Cybersecurity (Security+) and P4 Programmable Switches

Lab 7: Implementing a Stateful Packet Filter for the TCP protocol

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

Western Academy Support and Training Center (WASTC)
University of South Carolina (USC)
Energy Sciences Network (ESnet)

June 22nd, 2023

Outline

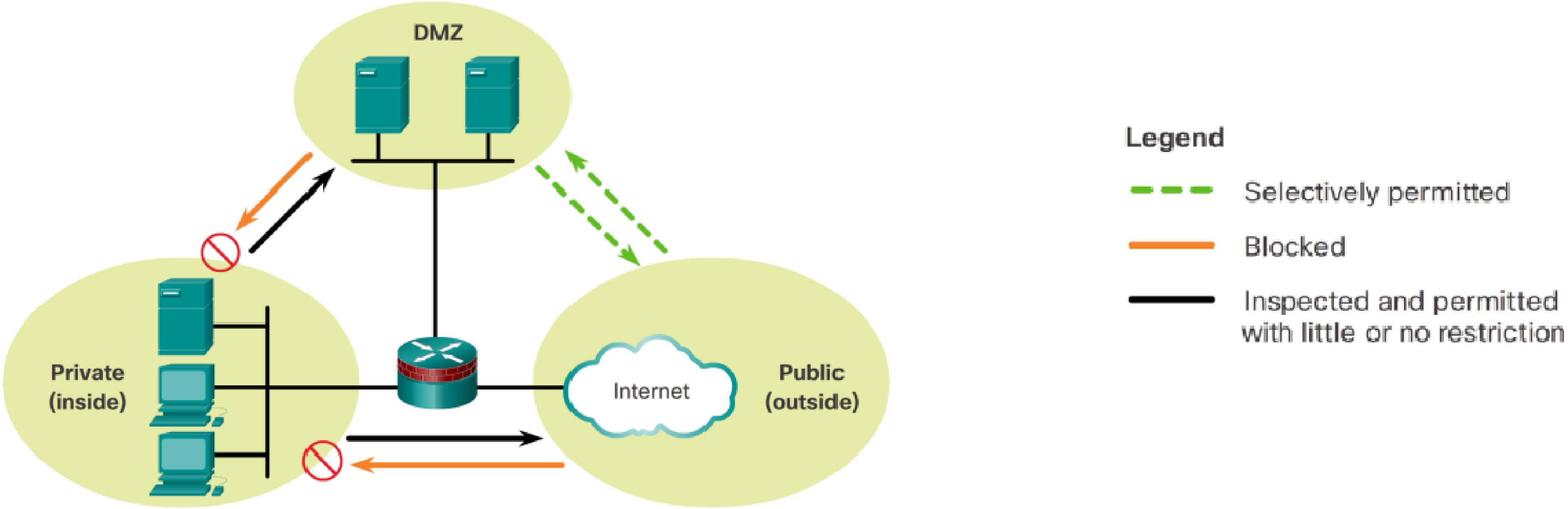
- Packet filters
- Implementing a stateful packet filter in P4
- P4 registers
- Topology
- Lab objectives

Packet Filters

- Packet filters control and manage the flow of data across a network
- They filter and analyze outgoing and incoming packets
- Packet filters can be **stateless** or **stateful**
- Stateless packet filters operate on a per-packet basis
- Stateful packet filters maintain state to track the status of ongoing connections

Packet Filters

- Stateful packet filter enable returning traffic to be allowed
- Traffic originating from private network is permitted; returning traffic from public network is permitted
- Traffic originating from public network to private network is denied

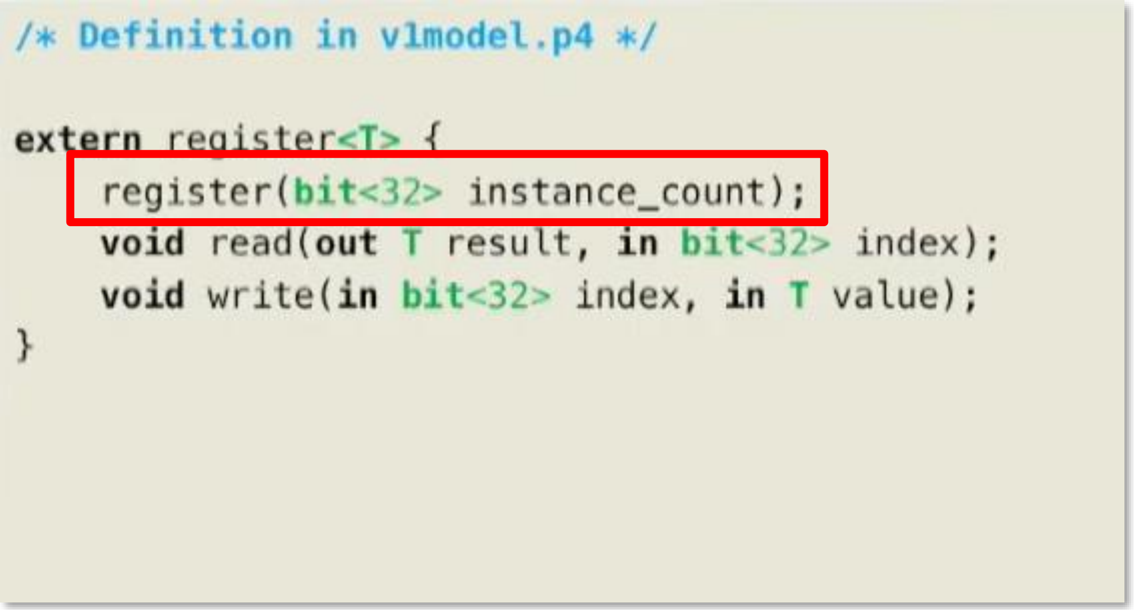


Implementing a Packet Filter in P4

- Implementing a stateful packet filter require maintaining state in the P4 switch
- **Registers** are stateful memories whose values can be read and written in actions in the data plane
- They can also be read and written by the control plane
- They are more general than counters; arbitrary data can be stored in registers

Registers in V1 Model

- The definition of the V1 Model register includes
  - `register` instantiation that receives an input parameter –number of elements of the register

