Building a Custom Tunnel with P4

In this lab you will be defining a custom protocol in P4. This will require revising the match tables and creating actions to encapsulate and decapsulate the custom header. The protocol we will define is called “xlan”. It will be used to tag Ethernet traffic with a 32-bit tag. Note, completing lab 1 is a prerequisite for this tutorial, as the instructions in this lab build on the work done previously.

1. Open the P4 source file (e.g. “p4lab.p4”) and amend the header type and header declaration section to appear as follows.

```
header_type eth_hdr {
  fields {
      dst : 48;
      src : 48;
      etype : 16;
  }
}

header_type xlan_hdr {
  fields {
      tag : 32;
      etype : 16;
  }
}

header eth_hdr eth;

header xlan_hdr xlan;
```

2. Now we describe how the switch will parse our new header. First, we need a new Ethernet type to indicate the presence of our header. Define it as follows near the ***xlan_hdr*** definition:

```
#define XLAN_ETYPE 0x9999
```

Next, modify the Ethernet parser to identify xlan for further parsing. Finally, add the xlan parser.

```
parser start {
  return eth_parse;
}

parser eth_parse {
  extract(eth);
  return select(eth.etype) {
      XLAN_ETYPE: xlan_parse;
      default: ingress;
  }
}

parser xlan_parse {
  extract(xlan);
  return ingress;
}
```

3. The next step is to modify the ingress data flow and table matching to account for the new encapsulation processing the switch will perform. First, change the ingress table name from ***in_tbl*** to the more suitable name ***encap_tbl***. This requires changing both the invocation of the table in the "control" statement and the name of the table in the table declaration.

```
control ingress {
  apply(encap_tbl);
}
```

Next, tweak the existing ingress table to reflect the fact that it is going to perform a new encapsulation action named ***encap_act***. This action will both encapsulate the packet and select its output port.

```
table encap_tbl {
  reads {
      standard_metadata.ingress_port : exact;
  }
  actions {
      encap_act;
      drop_act;
  }
}
```

4. Now modify the ingress action to encapsulate the ethernet payload with an xlan tag. The ***encap_act*** still starts by selecting the packet's output port. It then invokes a new action named ***xlan_encap*** to perform the actual encapsulation. In the P4 language, a table can specify only a single action to perform on a match. Thus, P4 programs must include actions that refer to the combinations of operations that the switch must actually carry out.

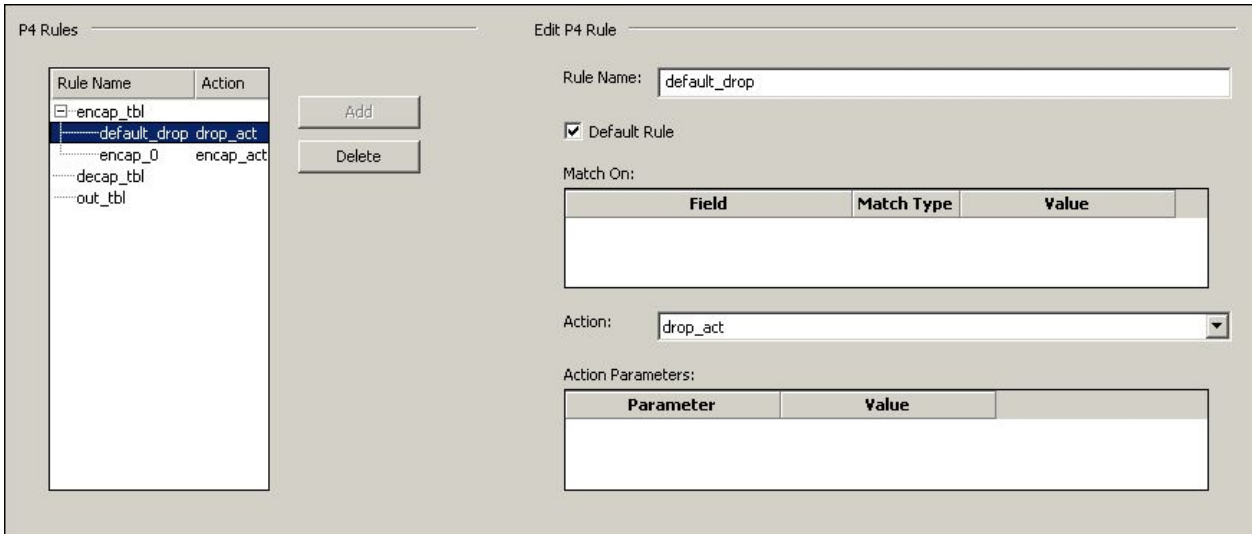
The ***xlan_encap*** action first adds the new xlan header and sets its tag as specified by the parameter specified in the match table rule. It then copies the Ethernet ethertype field into the xlan ethertype field and sets the Ethernet ethertype field to the xlan ethertype we defined earlier (0x9999).

```
action encap_act(prt, tag) {
  modify_field(standard_metadata.egress_spec, prt);
  xlan_encap(tag);
}

action xlan_encap(tag) {
  add_header(xlan);
  modify_field(xlan.tag, tag);
  modify_field(xlan.etype, eth.etype);
  modify_field(eth.etype, XLAN_ETYPE);
}
```

