Security Apps with P4 Programmable Switches

Introduction to Packet Parsing and Match-action Tables

Elie Kfoury, Jorge Crichigno
University of South Carolina
http://ce.sc.edu/cyberinfra

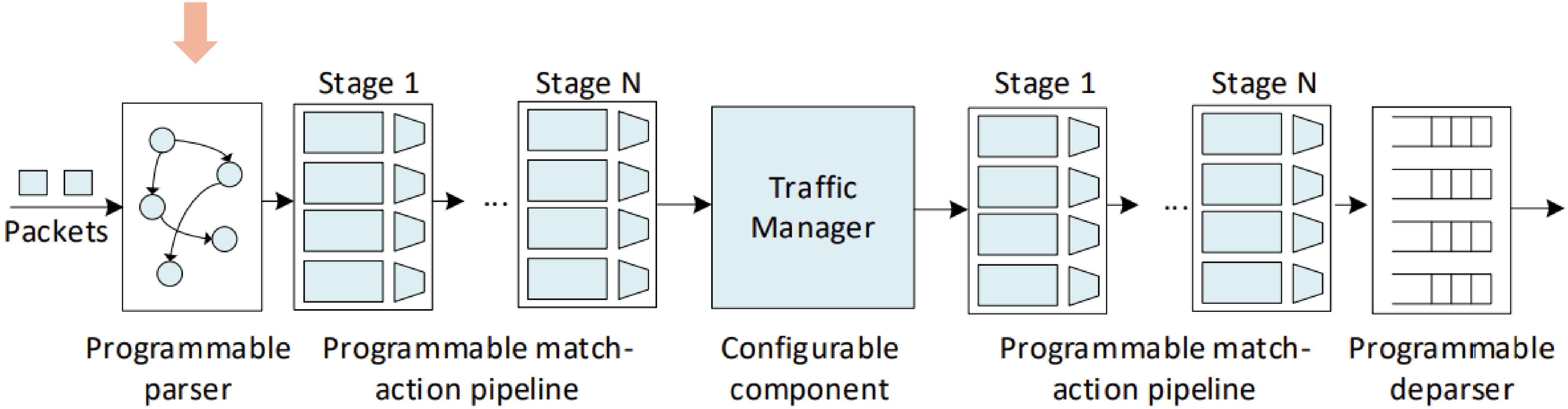
University of South Carolina (USC)
Energy Sciences Network (ESnet)

September 18, 2023

Parser

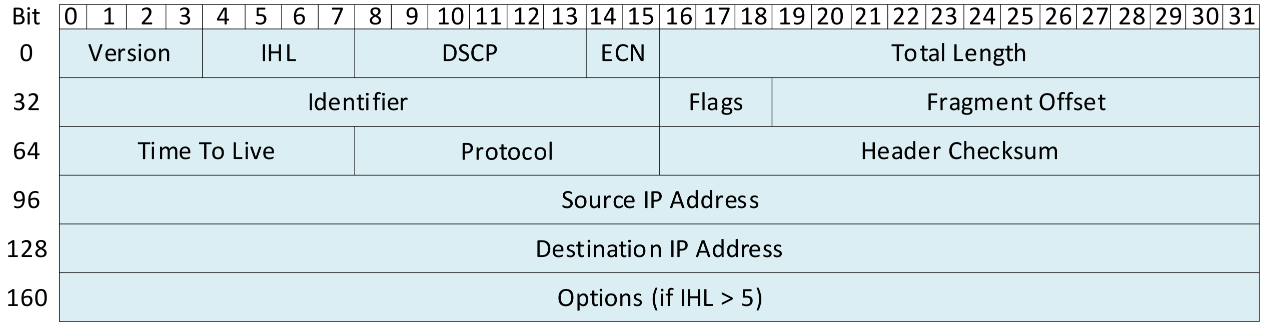
Programmable Parser

- The parser enables parsing arbitrary headers with a finite state machine
- The state machine follows the order of the headers within the packets
- The packet is split into the defined headers and the remaining is treated as the payload

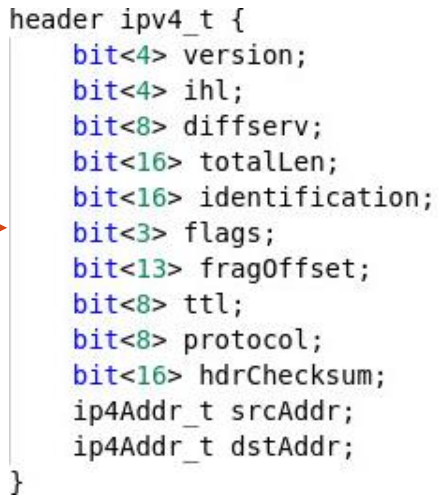


Packet Headers

- The packet headers are specified by the programmer
- The programmer has the flexibility of defining custom/non-standardized headers
- Such capability is not available in non-programmable devices



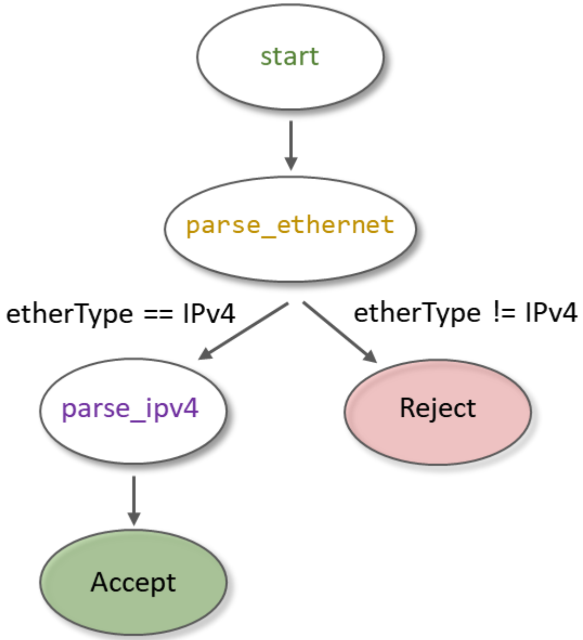
<table>
<tr><td>Bit</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>32</td><td colspan="16">Identifier</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>160</td><td colspan="32">Options (if IHL &gt; 5)</td></tr>
</table>