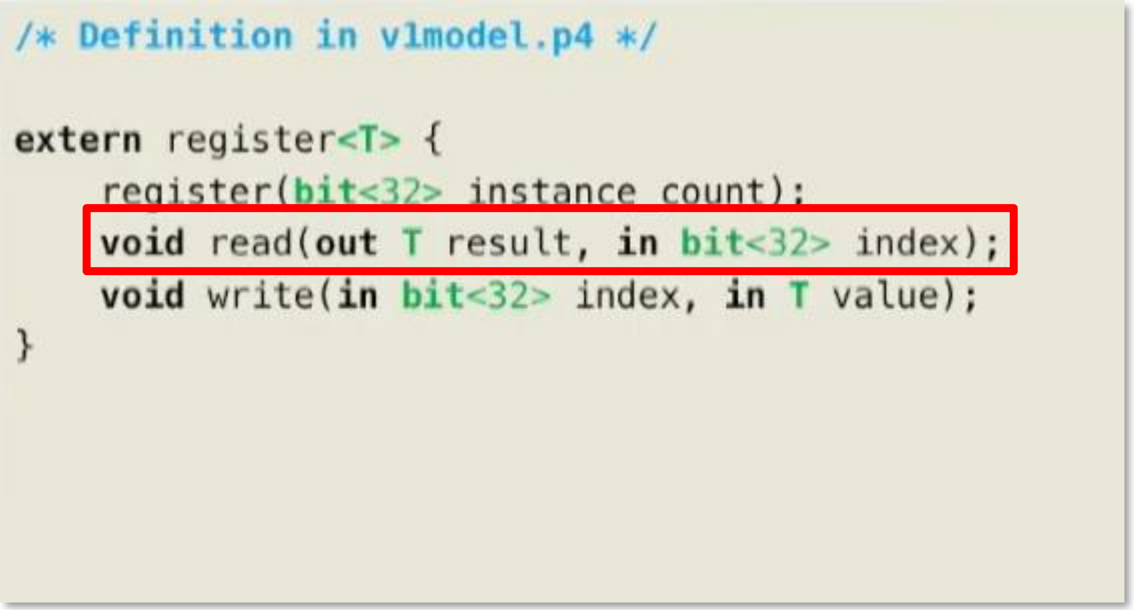


```
/* Definition in v1model.p4 */

extern register<T> {
    register(bit<32> instance_count);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

1. V. Gurevich, Introduction to P416. Online: https://tinyurl.com/2h93pnyd

Registers in V1 Model

- The definition of the V1 Model register includes
  - `register` instantiation that receives an input parameter –number of elements of the register
  - `read` method that receives an output parameter –where to store the register value– and an input parameter –index

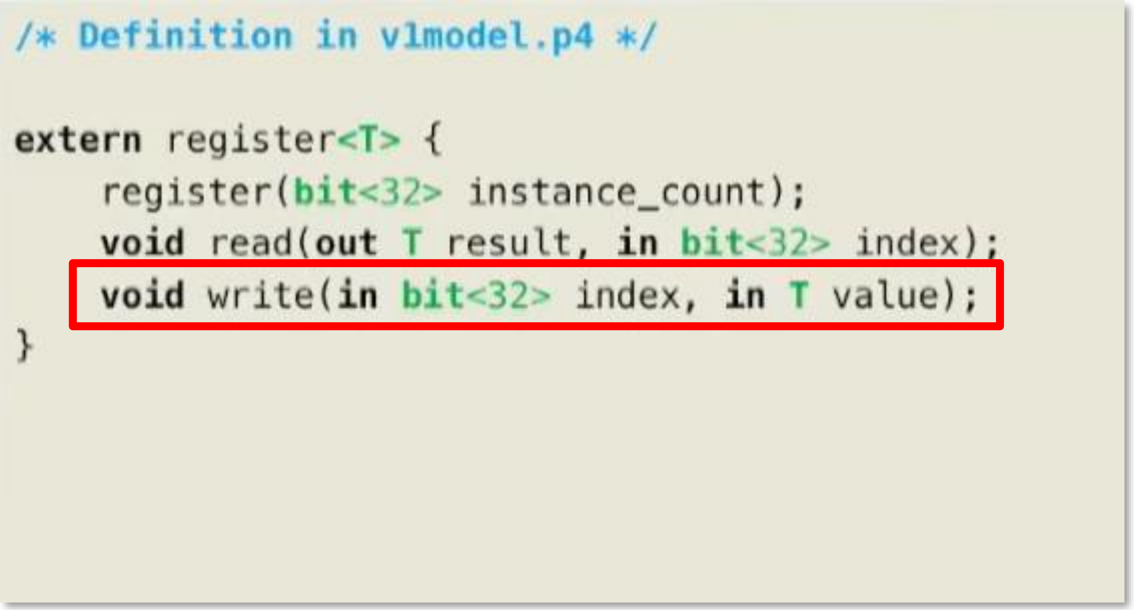


```
/* Definition in v1model.p4 */

extern register<T> {
    register(bit<32> instance_count);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

1. V. Gurevich, Introduction to P416. Online: https://tinyurl.com/2h93pnyd

Registers in V1 Model

- The definition of the V1 Model register includes
  - `register` instantiation that receives an input parameter –number of elements of the register
  - `read` method that receives an output parameter –where to store the register value– and an input parameter –index
  - `write` method that receives two input parameters, index and value to store in the register



```
/* Definition in v1model.p4 */

extern register<T> {
    register(bit<32> instance_count);
    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

1. V. Gurevich, Introduction to P416. Online: https://tinyurl.com/2h93pnyd

Instantiating a Single Element Register

- The syntax below shows how to instantiate a single element register in P4

```
register<bit<N>>(1) R1;
```

- Register R1 contains a single N-bit element

Register R1

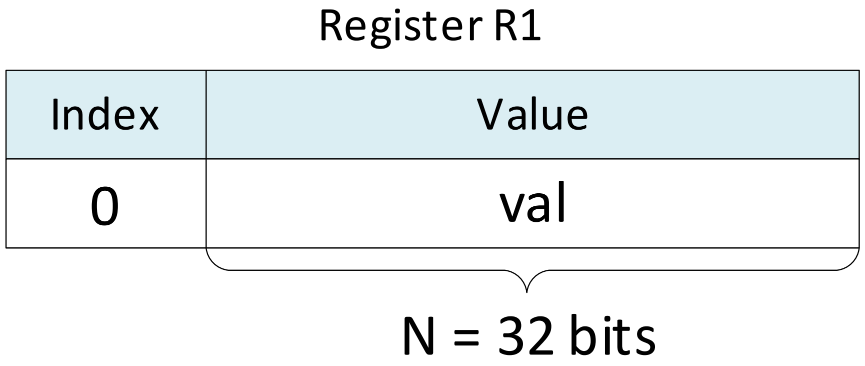
Index	Value
0	

N = 32 bits

Writing a Single Element Register

- The syntax below shows how to write (store) a value *val* in register R1 element 0

```
R1.write(0,val)
```



Register R1

Index	Value
0	val

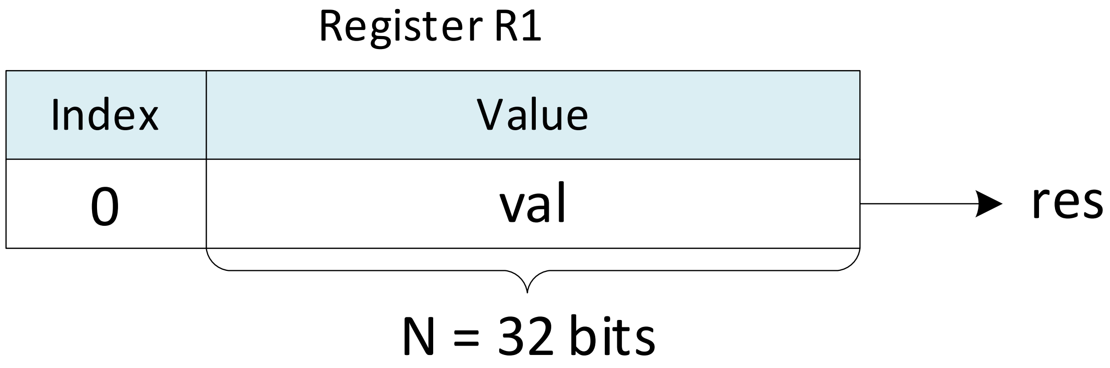
N = 32 bits

Reading a Single Element Register

- The syntax below shows how to read the value stored in element 0 of the register, and store it into the variable `res`