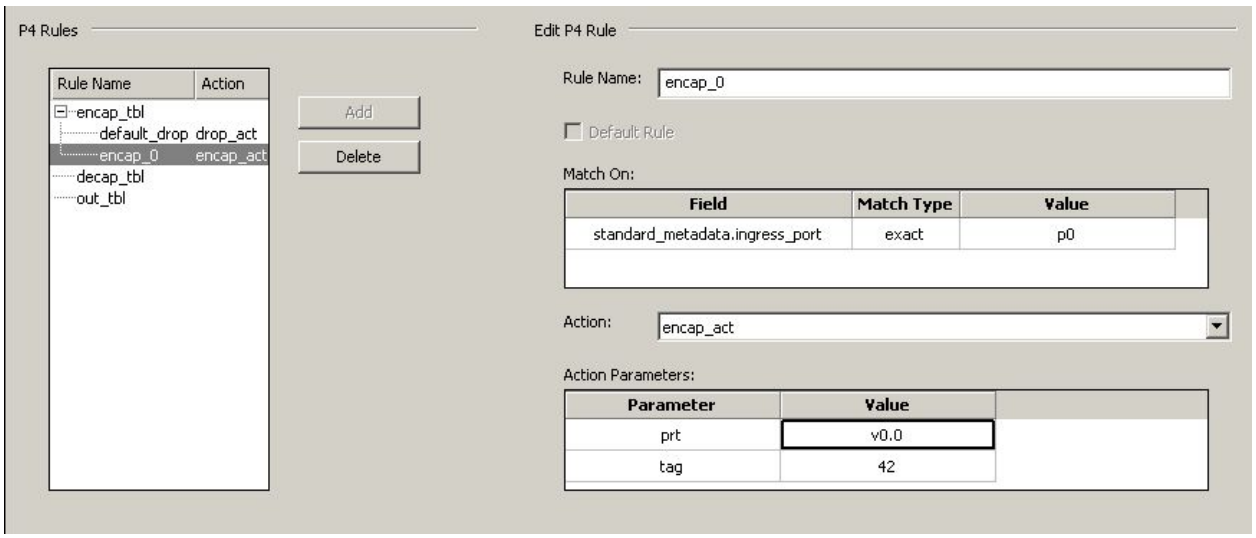
5. Now that the P4 program modifications are complete we need to invoke it with appropriately populated match/action tables. We will create two rules in the ***encap_tbl***: a default rule to drop packets and a rule to encapsulate packets.

First, press Ctrl-S to save the changes, and press F7 to rebuild the project. This ensures that the rules editor becomes aware of the new actions.

Now open the rules editor and delete the old rules. Next, add the drop rule as shown:



We will encapsulate traffic arriving on physical port 0 and forward to virtual port 0 on the host. To do this create a rule that matches standard_metadata.ingress_port to p0 and invokes the action ***encap_act***. Set the ***prt*** parameter of the action to v0.0 and the ***tag*** parameter to 42.