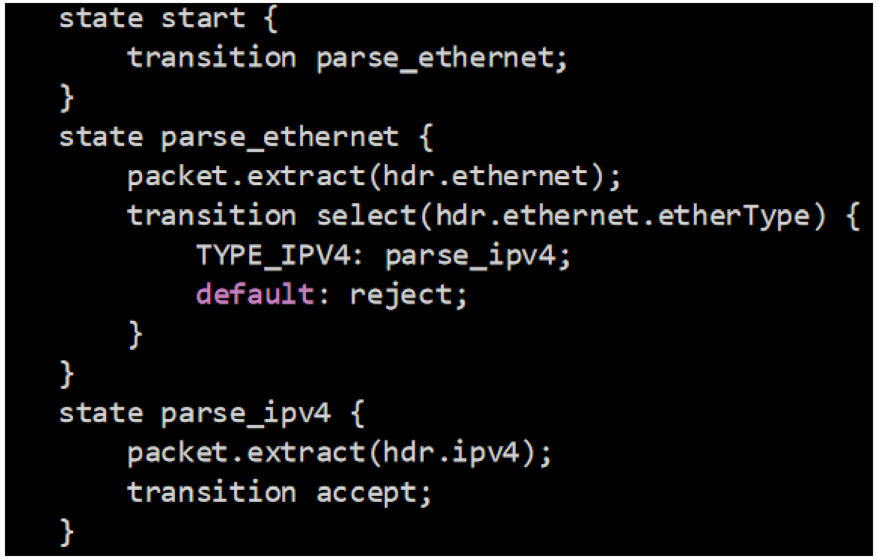


```
header ipv4_t {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

Programmable Parser

- Every parser has three predefined states: start, accept, and reject
- Other states may be defined by the programmer
- In each state, the parser executes statements and then transitions to another state



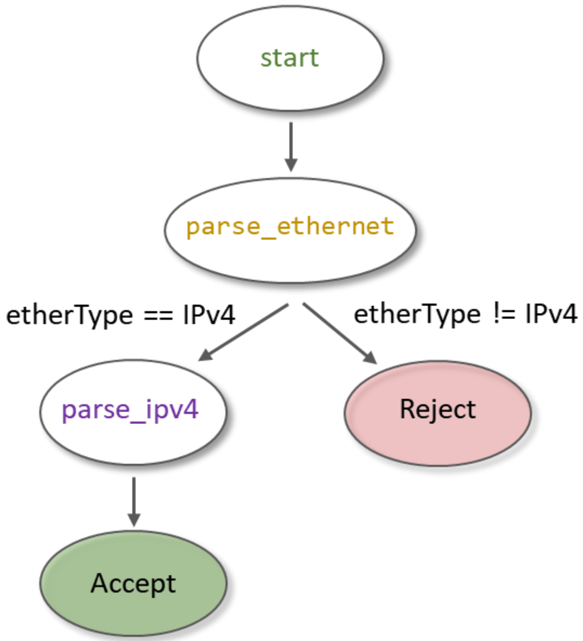


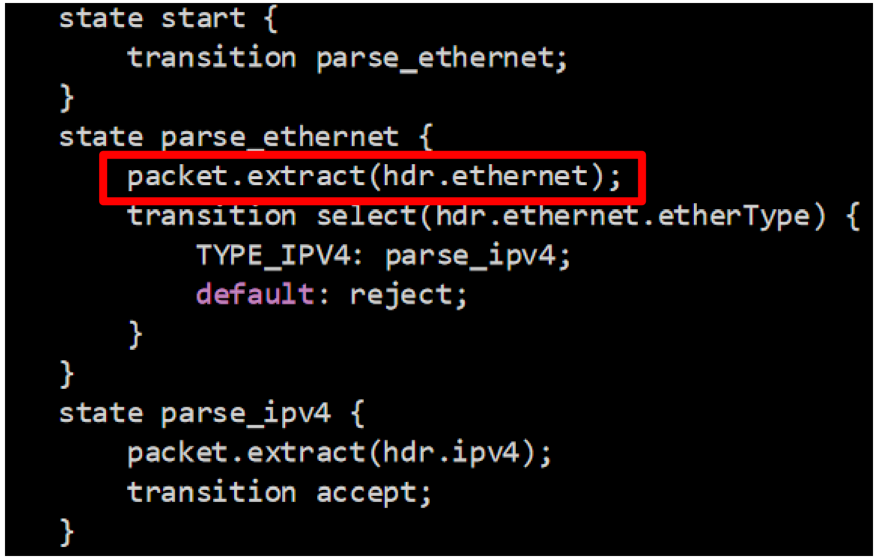
```
state start {
    transition parse_ethernet;
}
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        default: reject;
    }
}
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```

`packet` is an input parameter; `hdr` is an output parameter

Programmable Parser

- P4$_{16}$ has an extract method that can be used to “fill in” the fields of a header from the “raw” packet