```
R1.read(res,0)
```

- Note that the value `val` is stored in the variable `res`



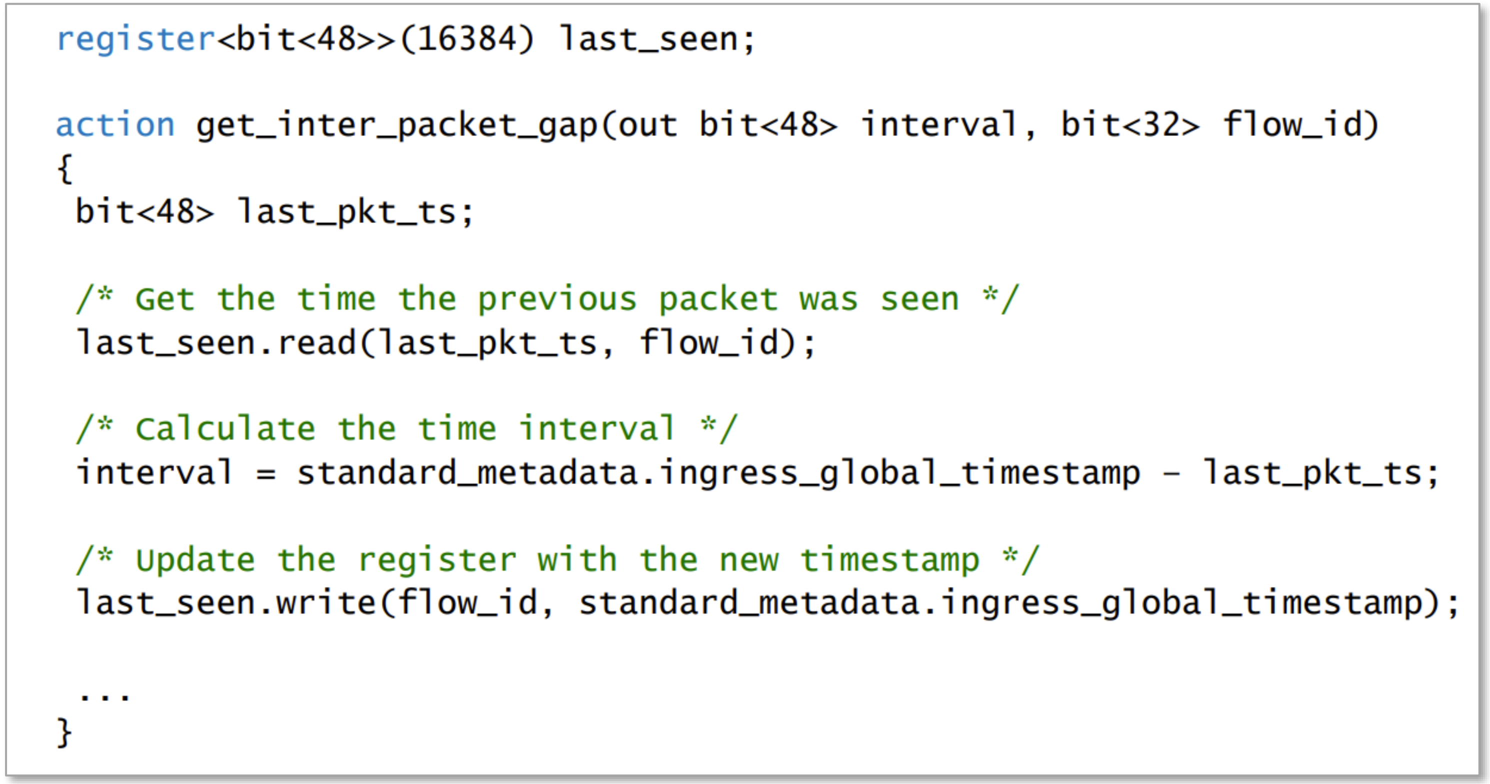
Register R1

Index	Value
0	val

Registers in V1 Model

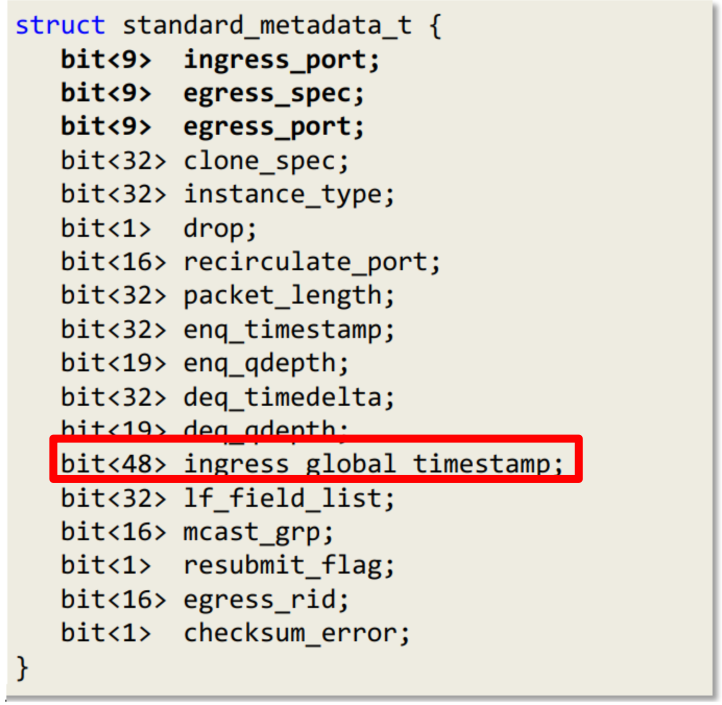
- Example: computing the time between two consecutive packets of a flow (inter-packet gap)

Code



```
register<bit<48>>(16384) last_seen;

action get_inter_packet_gap(out bit<48> interval, bit<32> flow_id)
{
 bit<48> last_pkt_ts;

 /* Get the time the previous packet was seen */
 last_seen.read(last_pkt_ts, flow_id);

 /* Calculate the time interval */
 interval = standard_metadata.ingress_global_timestamp – last_pkt_ts;

 /* Update the register with the new timestamp */
 last_seen.write(flow_id, standard_metadata.ingress_global_timestamp);

