</tbody>
</table>
As an example, assume we need to examine a SIP INVITE message to determine if the caller ID is correctly formed, and that a correctly formed caller ID consists of a minimum of 10 characters and a maximum of 20, where each can be a digit or hyphen, then optionally followed by an @ and one of two things following the @ - either an IP address of the form n.n.n.n or a domain name consisting of name.name[name..name]...

We can find the INVITE and Call_ID pieces by the regular expression

```
"INVITE .+? Call_ID: "
```

We can validate the first part of the ID with

```
"[0-9]0-9\-]{9,19}?"
```

which matches a digit followed by 9 to 19 digits and/or hyphens. Then we need to check for one optional following part of the form @n.n.n.n where each n is one to three digits, meaning

```
"@[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}?"
```

and the other optional form (for domain name) is

```
"@\[[[:alnum:]]+\(\.[[:alnum:]]+\)+?"?
```

This we need an either-or-none condition, we would put the above two expressions together inside an ((…)| (…)){1,1 }?, which says match one time either of the expressions in the inner parentheses. 52

Putting this all together in a SCAN expression, we would write 53

```
Rule SCAN(re"INVITE .+? Call_ID: [0-9]0-9\-]{9,19}?\((@[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}?|@[[:alnum:]]+\(.[[:alnum:]]+\)+?)\{1,1}? ",0,0)
```

Of course, with the Define statement, one can pull the detail out of the Rule and have a more readable rule, such as

```
Rule SCAN(look_for_valid_caller_IDs_in_SIP_INVITE_transactions,0,0)
```

52 In general, the use of the “|” operator will cause the implementation to separate this into two separate scans. It may be more efficient in some cases for the PPL programmer to do this explicitly, i.e., a regular expression that looks for a phone number followed by an IP address and another than looks for a phone number followed by a domain name.

53 We will be the first to admit that this is very cryptic, especially compared to the rest of PPL. However, balance this with the power provided by regular expressions and the fact that regular expressions are a fixture understood by many in the software world.
Examples

NAT

NAT can be expressed directly as PPL rules without any special mechanism. Let’s look at an example where we wish to translate between four internal IP addresses and four external addresses.

<table>
<thead>
<tr>
<th>Event (incoming)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule EQ(IP_DEST, 199.100.73.15) SET(IP_DEST, 200.0.0.100)</td>
</tr>
<tr>
<td>Rule EQ(IP_DEST, 199.100.73.16) SET(IP_DEST, 200.0.0.101)</td>
</tr>
<tr>
<td>Rule EQ(IP_DEST, 199.100.73.17) SET(IP_DEST, 200.0.0.102)</td>
</tr>
<tr>
<td>Rule EQ(IP_DEST, 199.100.73.18) SET(IP_DEST, 200.0.0.103)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event (outgoing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule EQ(IP_SOURCE, 200.0.0.100) SET(IP_SOURCE, 199.100.73.15)</td>
</tr>
<tr>
<td>Rule EQ(IP_SOURCE, 200.0.0.101) SET(IP_SOURCE, 199.100.73.16)</td>
</tr>
<tr>
<td>Rule EQ(IP_SOURCE, 200.0.0.102) SET(IP_SOURCE, 199.100.73.17)</td>
</tr>
<tr>
<td>Rule EQ(IP_SOURCE, 200.0.0.103) SET(IP_SOURCE, 199.100.73.18)</td>
</tr>
</tbody>
</table>

The above implements bidirectional NAT with static address assignment, as defined in RFC 2663. Note that other forms may be implemented in PPL logic. For instance, the following rule in the outgoing event

```
Rule SET(IP_SOURCE, IPLIST(0))
```

converts all internal addresses to the address in IPLIST(0). By itself, this is not particularly useful, because there is insufficient information to do the corresponding translation on the other path. However, if one knows the ports being used by the applications on the “inside systems” and can overload (by modifying) the source port number, one can write a stateless implementation of NABT (see RFC 2663) where the internal IP addresses are hidden behind one external IP address.

In writing rules for NAT, one needs to be aware of the shortcomings of NAT (e.g., if the application protocol embeds IP addresses in the layer-7 information, if the packet is a transport mode IPSec packet).