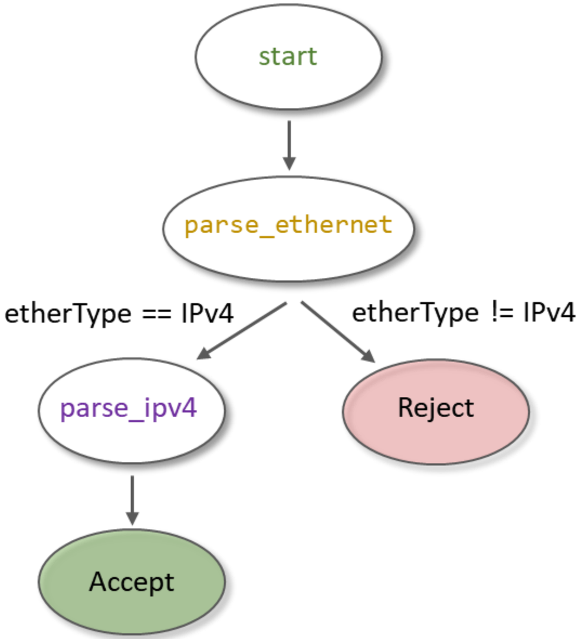


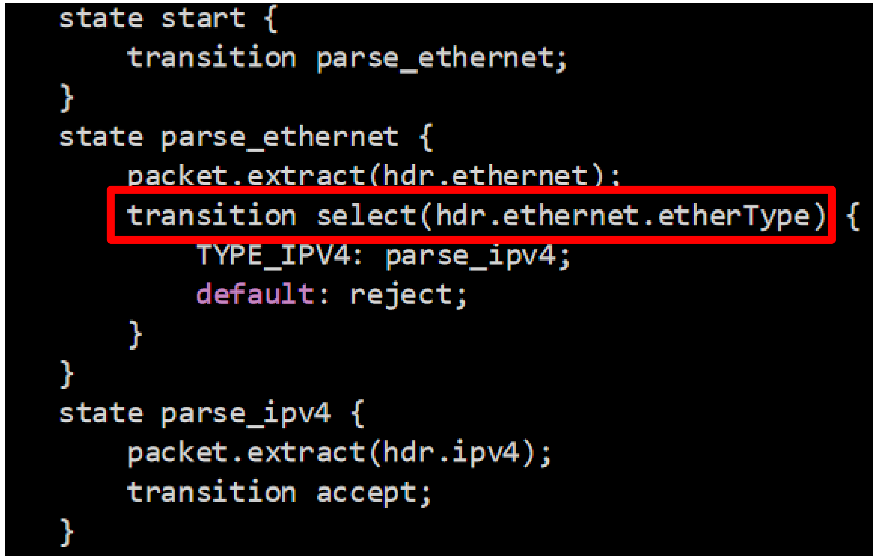
```
state start {
    transition parse_ethernet;
}
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        default: reject;
    }
}
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```

`packet` is an input parameter; `hdr` is an output parameter

Programmable Parser

- $P4_{16}$ has a select statement that can be used to branch in a parser





