Implementing a Stateful Packet Filter Using a P4 Programmable Switch

Andrew Smith & Nolan Pelino
Advisor: Ali AlSabeh

Department of Integrated Information Technology (IIT)
University of South Carolina

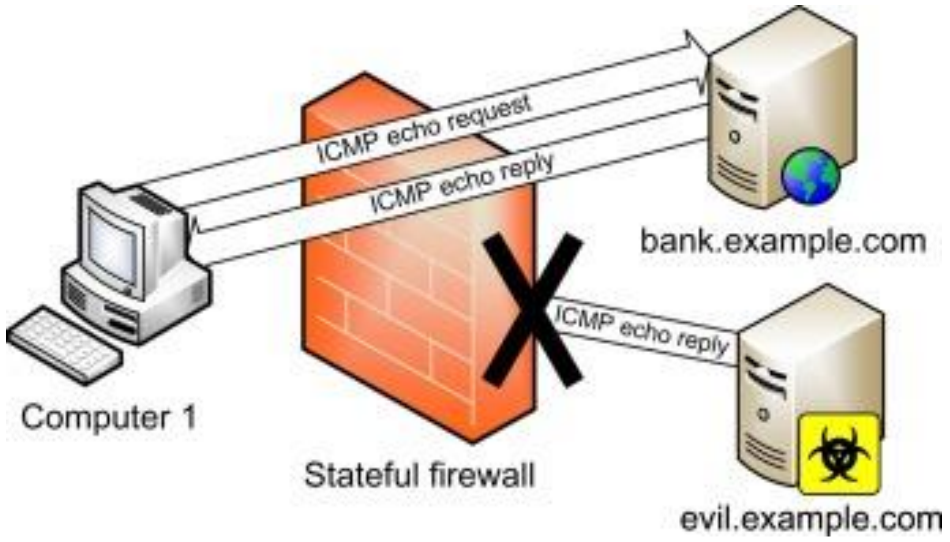
December 07, 2023

Agenda

- Introduction/Background
- Project Objective
- Implemented Solution
- Conclusion

Introduction

- The data plane is the structured pipeline that processes a stream of bits as they move through a switch
- P4 is a programming language that describes the behavior of the data plane, providing the programmer an unprecedented amount of control
- Stateful packet filters examine every packet passing through a switch, maintaining a log of various connections
- This data is then leveraged to enhance the precision of filtering decisions

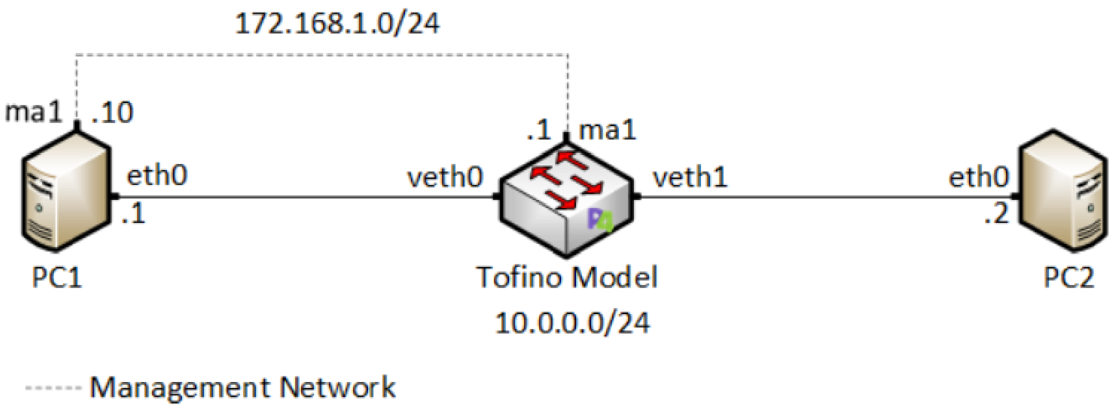


Project Objective

- The objective of this project is to implement a stateful packet filter on the Tofino P4-Programmable ethernet switches
- The packet filter is developed to interact with both TCP and ICMP
- The code is initially written on a BMv2 software switch and then migrated to be compatible with Tofino hardware switches
- The purpose of this project is to demonstrate P4's suitability in high-speed, stateful traffic filtering applications