  ...
}
```

Standard metadata

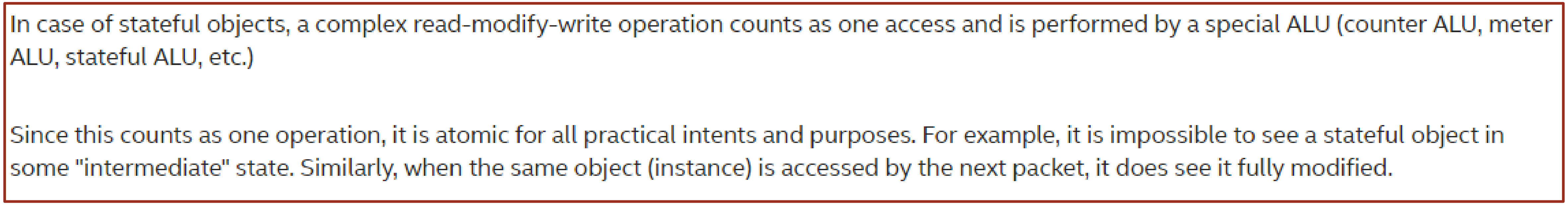


```
struct standard_metadata_t {
   bit<9>  ingress_port;
   bit<9>  egress_spec;
   bit<9>  egress_port;
   bit<32> clone_spec;
   bit<32> instance_type;
   bit<1>  drop;
   bit<16> recirculate_port;
   bit<32> packet_length;
   bit<32> enq_timestamp;
   bit<19> enq_qdepth;
   bit<32> deq_timedelta;
   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;
}
```

1. L. Vanbever, Programming Network Data Planes. Online: https://tinyurl.com/ywr3c6rb

Atomicity

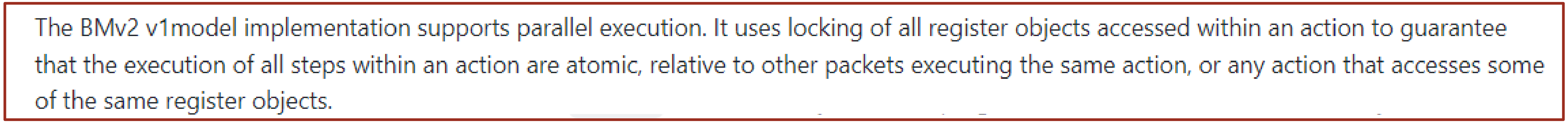
- Hardware and software targets use atomic operations on P4 stateful objects
- For Intel Tofino switch (Vladimir Gurevich)[1]:



In case of stateful objects, a complex read-modify-write operation counts as one access and is performed by a special ALU (counter ALU, meter ALU, stateful ALU, etc.)