```
state start {
    transition parse_ethernet;
}
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_IPV4: parse_ipv4;
        default: reject;
    }
}
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition accept;
}
```

`packet` is an input parameter; `hdr` is an output parameter

Headers Format

- Ethernet header:

48 bits	48 bits	16 bits
Destination Address	Source Address	Ether Type

- IPv4 header:

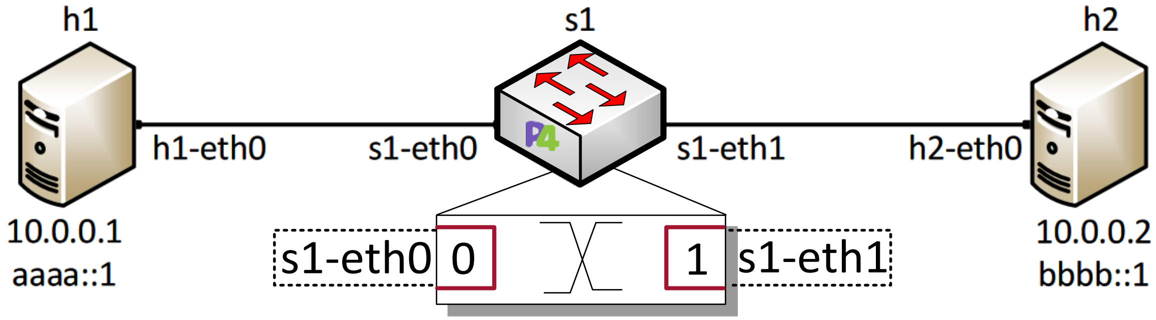
<table>
<tr><td>Bit</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>32</td><td colspan="16">Identifier</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>160</td><td colspan="32">Options (if IHL &gt; 5)</td></tr>
</table>

- IPv6 header:

<table>
<tr><td>Bit</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td colspan="4">Version</td><td colspan="8">Traffic Class</td><td colspan="20">Flow Label</td></tr>
<tr><td>32</td><td colspan="16">Payload Length</td><td colspan="8">Next Header</td><td colspan="8">Hop Limit</td></tr>