There may be cases where we wish to do address translation across a large number of addresses, where using individual rules is inefficient. We can easily accomplish this by bit manipulation (e.g., for subnetting) or by using an array if the translation is irregular. For instance, if we wish to translate addresses of the form 190.10.10.X to 192.68.0.X, we can write this as

```
Rule EQ(IP_DEST/24, 190.10.10.0) SET(IP_DEST, IP_DEST & 0xFF) SET(IP_DEST, IP_DEST | 192.68.0.0)
```

If we wish to translate 190.10.10.X to 256 unique IP addresses, we can do a table lookup, e.g.,
PPL Packet Processing Language

Intrusion Detection

It is relatively simple to use SCAN with either simple strings or regular expressions to accomplish scans for intrusion detection. One way to do this is to convert SNORT rules to PPL rules. For instance the following SNORT rule detects the eurocalculator virus

Alert tcp any any -> any 25 (msg:Virus – Successful eurocalculator execution”; flags:PA; content: funguscrack@hotmail.com; nocase; sid:736; classtype:misc-activity; rev:3;)

as PPL rule

Euroc736: Policy PROGRAM FUNCTION(alert) DATA(code_eurocalculatorvirus)
Rule EQ(L4_DPORT,25) EQ(TCP_PSH,1) EQ(TCP_ACK,1)
SCAN(ul"funguscrack@hotmail.com") APPLY(Euroc736)

And the following SNORT rule to detect the subseven 22 backdoor

Alert tcp $EXTERNAL_NET 27374 -> $HOME_NET any (msg:“BACKDOOR subseven 22”; flow:to_server,established; content:”|0d0a5b52504c5d3030320d0a|”; reference:arachnids,485; reference:url,www.hackfix.org/subseven/; classtype:misc-activity; sid:103; rev:5;)

Subsevensid103: Policy PROGRAM FUNCTION(alert) DATA(code_backdoor_subseven22)
Rule EQ(IP_PROT,TCP) EQ(L4_SPORT,27374) EQ(CX_STATE,ESTABLISHED)
SCAN(”|0d0a5b52504c5d3030320d0a|”) APPLY(Subsevensid103)
# Signature from arachnids IDS485

A large number of scans cannot always be done without an impact in performance. One can combine the power of the association tables with scan to make blocking intruders much more efficient. The concept is that when a signature is detected, we put the intruding source IP address in a table, and prior to performing the scans for signatures, we look up each packet’s source IP address in the table.54 The code below is a small example of how to do this. In this code, we assume a stern approach to potential intrusions – we drop the packets instead of just composing an alert.

Define sid=“RE11”
Define ABSOLUTE=“1”
Intruderlist: POLICY ASSOCIATE NUMBER(1000) SEARCHKEYS(IP_SOURCE)

54 This does of course have a shortcoming in the case where the intruder is spoofing the source IP address, especially if the intruder spoofs with an IP address of a legitimate user. We could extend the code to check for this.
TIMEOUT(ABSOLUTE,6000000)  #delete each after ~100 min

Intrudertest: POLICY RECALL LINKED(Intruderlist) SEARCHKEYS(IP_SOURCE)
Repeatintruder: POLICY PROGRAM FUNCTION(alert)
DATA(code_repeatintruder,IP_SOURCE)
Newintruder: POLICY PROGRAM FUNCTION(alert)
DATA(code_newintruder,IP_SOURCE,sid)

Rule APPLY(Intrudertest) NE(RR0,FuF) ALERT(Repeatintruder) DROP STOP

Note that there are several ways to organize the possible long list of scanning rules. One might have the urge to want to place each in a run group (i.e., specify “Run Rule”) for each, but this doesn’t add any parallelism because run groups affect only actions, and the scans by definition are part of the rule evaluation and thus done in parallel. In the approach we took above, we will determine if we should perform the action of each and every rule, even in the worst case where the first rule is true. If we wanted to optimize this, and/or if we wanted to optimize the program under the assumption that the implementation can’t evaluate an endless number of rules in parallel, we could add a JUMP action to each of the rules. For a large number of scans, the PATTERNS policy is a considerably faster alternative.

DOS Attack

There are many many rules that can be written to detect DOS attacks. As an example, Bubonic is an attack that attempts to overload a system with pseudo-random TCP packets. There is one understood signature using some layer 3 fields.