Since this counts as one operation, it is atomic for all practical intents and purposes. For example, it is impossible to see a stateful object in some "intermediate" state. Similarly, when the same object (instance) is accessed by the next packet, it does see it fully modified.

- For BMv2 v1model implementation[2]:



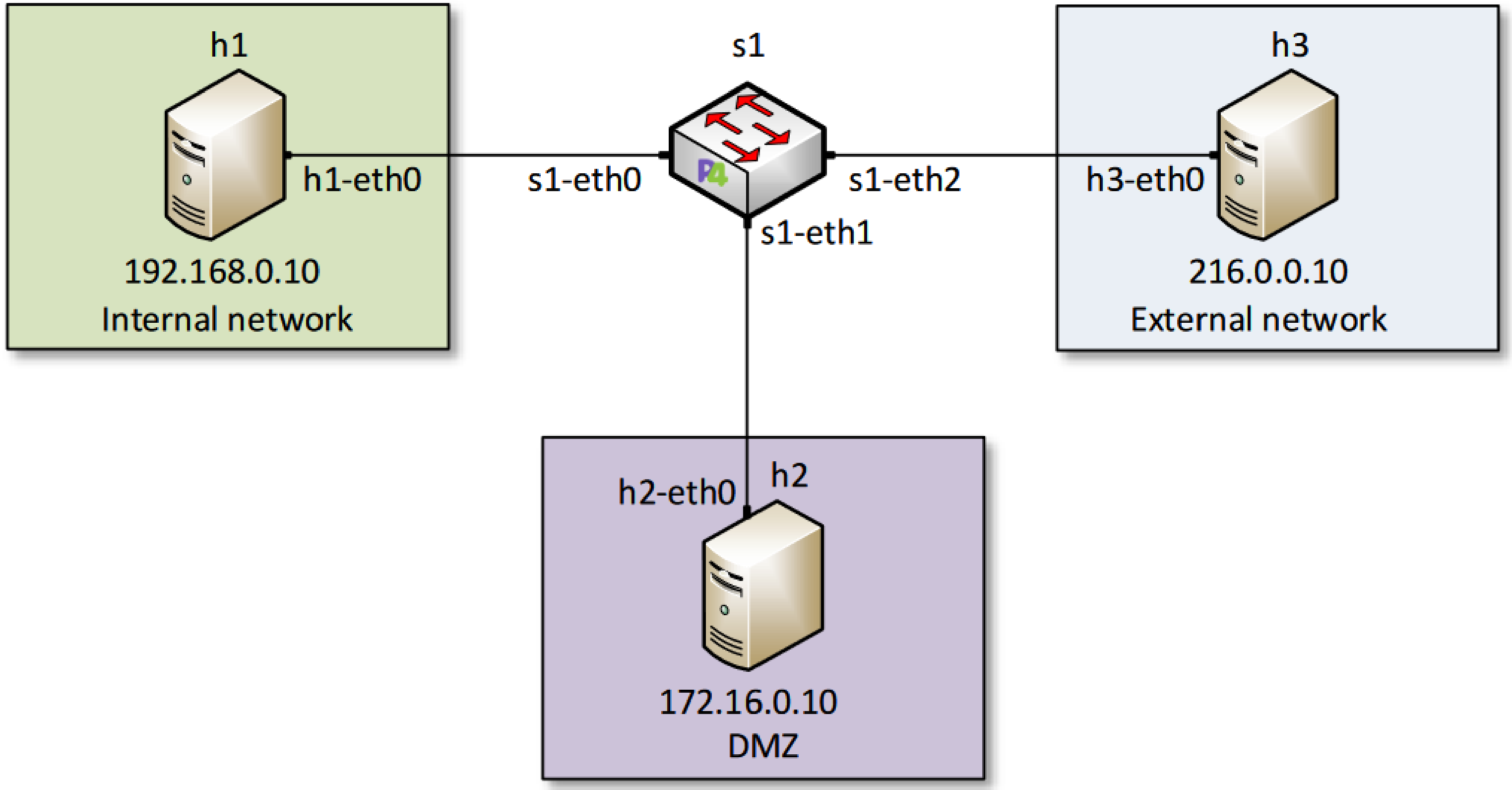
The BMv2 v1model implementation supports parallel execution. It uses locking of all register objects accessed within an action to guarantee that the execution of all steps within an action are atomic, relative to other packets executing the same action, or any action that accesses some of the same register objects.