<tr><td>64</td><td colspan="32">Source IP Address</td></tr>
<tr><td>192</td><td colspan="32">Destination IP Address</td></tr>
</table>

Lab 4 Topology and Objectives

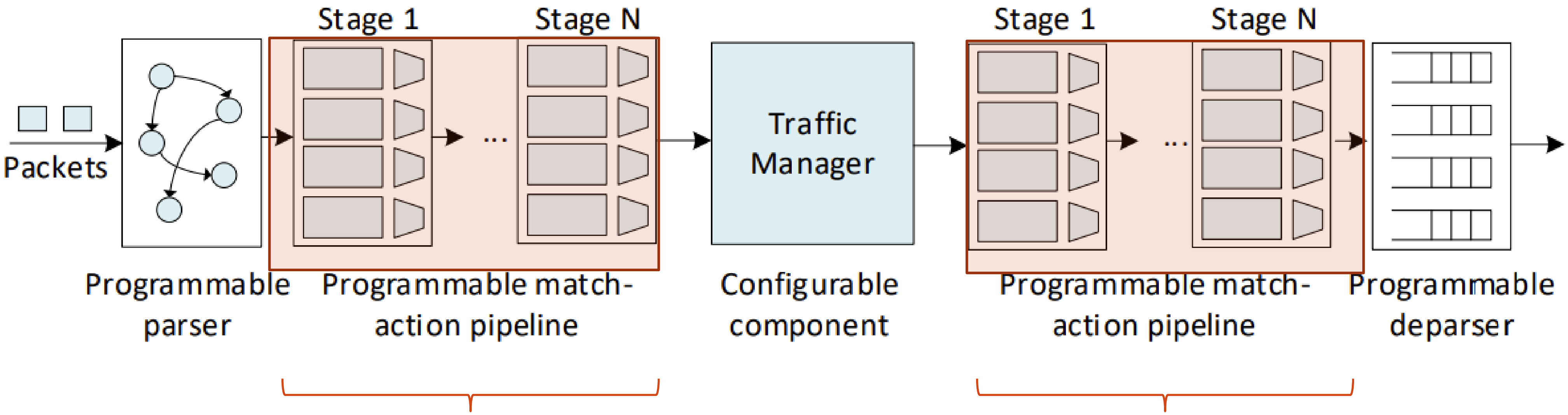
- The topology consists of two hosts: h1 and h2; one P4 switch: s1
- The objectives are:
  - Defining the headers for Ethernet, IPv4 and IPv6
  - Implementing the parser
  - Testing and verifying the switch behavior when IPv4 and IPv6 packets are received



Match-action Tables

Match-action Pipeline

- Tables are the fundamental unit of a Match-Action Pipeline; they define the processing logic inside the match-action pipeline
- They can be used to implement traditional switch tables (e.g., routing, flow lookup, access-control lists)
- They can implement custom user-defined complex logic



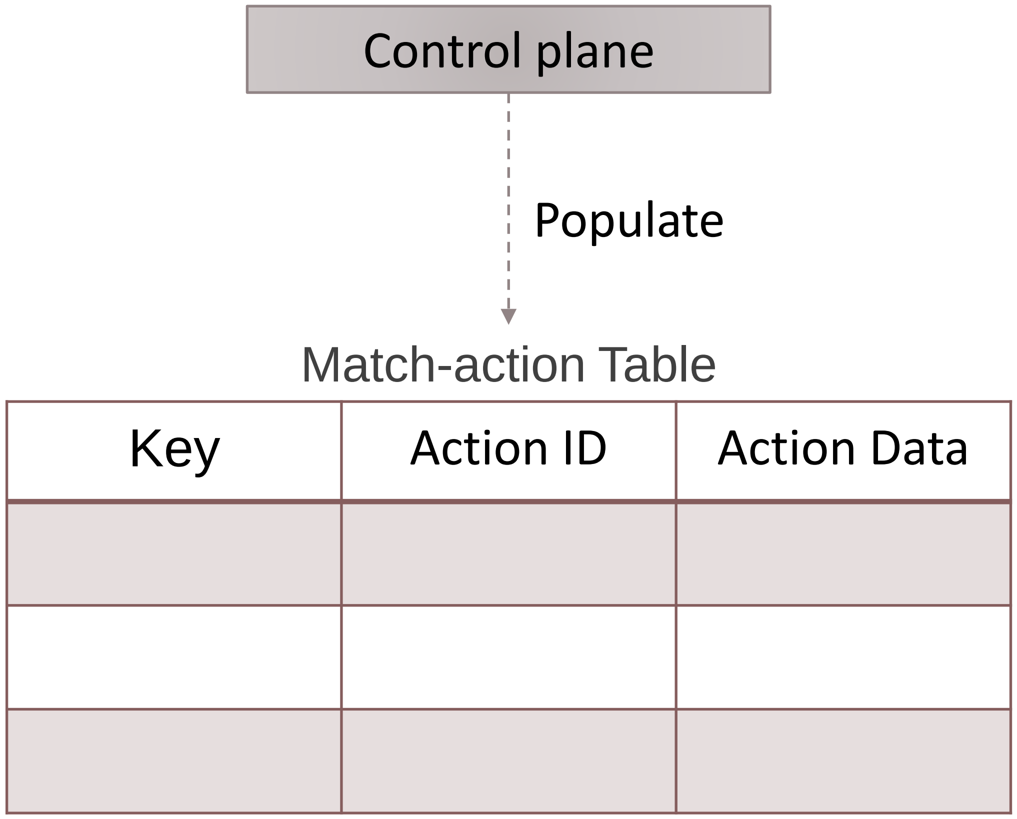
Tables are implemented here

Tables are implemented here

Match-action Table

- Specifies what data to match on
- Specifies a list of possible actions
- Optionally specifies a number of table properties; e.g.,