Datagram Reconstruction from Fragments

We show a program here that keeps track of fragments (whose order of arrival is unpredictable) from multiple datagrams and builds whole datagrams as packets from them. This program is a good illustration of the ASSOCIATE, RECALL, and NEWPACKET policies. Note that an easier way to develop the same program would be to use superpackets. Easier yet – use the DEFRAG policy.
# fragments. We might do this if we were building an “out of line” intrusion
detection system and we wished to assemble a complete datagram so that we
weren’t tricked by signatures that overlap fragments.

# The strategy we will use here is to create a fixed number of maximum size
datagrams (64K long with 60 byte IP headers) at startup time. We will keep
their packet handles in a table. We will also use a table with the same
number of rows to track fragmented datagrams arriving. This table will use
the search keys: IP source and destination addresses, protocol, identifier.

# In this example we don’t handle overlapping fragments. We assume we have
# the whole datagram when the number of data bytes computed from the MF=0
# packet equals the number of bytes copied to our constructed datagram.

# The program logic for handling fragments is summarized as:
# 1. Check the fragment for fragment-based attacks (teardrop, teardrop2,
# ping of death, etc)
# 2. If this fragment part of a new datagram we don’t know about? If yes,
# “allocate” one of the datagram buffers and for now indicate that the
# fragment must be part of a maximum length datagram. Remember the
# “datagram signature” (IP addresses, protocol, identifier)
# 3. Then if this is a “first” fragment (offset=0), copy the IP header and
# options to the datagram buffer.
# 4. Then if this is a “last” fragment (MF=0), compute the implied size of
# the original unfragmented datagram and store this.
# 5. Then copy the data of the fragment into the datagram buffer.
# 6. Compute if we have the whole datagram. If so, make it the current
# packet, fix up the remaining fields in its header, and destroy the
# table entry.

Define start_up = "999"
Define max_fraged_datagrams="64"
Define dg_buff_headerlength="20"   #Assumed IP hdr length in datagram buffers
Define ac="RR0"
Define Regdatatodate="RE0"
Define Regfullsize="RE1"
Define Fdrow="Re2"
Define Regdbuffhandle="RE3"
Define Regoffset="RE4"

#
dg_buff_table: Policy ASSOCIATE NUMBER(max_fraged_datagrams)
    SEARCHKEYS(ac)
        get_dg_buff_handle: Policy RECALL LINKED(dg_buff_table)
            ROW(ac) SEARCHKEYS(Regdgbuffhandle)

#
fraged_datagrams: Policy ASSOCIATE NUMBER(max_fraged_datagrams)
    SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
    TIMEOUT(ABSOLÜTE,30000)
    one_underway: Policy RECALL LINKED(fraged_datagrams)
        SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
    destroy_fraged: Policy DISASSOCIATE LINKED(fraged_datagrams)
        SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
    Datatodate: ARRAY(max_fraged_datagrams)
    Fullsize: ARRAY(max_fraged_datagrams)

#
# dg_buff_table is a table of datagram buffer handles created at startup time
# fraged_datagrams is a table of same number of rows. An entry exists if we
# are in the midst of building a datagram from fragments. An entry contains
# the “datagram signature” and two corresponding arrays contain data fields:
# the data we’ve copied to data, and the known full datagram size (or max if
# not known). Entry disappears in 30 seconds if not accessed.

dgbuff: Policy NEWPACKET SIZE(65536) HEADERS(0,20,20)
copy_header: Policy PACKET COPY(Regdgbuffhandle,0,0,dg_buff_hdrlength)
copy_frag: Policy PACKET COPY(Regdgbuffhandle,Regoffset,PS_IPDATAOFFSET,PS_IPDATASIZE)
make_curr: Policy PACKET CURRENT(Regdgbuffhandle)

# Event(start_up)
Rule SET(ac,max_fraged_datagrams - 1)
Loop:
Rule APPLY(dgbuff) # Create a max-length datagram buffer
  APPLY(db_buff_table) # Put its handle in table
  SET(ac,ac - 1)
  GE(ac,0) JUMP(Loop) # Do max_fraged_datagrams times