Implemented Solution

- Topology
  - PC1 and PC2 are hosts on the 10.0.0.0/24 network
  - PC1 is used as the management interface for the Tofino switch
  - The goal is to allow connections initiated from PC1 to PC2 only, at the level of TCP and ICMP protocols



Implemented Solution

- Registers
  - Registers are used within P4 to save arbitrary data
  - Multiple packets can access or modify register data
  - We use them here to store and access the ICMP identifier field of packets

ICMP register

Index	Value
register_index1	ICMP Flow Identifier1
register_index2	ICMP Flow Identifier2
…	…
register_indexN	ICMP Flow IdentifierN

TCP register

Index	Value
register_index1	TCP Flow Identifier1
register_index2	TCP Flow Identifier2
…	…
register_indexM	TCP Flow IdentifierM

Implemented Solution

- Hashes
  - A hash function maps any given input values to an output of fixed length
  - In P4, we specify the hashing algorithm (CRC16) then initialize the variable that the hash will be stored in
  - Subsequently, the hash will be used to access the register entry corresponding to flow we are monitoring

```
Hash<bit<16>>(HashAlgorithm_t.CRC16) hash;
bit<16> sel_hash;
```

Line 1 specifies the algorithm we will use for the hash
Line 2 initializes the variable that the hash will be stored in

```
sel_hash = hash.get({hdr.ipv4.src_addr, hdr.ipv4.dst_addr});
```

The input of the hash function is the source and destination IP addresses of the packet. The output is a 16-bit hash value and will be stored in the variable sel_hash. This will later be used as the register index for the flow

Implemented Solution

```
write_data.execute(sel_hash);
```

```
sel_hash = hash.get({hdr.ipv4.src_addr, hdr.ipv4.dst_addr});
```

Index	Value
sel_hash	ICMP Flow Identifier
sel_hash	ICMP Flow Identifier
sel_hash	ICMP Flow Identifier

```
read_data.execute(sel_hash2);
```

Implemented Solution

- Match-action Tables
  - Match-action tables allow the programmer to define actions to be executed on a certain match (key), specified in the control plane
  - The *icmp_policy* and *tcp_policy* tables were designed to implement policies for the ICMP and TCP protocols
  - The key for the tables identify the flow ID
    - For example, in ICMP the flow ID is the source and destination IP addresses

```
table icmp_policy {
    key = {
        hdr.ipv4.src_addr: lpm;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {

    }
    size = 1024;
}
```

Implemented Solution

- Match-action Tables
  - There are no actions in the policy tables (NoAction), since no logic needs to be applied from the control plane
  - When a match occurs, we can interpret this as “true – the packet matches the policy, forward the packet” and on no match, as “false – the packet does not match the policy, drop the packet”
  - The size of the table is 1024, which is the number of entries in the table (i.e., number of policies we can implement)

```
table icmp_policy {
    key = {
        hdr.ipv4.src_addr: lpm;
        hdr.ipv4.dst_addr: ternary;
    }
    actions = {

    }
    size = 1024;
}
```

Implemented Solution

- Apply Logic
  - First check which of the ICMP or TCP headers are valid before processing

```
apply {
    //  ICMP
    if (hdr.icmp.isValid()) {
```

...

```
// TCP
else if (hdr.tcp.isValid()) {
```

  - If the ICMP packet is an echo request, store the ICMP ID in a register at the hashed index then forward

```
if (hdr.icmp.isValid()) {
    if (hdr.icmp.type == 8) {  // ICMP echo request
        if (icmp_policy.apply().hit) {

            sel_hash = hash.get({hdr.ipv4.src_addr, hdr.ipv4.dst_addr});  // Store hash value of source and destination address

            write_data.execute(sel_hash);  //Write To ICMP Register Based On Hashed Value (store identifier in header)

            ipv4_host.apply();
        }
```