  - Size
  - Default action
  - Static entries
- An entry contains
  - A specific key to match on
  - An action that is executed when a packet matches the entry
  - Action data (possibly empty)

Match-action Table



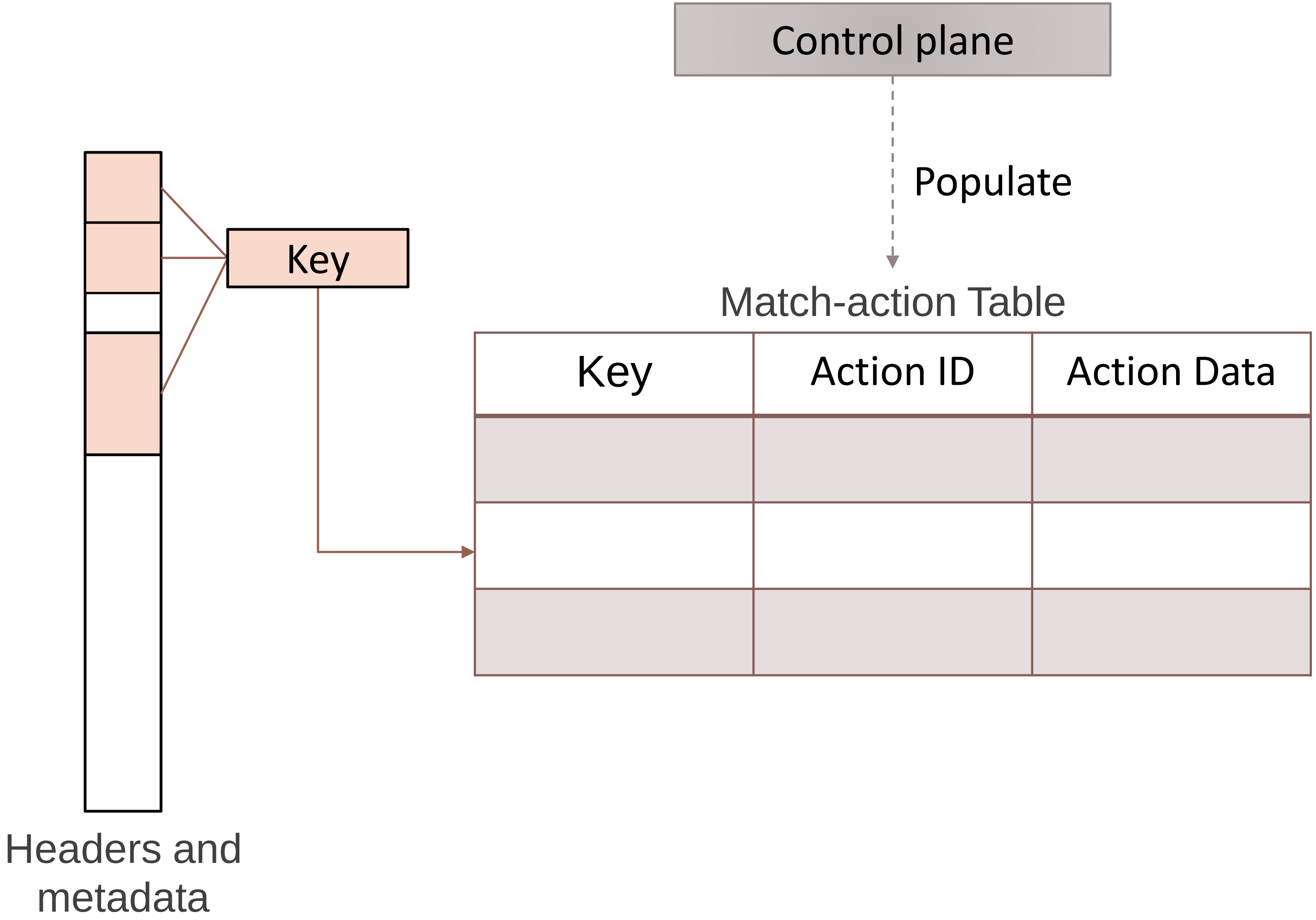
Control plane

Populate

Match-action Table

Key	Action ID	Action Data

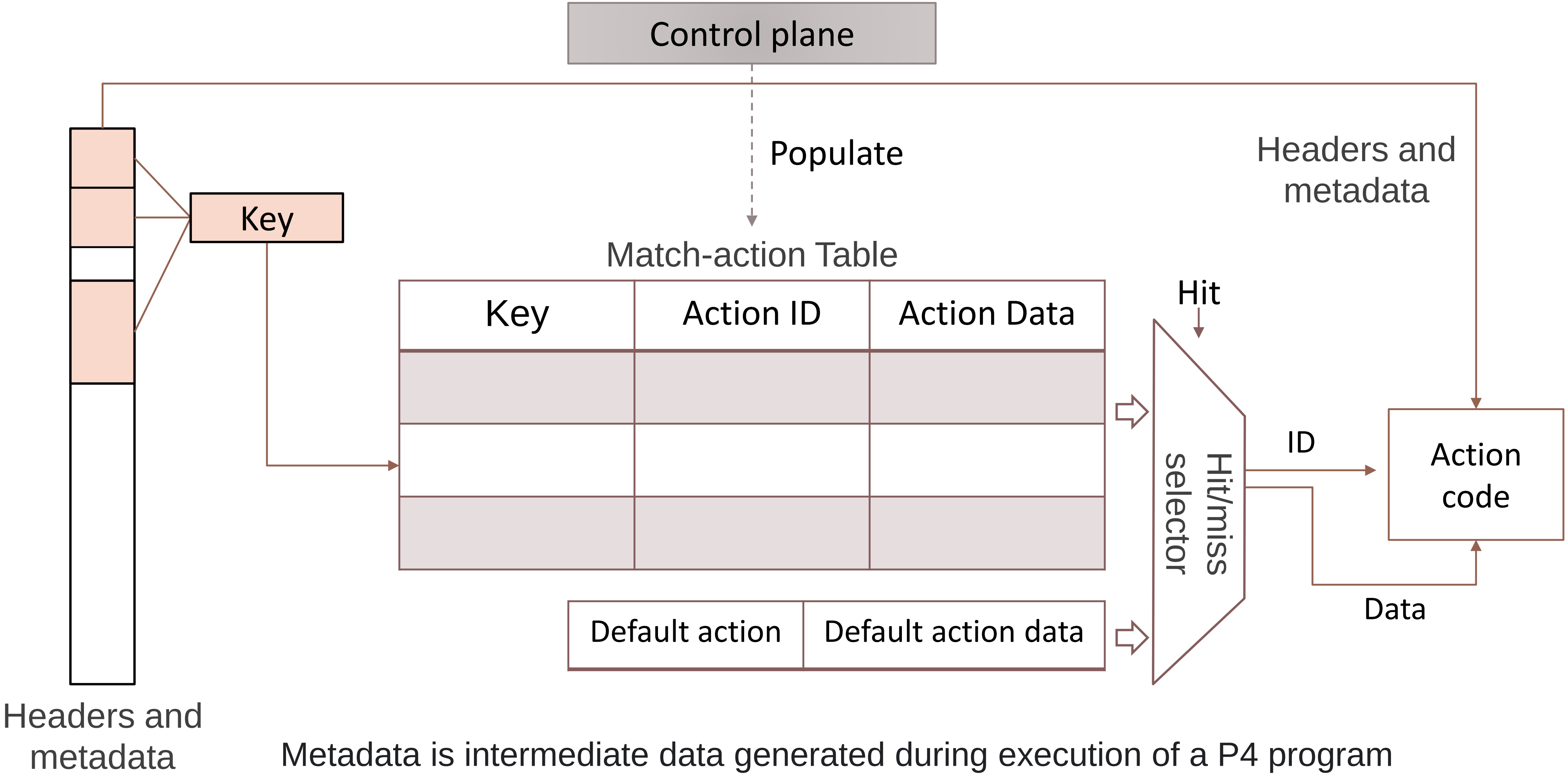
Match-action Table



Match-action Table

Key	Action ID	Action Data

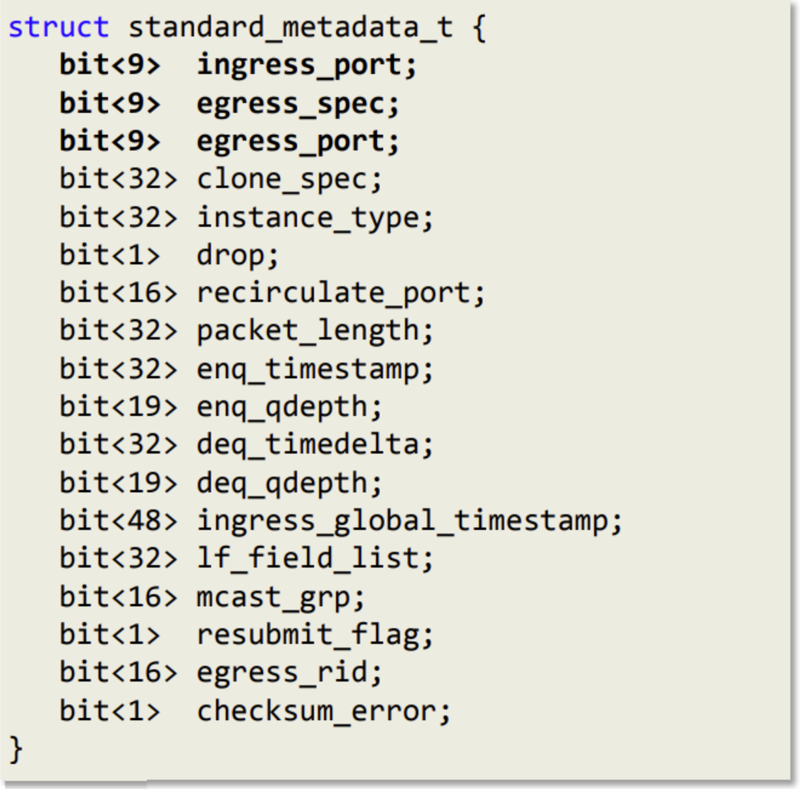
Match-action Table



Metadata is intermediate data generated during execution of a P4 program

Match-action Table

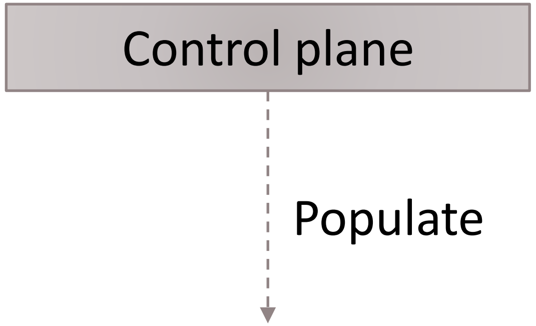
- Metadata is intermediate data generated during execution of a P4 program
- Standard metadata - data that must be provided by targets
  - `ingress_port`: port on which the packet arrived
  - `egress_spec`: port to which the packet should be sent to
  - `egress_port`: port on which the packet is departing from (read only in egress pipeline; useful value on ingress pipeline only)