Event(0)
  # We assume we’ve reached this point in the situation where we have a
  # packet representing a fragment i.e., either MF=1 or frag offset not = 0
  # Step 1 of 6
Rule EQ(PS_NOORFIRSTFRAG,1) EQ(PS_TCPUDP,1) LE(PS_IPDATASIZE,20) JUMP(XXXX)
  # Don’t accept a first fragment of TCP/UDP protocol not big enough to
  # hold the TCP or UDP header
Rule EQ(IP_MF,1) LE(PS_IPDATASIZE,50) JUMP(XXXX)
  # Arbitrarily, don’t accept small fragments other than last fragment
Rule NE(PS_NOORFIRSTFRAGMENT,0) LE(IP_FRAGOFFSET,7) JUMP(XXXX)
  # Check for Teardrop – an attempt in other than first fragment to specify
  # an offset too low – e.g., to overwrite the layer 4 header
Rule SET(ac,IP_FRAGOFFSET >> 3) SET(ac,ac + PS_IPDATASIZE)
  GE(ac,65516) JUMP(XXXX)
  # Check for a set of attacks – Teardrop2, ping of death, etc – an attempt
  # in any fragment to describe a fragment that would exceed maximum
  # datagram size
  # Step 2 of 6
Rule APPLY(one_underway) # See if this fragment is part of something
  NE(ac,FuF) SET(fdrow,ac) JUMP(inprocess) # Jump if we know about it.
  Rule SET(Regdatatodate,0)
  APPLY(fraged_datagrams) # datagram signature and initial data items
  EQ(ac,FuF) JUMP(XXX) # Exception case where we have too many
  # simultaneous frag’d datagrams to handle
inprocess:
  # here ac is row number of table entry
Rule APPLY(get_dg_buff_handle) # returns Regdgbuffhandle
  # Step 3 of 6
Rule EQ(PS_NOORFIRSTFRAGMENT,1)# If this is the lowest offset fragment, copy
  APPLY(copy_header) # the IP hdr (for simplicity, ignore options
  # Step 4 of 6
Rule EQ(IP_MF,0) # If the highest offset fragment, compute and
# store now-known datagram len in table entry
SET(Regfullsize,IP_FRAGOFFSET >> 3)
SET(Regfullsize,Regfullsize + PS_IPDATASIZE)
SET(fullsize(dfrow),Regfullsize)

# Step 5 of 6
SET(Regoffset,IP_FRAGOFFSET >> 3)
SET(Regoffset,Regoffset + dg_buff_hdrlength)# Compute offset in dg buff
APPLY(copy_frag)                     # copy fragment in
SET(datatodate(dfrow),datatodate(dfrow) + PS_IPDATASIZE)

# Update table entry with new data

# Step 6 of 6
GE(datatodate(dfrow),Regfullsize)  # Have all fragments apparently?
APPLY(make_curr)                   # Switch packets, now datagram buffer
SET(IP_PACKETLEN,Regfullsize)      # Update correct length in new hdr
APPLY(destroy_fragged)             # Destroy the matching entry

### Finding and Converting an IPv6 address in a Layer 7 Protocol

The following looks for something that appears to be an IPv6 address in the packet payload and converts it to a binary IPv6 address. It accounts for the “::” zero-compression notation. It leaves a 128-bit binary IPv6 address in register Rr0.q.

Define nums = "\[0-9A-F\]{1,4}?"  
Rule SCANB(reul"(nums:){7,7}?nums |nums:(nums){0,6}?:: |::(nums:){0,6}?nums | nums:(nums){0,5}?::(nums:){0,5}?nums ")
NE(Rr0,FuF) COMPUTE(CBIP,Rr0.q,Rr0)
An Approach to Handling Fragments in the Normal Course

# The following is an example of one approach to handling IP fragments in a # “bump-in-the-wire” application. The strategy is the following:
#
# 1. When we make a decision about something we want to do with a packet ( #   (e.g., drop it, forward it to a specific IP address as a result of a #   load balancing decision) and we notice that the packet is actually #   a first-fragment of a longer datagram (MF flag is set and frag offset #   is zero), we will remember the packet via its “datagram signature” (IP #   addresses, protocol, identifier) in a table.
# 2. When we encounter a fragment that is other than the lowest-offset #   fragment of a datagram, we will do one of two things:
#    a. If we find its signature in the table, we will do what the table #       designates
#    b. Otherwise we will punt and just discard the fragment.
#
# The 2b action seems a bit drastic, but it should be OK in practice, for # three reasons. First, fragmentation is anticipated to be a rare # occurrence. Second, although fragments can arrive in any order, we’d # expect the lowest fragment (offset 0) to arrive first most of the time, and # we’d expect the most common case of fragmentation to be a two-fragment # datagram. Third, when we do decide to drop a fragment, retransmission of # the full datagram will occur in the near future.