6. We are now ready to test this encapsulation. First, run the program by pressing F12 and F5 as in previous labs.

Next, prepare start tcpdump on the vf0 interface:

```
tcpdump -i vf0 -xxx -n
```

Now inject a packet on the physical interface:

```
tcpreplay -i p4p1 udp.pcap
```

This displays the following output:

```
sending out p4p1
processing file: udp.pcap
Actual: 1 packets (58 bytes) sent in 0.02 seconds.
Rated: 2900.0 bps, 0.02 Mbps, 50.00 pps
Statistics for network device: p4p1
     Attempted packets:         1
     Successful packets:        1
     Failed packets:            0
     Retried packets (ENOBUFS): 0
     Retried packets (EAGAIN):  0
```

The tcpdump program should now display the encapsulated packet:

```
14:40:30.726060 11:22:33:44:55:66 (oui Unknown) > 22:33:44:55:66:77
(oui Unknown), ethertype Unknown (0x9999), length 66:
     0x0000:  2233 4455 6677 1122 3344 5566 9999 0000
     0x0010:  0082 0800 4500 002c 0001 0000 4011 665c
     0x0020:  0a00 0001 0a00 0064 0bb8 0fa0 0018 97c1
     0x0030:  0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
     0x0040:  0000
```

Stop tcpdump by pressing Ctrl-C and stop debugging by pressing Ctrl-F12 in Programmer Studio.

7. Now we will add decapsulation to the P4 program, starting with modifying the control flow. This requires rewriting the ingress control block to execute a match table called ***decap_tbl*** if the switch detects an xlan header:

```
control ingress {
  if (valid(xlan)) {
      apply(decap_tbl);
  } else {
      apply(encap_tbl);
  }
}
```

8. Now we must add the table to either drop or decapsulate incoming xlan packets depending on the ingress port:

```
table decap_tbl {
  reads {
      standard_metadata.ingress_port : exact;
  }
  actions {
      decap_act;
      drop_act;
  }
}
```

9. At this point, the program needs a decapsulation action which we name ***decap_act***. In the decapsulation action, as with encapsulation, the action first sets a forwarding port. Next, it moves the xlan ethertype to the Ethernet ethertype. Finally it removes the xlan header from the packet.

```
action decap_act(prt) {
  modify_field(standard_metadata.egress_spec, prt);
  modify_field(eth.etype, xlan.etype);
  remove_header(xlan);
}
```

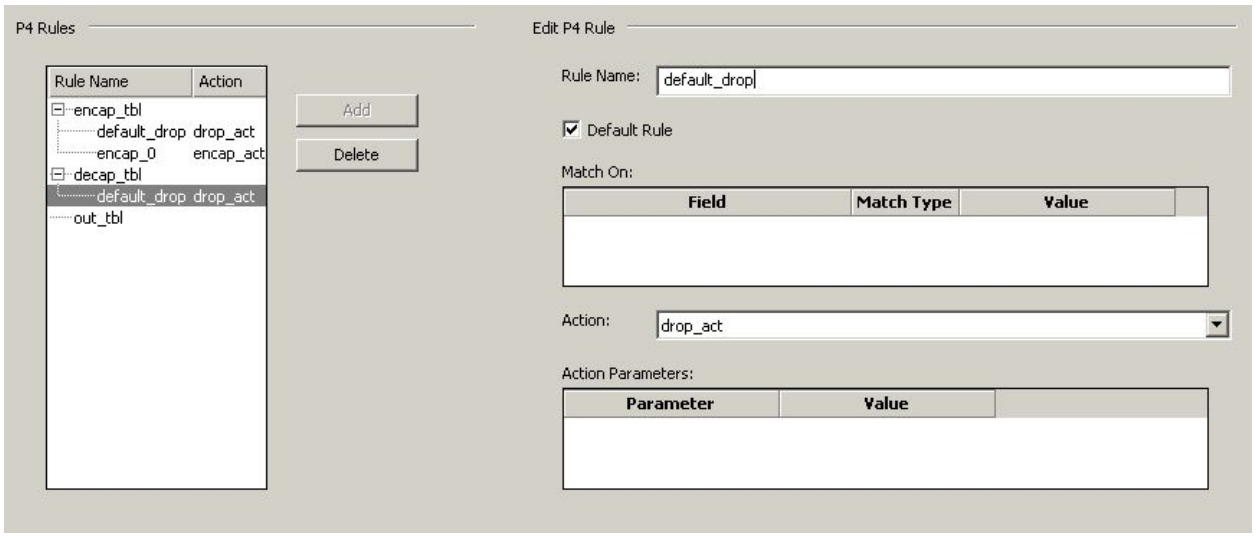
10. While adding our decapsulation logic let us also add some P4 counters to provide encapsulate and decapsulate statistics. First instantiate two global P4 counters as below. Create one packet and one byte counter:

```
counter encap_counter {
  type : packets;
  instance_count : 1;
}

counter decap_counter {
  type : bytes;
  instance_count : 1;
}
```