```
struct standard_metadata_t {
    bit<9>   ingress_port;
    bit<9>   egress_spec;
    bit<9>   egress_port;
    bit<32> clone_spec;
    bit<32> instance_type;
    bit<1>   drop;
    bit<16> recirculate_port;
    bit<32> packet_length;
    bit<32> enq_timestamp;
    bit<19> enq_qdepth;
    bit<32> deq_timedelta;
    bit<19> deq_qdepth;
    bit<48> ingress_global_timestamp;
    bit<32> lf_field_list;
    bit<16> mcast_grp;
    bit<1>   resubmit_flag;
    bit<16> egress_rid;
    bit<1>   checksum_error;
}
```

V1 model standard metadata

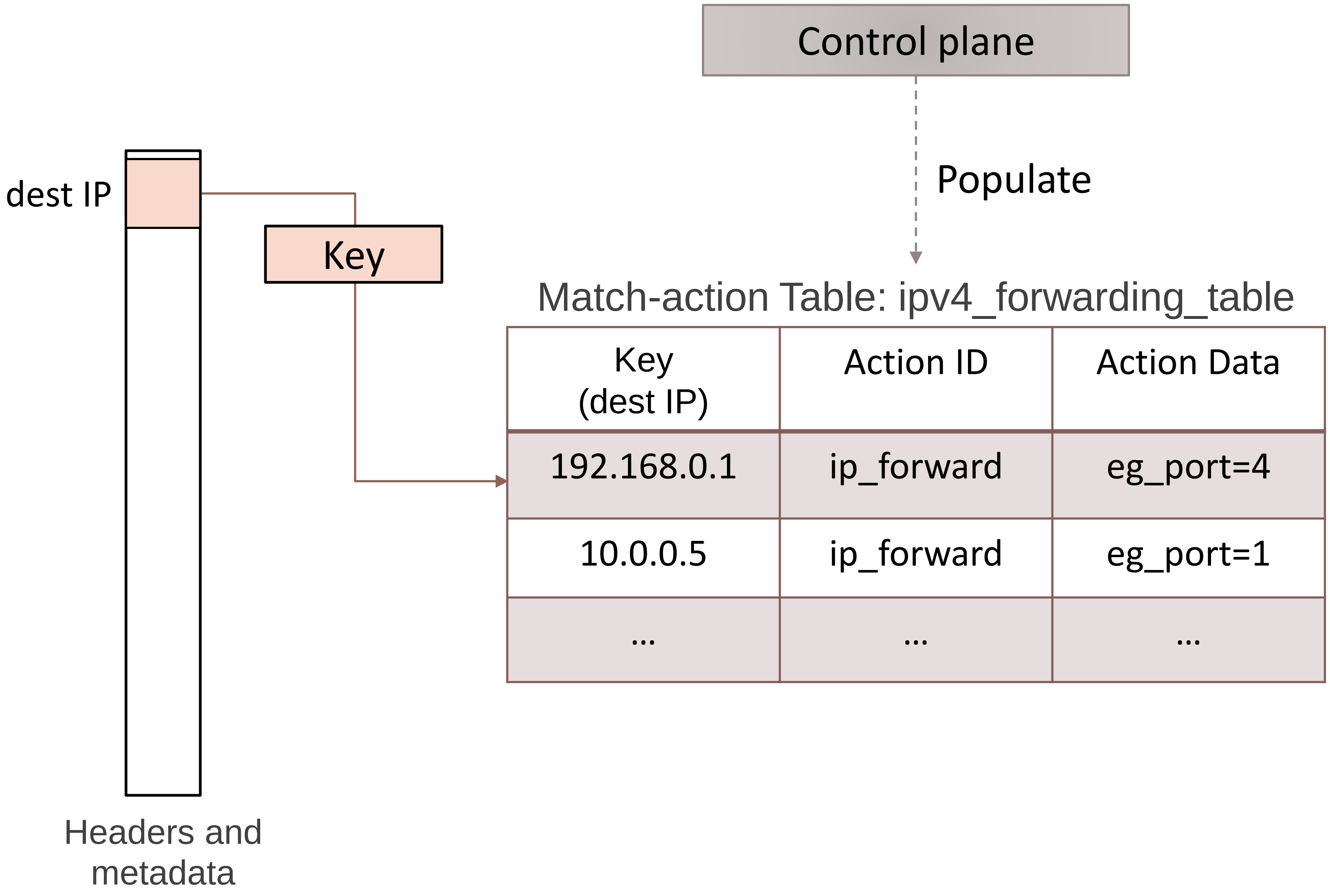
Example: IPv4 Forwarding



Match-action Table: ipv4_forwarding_table

Key (dest IP)	Action ID	Action Data
192.168.0.1	ip_forward	eg_port=4
10.0.0.5	ip_forward	eg_port=1
…	…	…

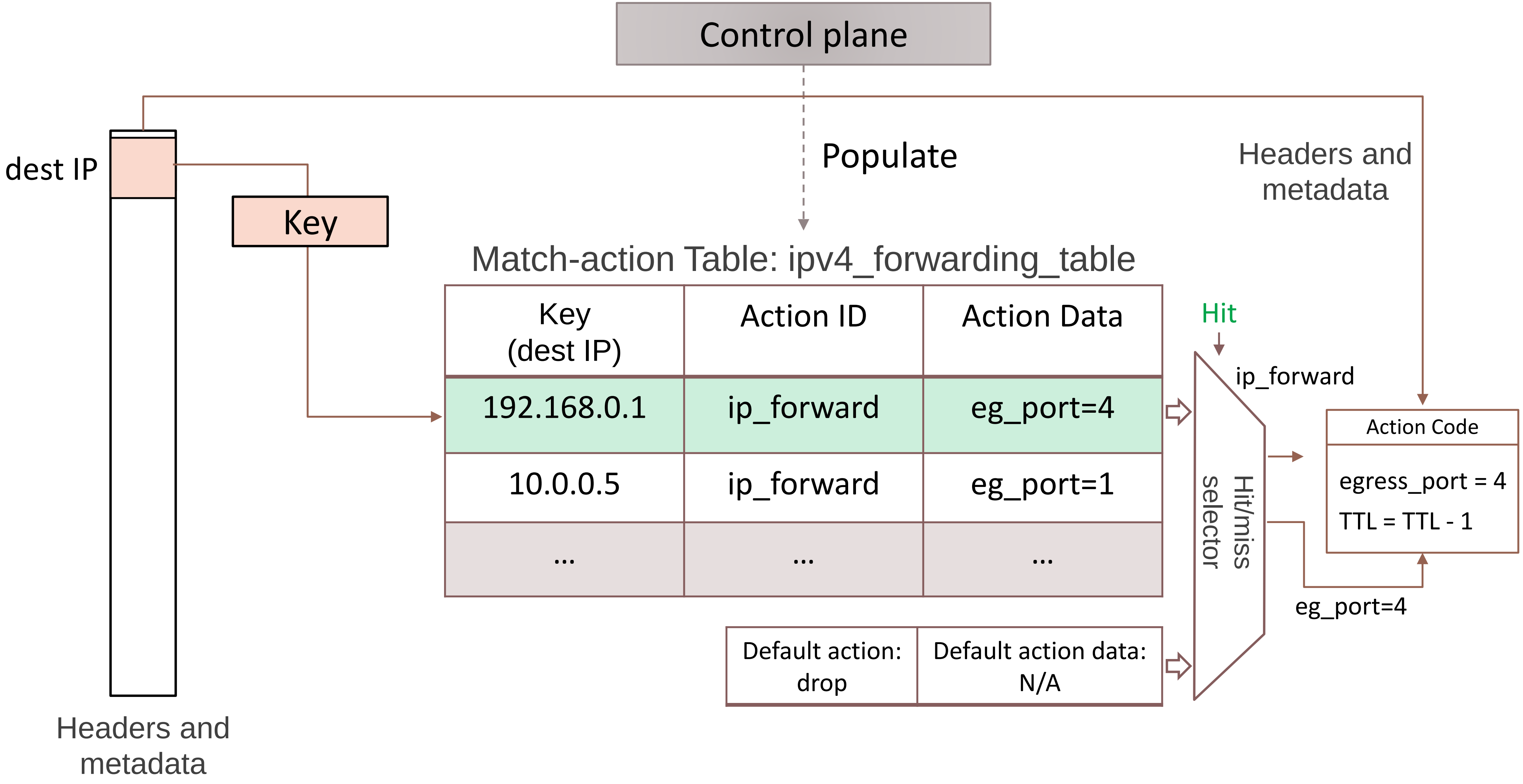
Example: IPv4 Forwarding



Match-action Table: ipv4_forwarding_table

Key (dest IP)	Action ID	Action Data
192.168.0.1	ip_forward	eg_port=4
10.0.0.5	ip_forward	eg_port=1
…	…	…

Example: IPv4 Forwarding



Control plane

Populate

dest IP

Key

Headers and metadata

Match-action Table: ipv4_forwarding_table

Key (dest IP)	Action ID	Action Data
192.168.0.1	ip_forward	eg_port=4
10.0.0.5	ip_forward	eg_port=1
…	…	…

Default action: drop	Default action data: N/A

Hit