frag_tracker: Policy ASSOCIATE NUMBER(10000)
   SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
   TIMEOUT(Absolute,30000)
we_know_frag: Policy RECALL LINKED(frag_tracker)
   SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
destroy_track: Policy DISASSOCIATE LINKED(frag_tracker)
   SEARCHKEYS(IP_SOURCE,IP_DEST,IP_PROT,IP_IDENTIFIER)
IP_addr: ARRAY(10000)

# Insert the following code at a point(s) in the larger program where some # overt action has been determined to be taken with the current packet. Here # as an example we will assume the action is a FORWARD action with a specific # IP address

Rule EQ(PS_FRAGMENT,1) EQ(PS_NOORFIRSTFRAGMENT,1) GE(PS_IPDATASIZE,20) APPLY(frag_tracker) SET(IP_addr(Rr0),RR1) # ... but we also see our packet is the first fragment. Create table entry and store new IP destination that was in RR1 in the array. If full don’t care. But don’t let someone sneak by with a too-small first fragment

# Insert the following code at some other point in the larger program where # we want to handle the special case of fragments

Rule NE(PS_NOORFIRSTFRAGMENT,1) APPLY(we_know_frag) EQ(RR0,FuF) DROP STOP # If fragment other than the lowest check for signature in table Just drop it if we don’t know it
Rule NE(RR0,FuF) SET(IP_DEST,IP_ADDR(Rr0)) FORWARD STOP

# Note that in this situation there is no known good time when we can destroy # the frag_tracker table entry. Thus we don’t use the destroy_track policy # and we need to rely on just timeout specified in frag_tracker.
Currently Unimplemented Language Parts and Restrictions

This section lists those aspects of the PPL language that will not likely be implemented in the first product.

Dynamic compilation.

Run groups. Run and Wait statements will be allowed, but they will not generate a further level of concurrency.

Events. There can be a maximum of 255 events (255 event handlers, although there can be more than 255 logical port numbers in use)

Associate/Recall/Disassociate policies. Only one searchkey parameter is supported. It can be 128 bits, so all of the examples herein can be implemented simply by mapping multiple search key parameters into one 128-bit value.

Timeout and reset times. If expressed to be larger than approximately 49 days, the implementation reduces the value to approximately 49 days. More importantly, one should restrict timeout intervals to a maximum of approximately seven days because a “wrapped-around” time (as a 32-bit value) cannot be reliably detected if the interval is significantly larger than seven days.

Virtual network value. Must be in the range 0..4095 (same as 802.1Q Ethernet VLAN tag). If > 4095 but not Fuf, an invalid-value exception (3) is generated.

Invalid packet handles. A very high percentage, but not 100%, of random values are detected as invalid packet handles. All zero and FuF are certainly detected as invalid.

PACKET policy header insertion/strip. If there is insufficient space such that the only choice is moving the whole packet, this is not done and the insufficient-storage exception occurs.\(^{55}\) In STRIP PREP, the L2 header’s (if present) values are not guaranteed to be moved correctly if the strip amount is less than the size of the L2 header after strip (i.e., if moving the header “to the right” causes it to overwrite itself).

COMPUTE CBN function. Maximum number of digits that will be converted is 16.

CIPHER HASH IPAD function. In the IXP2850 implementation, one can use this to accumulate the IPAD digest in the current hash state (“on the fly”) or to precompute the IPAD digest, but not both. If the DIGEST keyword indicates that the IPAD digest will be returned to the program, it will not also be accumulated in the ongoing hash correctly. This is a minor restriction because it is unlikely to want to do both together.

\(^{55}\) See the PREPEND option in the IXP2xxx DeviceMap, which allows one to ensure a certain amount of space is reserved at the front of packets.