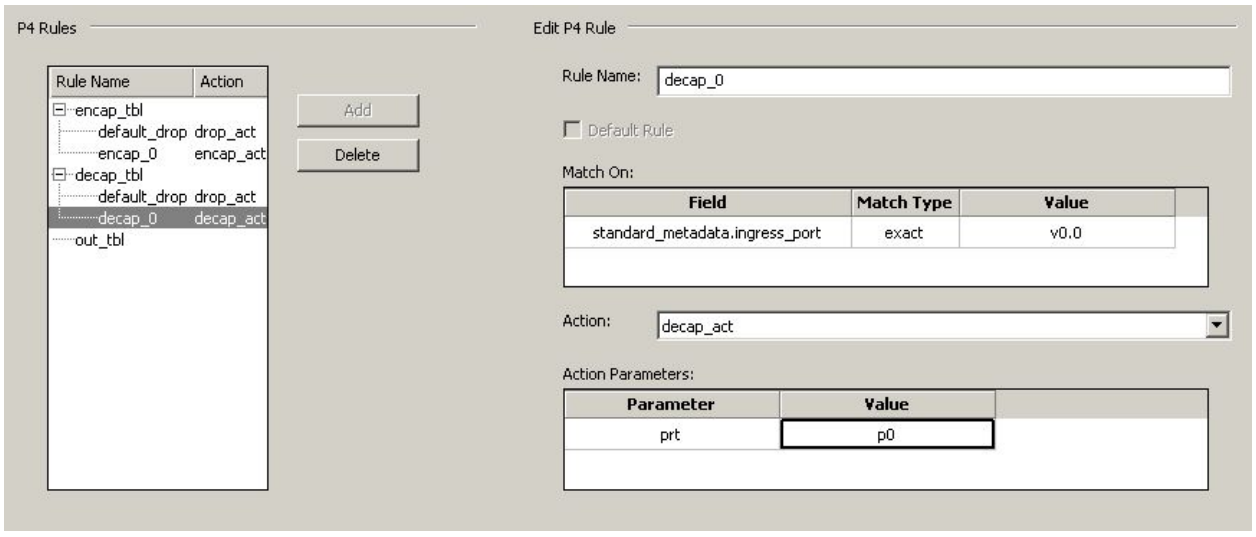
Then, add count primitives to the encapsulate and decapsulate actions:

```
action xlan_encap(tag) {
  add_header(xlan);
  modify_field(xlan.tag, tag);
  modify_field(xlan.etype, eth.etype);
  modify_field(eth.etype, XLAN_ETYPE);
  count(encap_counter, 0);
}

action decap_act(prt) {
  modify_field(standard_metadata.egress_spec, prt);
  modify_field(eth.etype, xlan.etype);
  remove_header(xlan);
  count(decap_counter, 0);
}
```

11. First add a default drop rule:



Now we add a decapsulate rule to forward the virtual port 0 to the physical port:



12. To test decapsulation, start by loading and running the P4 program - press F12 and F5 in Programmer Studio. After that, start tcpdump to display the forwarded packet:

```
tcpdump -i vf0 -xxx -n
```