Implemented Solution

- Apply Logic
  - If the ICMP packet is an echo reply, read the value at the hashed index and forward if the current ICMP ID matches
  - The hashed index is the source and destination IP addresses reverse
  - This results in the same hash value of the ICMP request, where the flow ID was stored

```
else if (hdr.icmp.type == 0) {  // ICMP echo reply

  // hash with dst and src address
  sel_hash2 = hash2.get({hdr.ipv4.dst_addr, hdr.ipv4.src_addr});

  // Register to Read identifier from hash index location (grab stored identifier)
  register_data_entry = read_data.execute(sel_hash2);

  if (register_data_entry == hdr.icmp.identifier) {  // if retrieved id == header of incoming reply packet
      ipv4_host.apply();
  }
```

Implemented Solution

- Apply Logic
  - If the TCP packet is valid and hits on tcp_policy, store the source and destination port at the hashed index of the register
  - If the TCP packet does not hit on tcp_policy, read the value at the hashed index. If the stored source and destination ports match the current packet, proceed in forwarding
  - Otherwise, drop all packets
  - The hashed index consists of the source and destination IP addresses and port numbers

Implemented Solution

TCP apply block

```
// TCP
else if (hdr.tcp.isValid()) {
    if (tcp_policy.apply().hit) {
        // Hash src, dst addresses and src, dst ports into sel_hash
        sel_hash_tcp = hash_tcp.get({hdr.ipv4.src_addr, hdr.ipv4.dst_addr, hdr.tcp.srcPort, hdr.tcp.dstPort});

        write_data_tcp.execute(sel_hash_tcp);  // Store source and dest port at location of hash

        ipv4_host.apply();
    } else {
        // Hash src, dst addresses and src, dst ports and store in sel_hash2
        sel_hash2_tcp = hash2_tcp.get({hdr.ipv4.dst_addr, hdr.ipv4.src_addr, hdr.tcp.dstPort, hdr.tcp.srcPort});

        // Return a boolean result, determining if src and dst ports match incoming packet
        bit<1> is_match;
        is_match = read_data_tcp.execute(sel_hash2_tcp);

        if (is_match == 1) {
            ipv4_host.apply();
        }
```

ICMP Evaluation

- When attempting to initiate a ping from PC1 to PC2, the action is successful
- However, attempting to ping PC1 from PC2 fails due to the implemented policy

PC1 -> PC2

```
admin@PC1:~$ ping 10.0.0.2 -c 1
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=28.8 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 28.765/28.765/28.765/0.000 ms
```

PC2 -> PC1

```
admin@PC2:~$ ping 10.0.0.1 -c 1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

TCP Evaluation

- Setting up an iPerf server on PC2 and configuring PC1 as the client results in a successful client-server connection
- Attempting to initiate PC1 as the server and PC2 as the client leads to connection refusal, again, due to TCP forwarding rules and apply logic

Successful connection

```
admin@PC1:~$ iperf -c 10.0.0.2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.1 port 34168 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-12.5 sec   640 KBytes   420 Kbits/sec
```

```
admin@PC2:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.2 port 5001 connected with 10.0.0.1 port 34168
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-13.5 sec   640 KBytes   387 Kbits/sec
```

Failed connection

```
admin@PC1:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
```

```
admin@PC2:~$ iperf -c 10.0.0.1
connect failed: Operation now in progress
```

Conclusion

- We implemented a stateful packet filter on a BMv2 software switch and on Tofino hardware switch
- Our P4 code successfully demonstrates the packet filter in accordance with environment configuration and forwarding tables
- We tested our code to allow connections originating from the internal network only
- For future work, we plan on implementing timeout on stale entries and collision resolving on flows that produce the same hash value