1. Intel® Connectivity Research Program (Private). Memory semantics of Tofino architecture. Online: https://tinyurl.com/yz7hzydr
2. P4Lang Consortium, The BMv2 Simple Switch target. Online: https://tinyurl.com/26b762m3

Lab 7: Implementing a Stateful Packet Filter for the TCP protocol

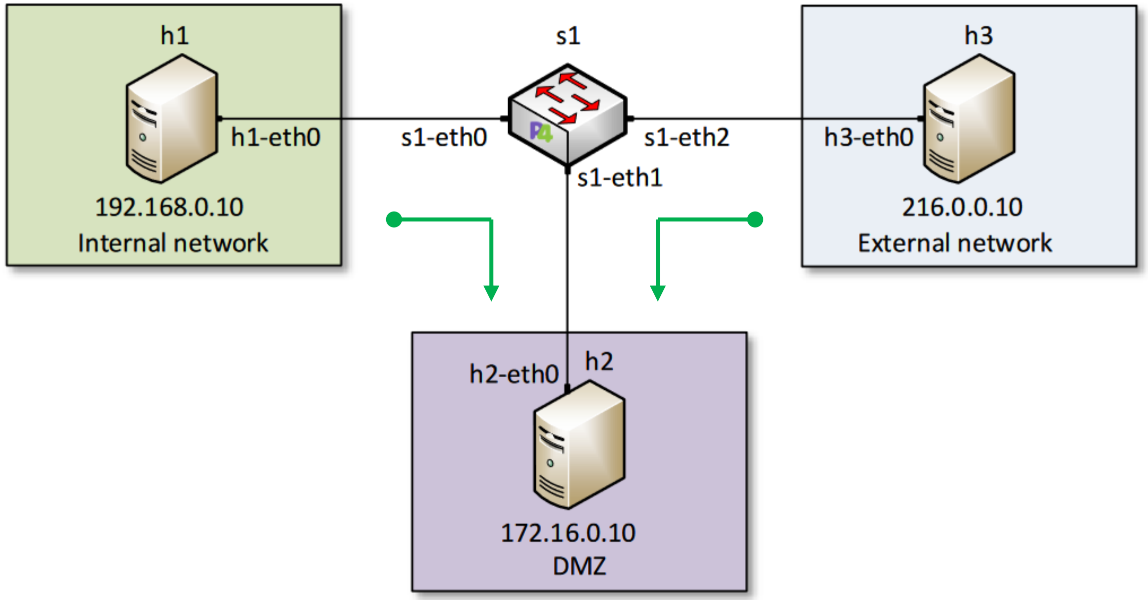
Topology

- The topology consists of:
  - Internal network: host h1
  - External network: host h3
  - DMZ network: host h2
  - P4 switch (packet filter)

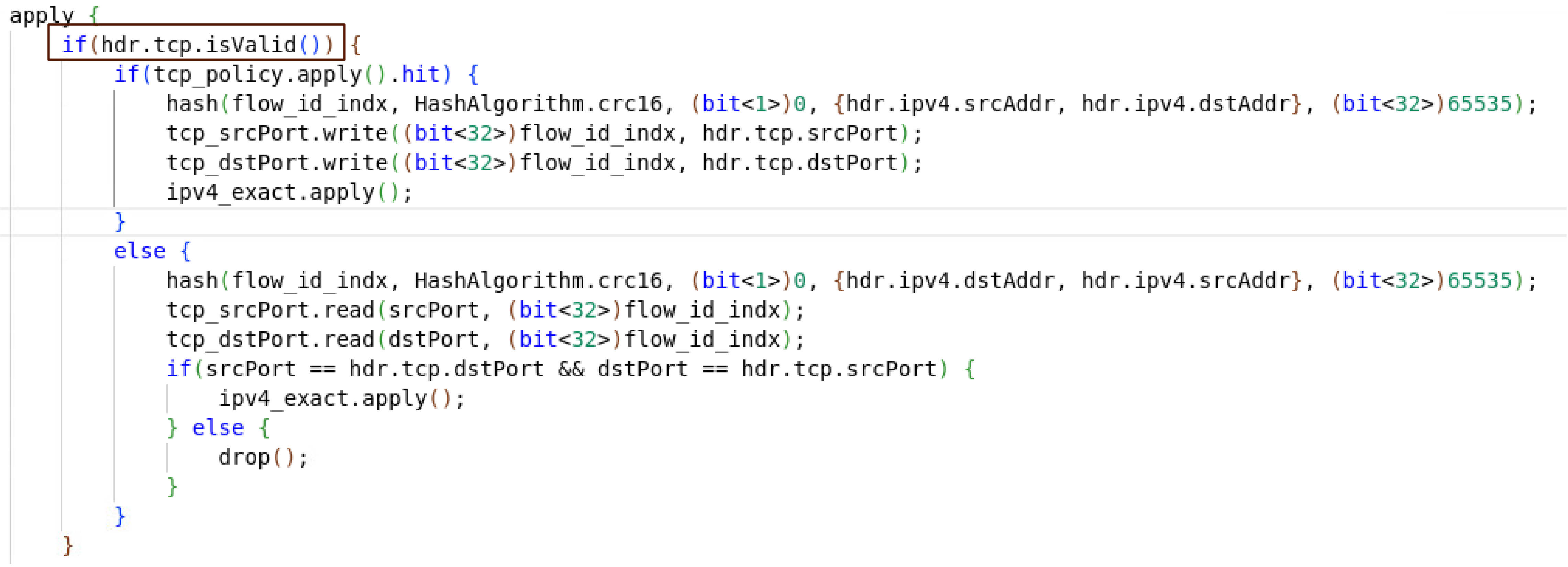


Lab Objectives

- Write a P4 program to implement a stateful packet filter for TCP
- TCP traffic originating from the internal or the external network towards the DMZ (172.16.0.10) is allowed
- All other traffic is blocked

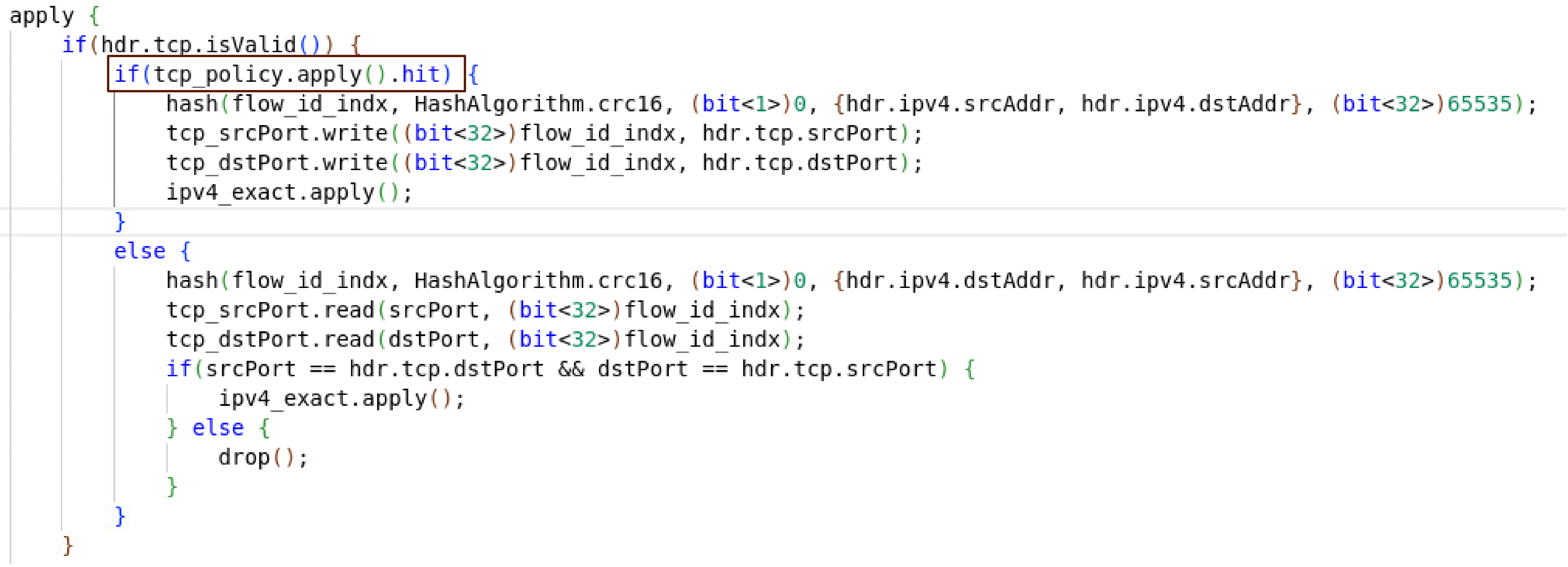


Lab Objectives