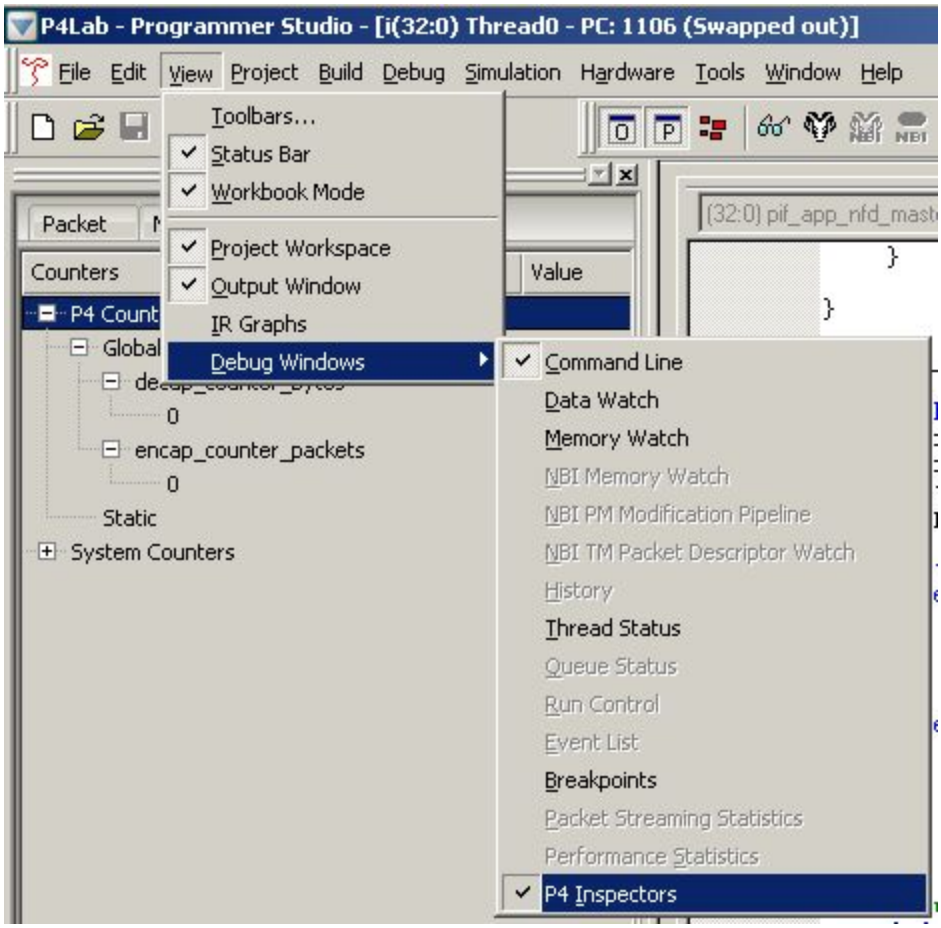
Next, inject an xlan packet on the physical interface:

```
tcpreplay -i p4p1 xlan.pcap
```

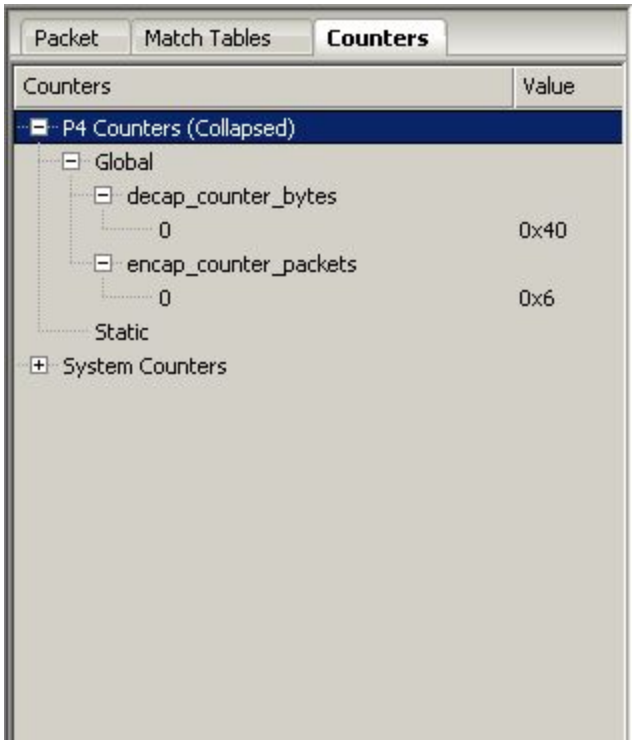
The tcpdump program should now show the decapsulated packet:

```
14:59:08.380695 IP 10.0.0.1.3000 > 10.0.0.100.4000: UDP, length 16
      0x0000:  2233 4455 6677 1122 3344 5566 0800 4500
      0x0010:  002c 0001 0000 4011 665c 0a00 0001 0a00
      0x0020:  0064 0bb8 0fa0 0018 97c1 0001 0203 0405
      0x0030:  0607 0809 0a0b 0c0d 0e0f 0000
```

13. At this point, we can inspect the P4 counters. First open the P4 inspectors:



The P4 counters should reflect the forwarded packets.



We have now implemented a custom tunneling protocol and introduced counters to monitor forwarding behavior.