ip_forward

Hit/miss selector

eg_port=4

Action Code

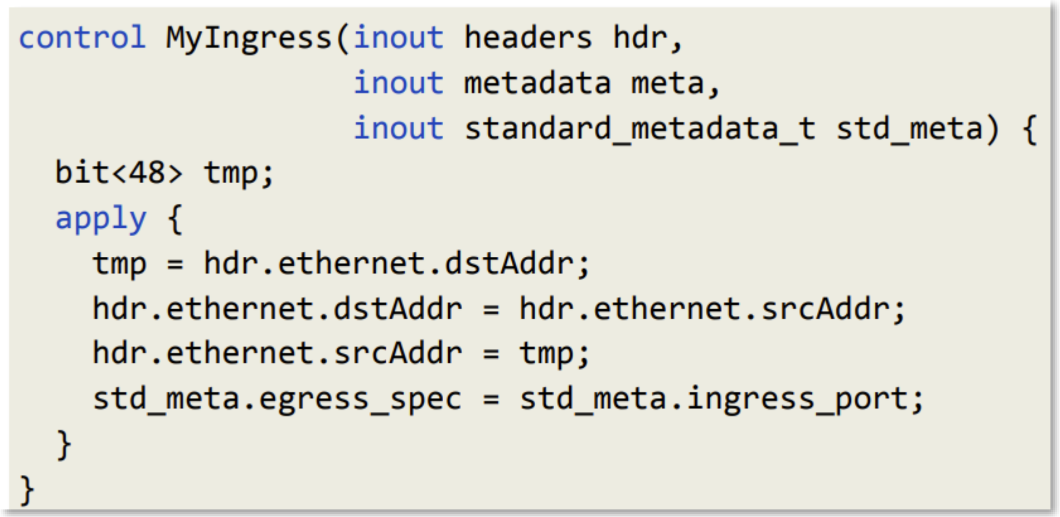
egress_port = 4

TTL = TTL - 1

Headers and metadata

Controls

- Similar to C functions (without loops)
- Can declare tables, variables
- Functionality specified by code in `apply` statement



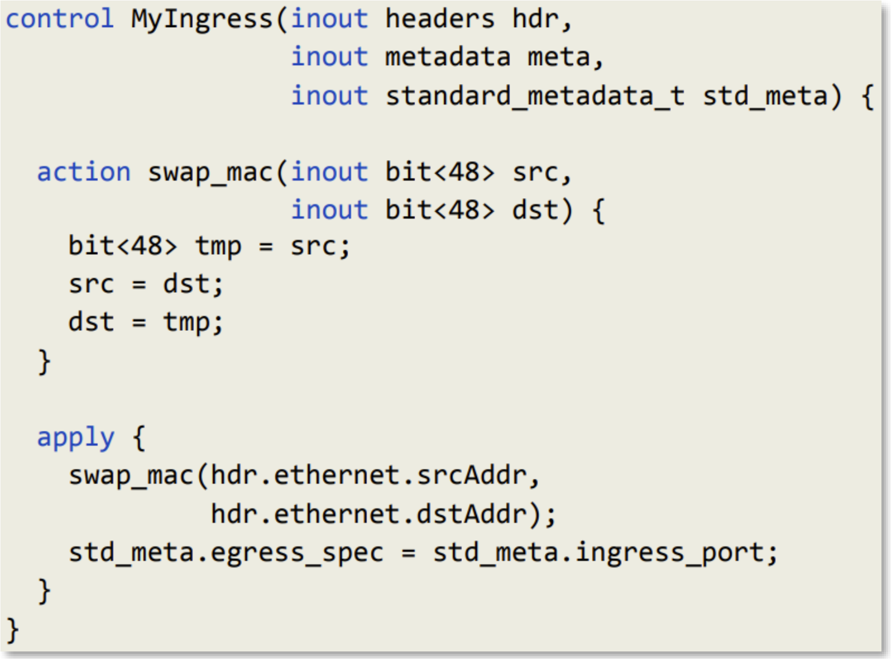
```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {
  bit<48> tmp;
  apply {
    tmp = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
    hdr.ethernet.srcAddr = tmp;
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

Swap source and destination MAC addresses

Bounce the packet back out on the physical port that it came into the switch on

Actions

- Similar to C functions
- Can be declared inside a control or globally
- Parameters have type and direction



```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {

  action swap_mac(inout bit<48> src,
                  inout bit<48> dst) {
    bit<48> tmp = src;
    src = dst;
    dst = tmp;
  }

  apply {
    swap_mac(hdr.ethernet.srcAddr,
             hdr.ethernet.dstAddr);
    std_meta.egress_spec = std_meta.ingress_port;
  }
}
```

Swap source and destination
MAC addresses

Bounce the packet back out on
the physical port that it came
into the switch on