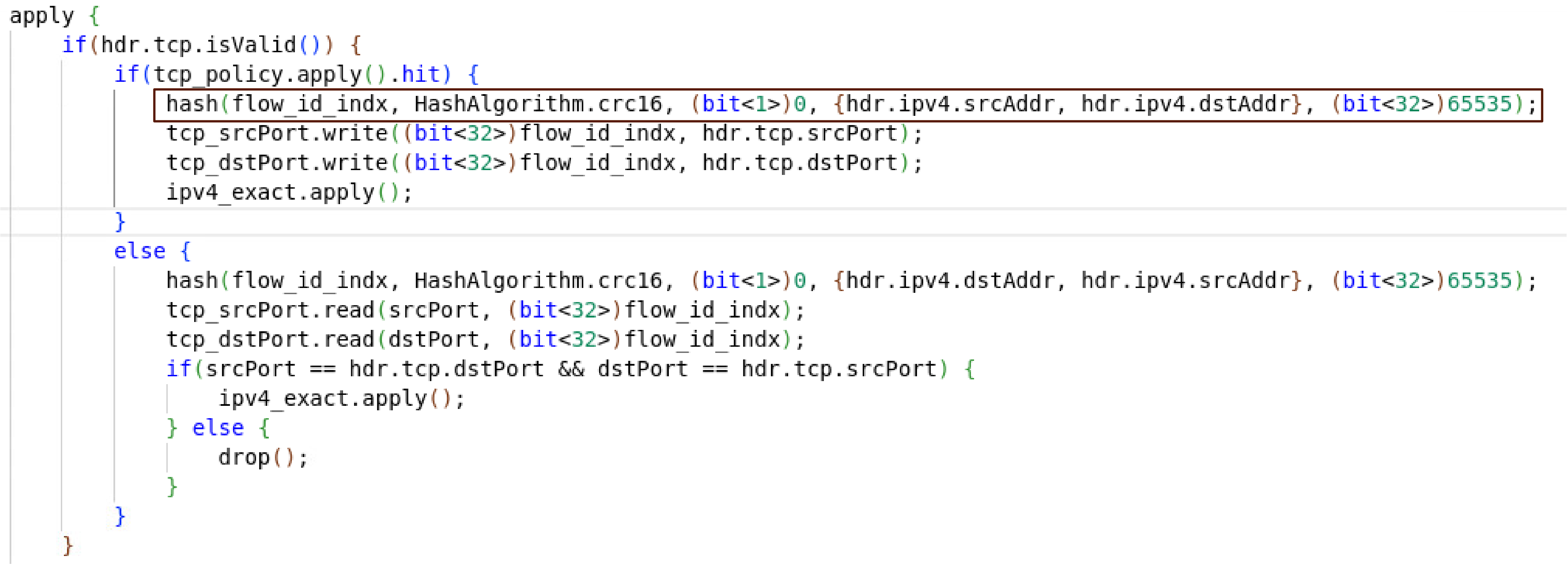
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



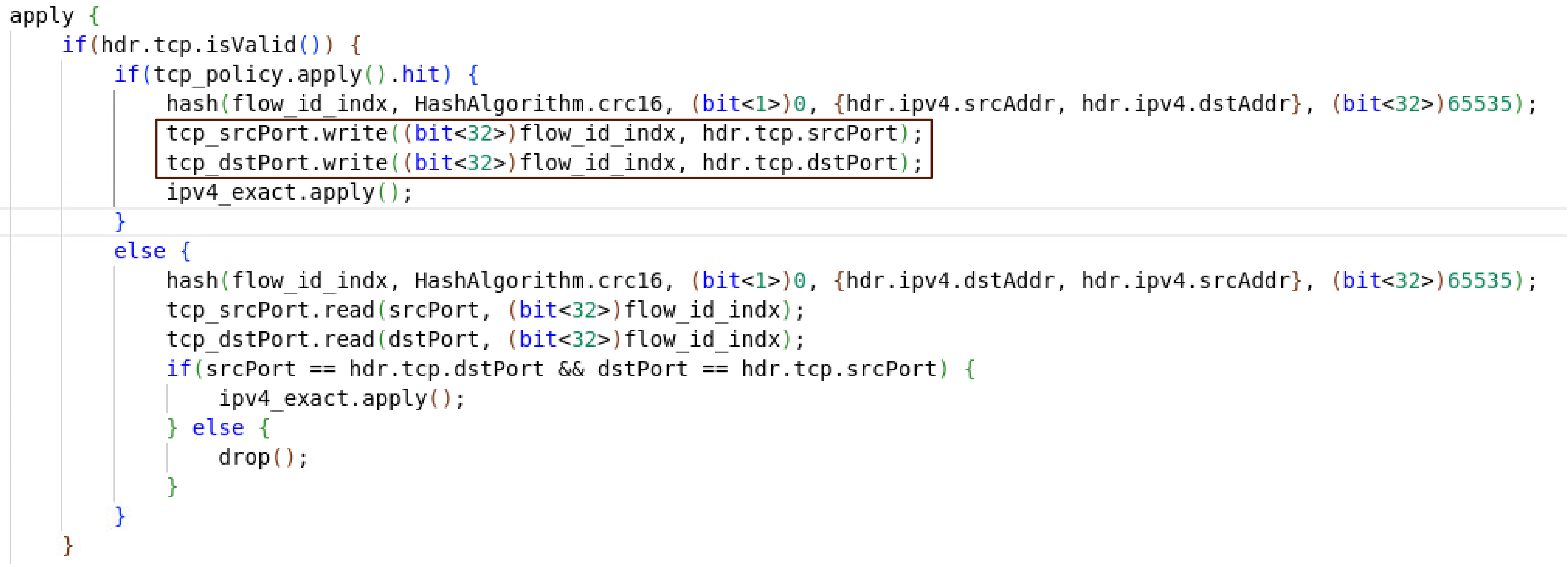
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



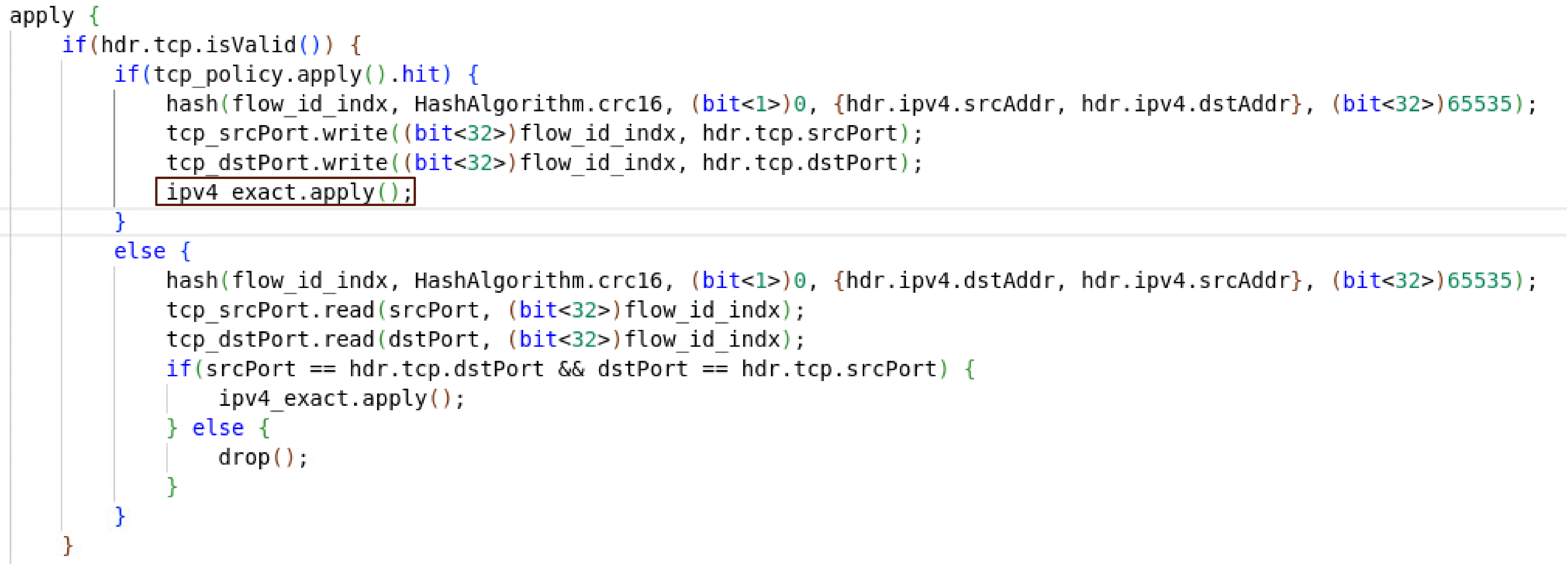
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



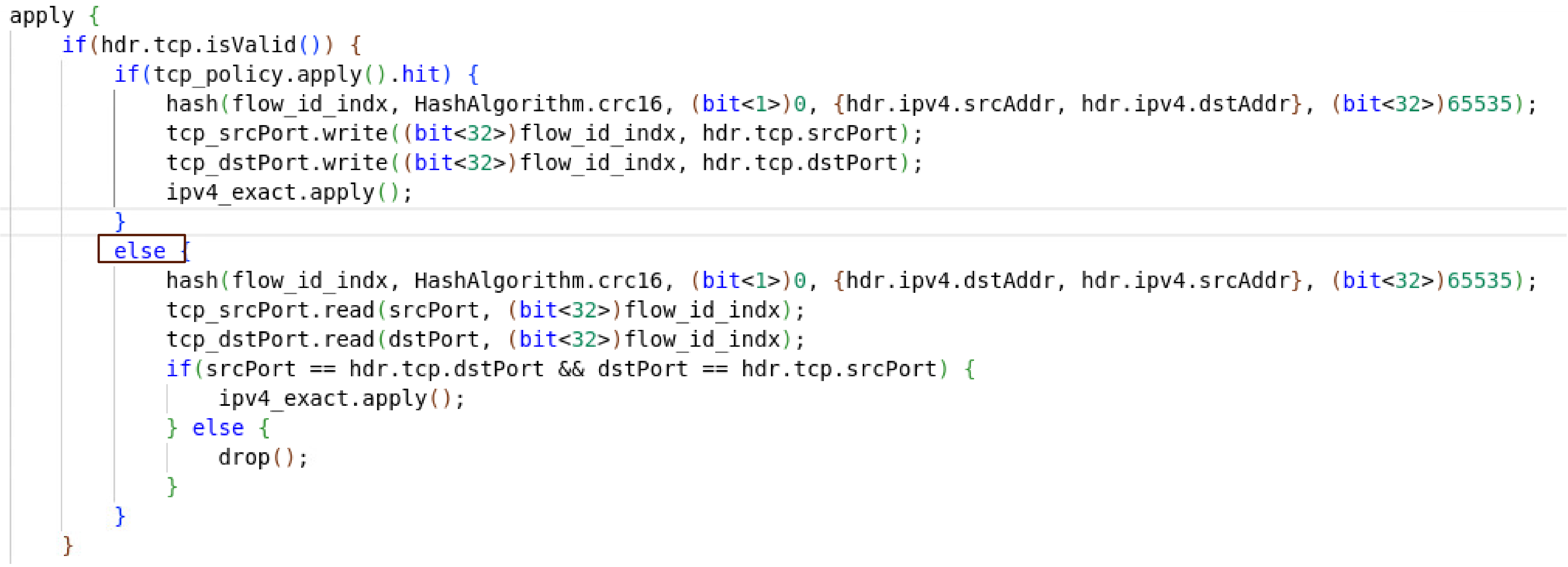
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



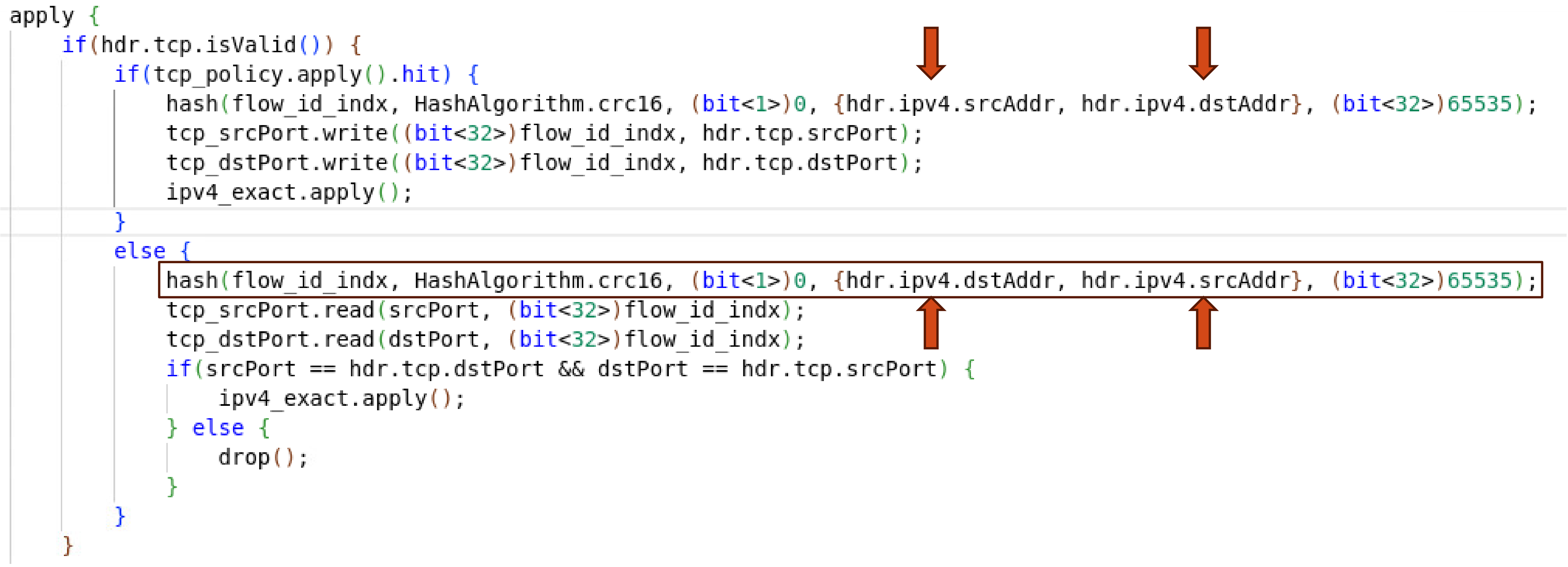
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



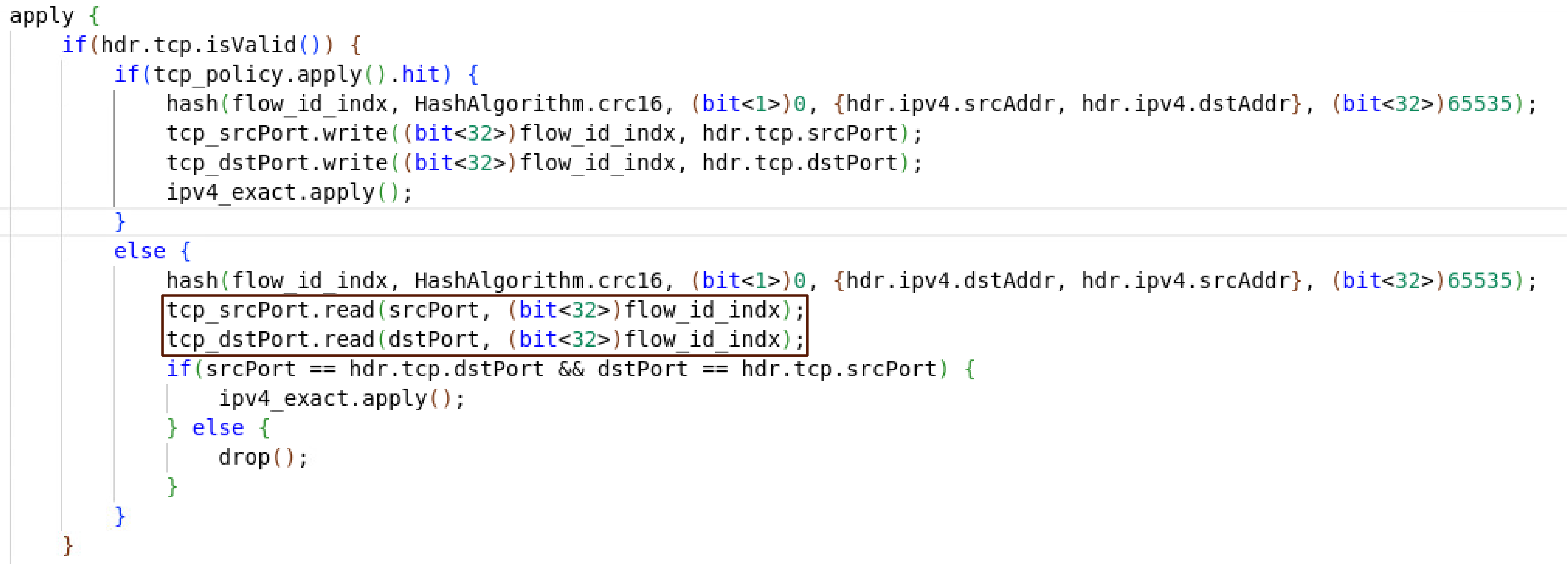
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



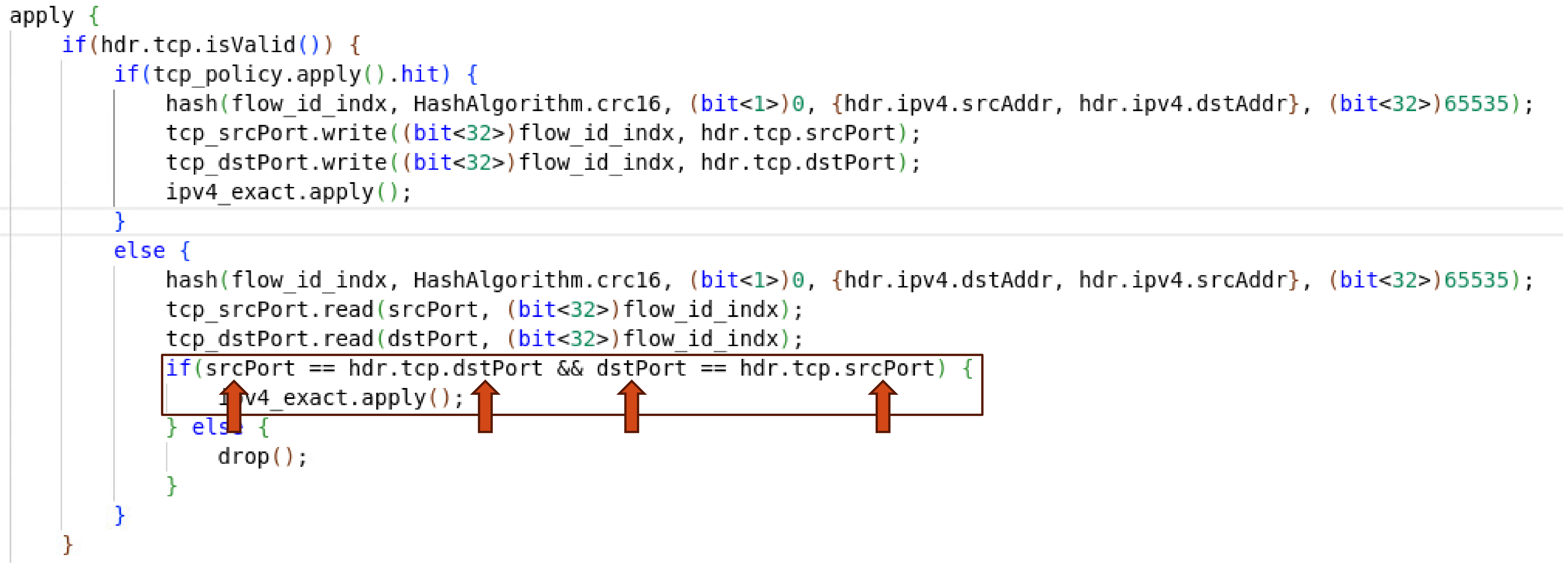
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



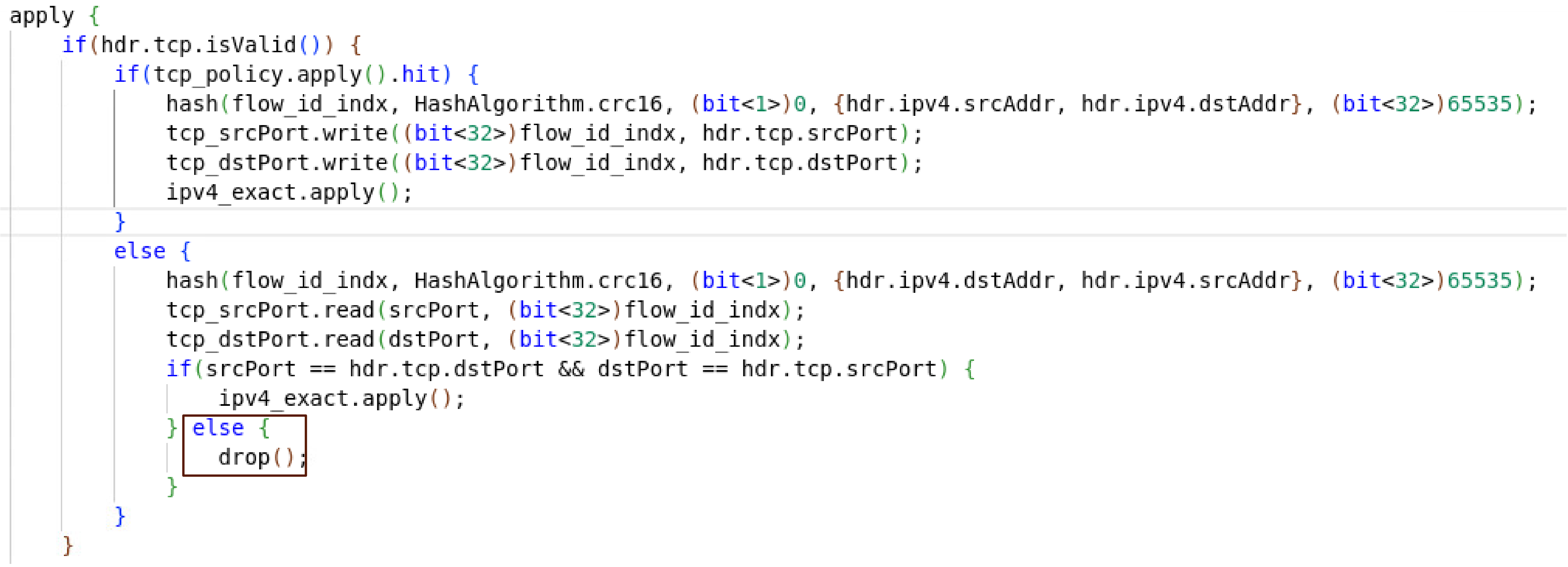
```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```

Lab Objectives



```
apply {
    if(hdr.tcp.isValid()) {
        if(tcp_policy.apply().hit) {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr}, (bit<32>)65535);
            tcp_srcPort.write((bit<32>)flow_id_indx, hdr.tcp.srcPort);
            tcp_dstPort.write((bit<32>)flow_id_indx, hdr.tcp.dstPort);
            ipv4_exact.apply();
        }
        else {
            hash(flow_id_indx, HashAlgorithm.crc16, (bit<1>)0, {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr}, (bit<32>)65535);
            tcp_srcPort.read(srcPort, (bit<32>)flow_id_indx);
            tcp_dstPort.read(dstPort, (bit<32>)flow_id_indx);
            if(srcPort == hdr.tcp.dstPort && dstPort == hdr.tcp.srcPort) {
                ipv4_exact.apply();
            } else {
                drop();
            }
        }
    }
```