Danmarks Tekniske Universitet

SDN with Programmable P4-Dataplane Final Report

342x9 - Project Work in Cyber Technology and Synthesis Project for Communication Technologies

24-06-2024

AUTHORS

Rasmus A. Lindholdt	s200745
Joachim L. Espersen	s194287
Lasse Friis Olsen	s153593
Nikolai Buus	s224350

ADVISORS

Mingyuan Zang
Henrik Wessing

Abstract

This report explores how the P4 programming language can be used to implement a programmable data plane on software as well as hardware targets. Starting off on the BMv2 software target, transitioning into the Intel Tofino hardware target. The primary objective was to understand and apply various networking functionalities like simple forwarding based on IPv4 addresses, as well as more complex functionalities like a stateful firewall.
These features were successfully implemented, the probabilistic data structure of bloom filters is covered in theory as well as in practice. Furthermore the setup of the RARE/FreeRtr integration and as the BFRT/TNA enviroment are covered in detail.
To make a fully functioning P4 program is not trivial as architectures differs significantly. Forwarding as well as a stateful firewall was, however, successfull on both targets.
Future work could focus on different architectures like the Xilinx XSA architecture, offloading the hashing to a high performance FPGA as well as further stress testing and more networking features like VLANs and load balancing.

Revision Dates

- **17th of March - Functional Specification Report**
  In the functional specification report the group presents the overall idea and a broad scope of the project. An initial timeplan for the 13 week period has been presented. Finally the idea of in-group interfacing between an FPGA and the Tofino has been presented, and will be assessed later on in the project.

- **4th of June - Midterm Report**
  The scope of the project is now more specific where the router will be working along with FPGA-based hardware. Further some functionality like stateful firewall will be added to the scope of the Tofino router, and later moved to the FPGA where the P4 assessment will happen.
  A wide range of different tests has been added to the report due to the group having a functioning router capable of handling basic tasks.
  The previous idea of having an internal interfacing between two parts of the overall system has been scrapped, and the group now works together on each individual part of the project. This is due to FPGA hardware arriving late and for simplification purposes.
  The timeplan has been modified to a more specific plan for the 3-week period. Now featuring exact dates for when the group will look into different parts of the project.

- **24th of June - Final Report**
  Minor errors corrected from the previous report, and peer feedback taken into concideration for this report.
  The scope of the report has been modified. The FPGA hardware part of the implemented system has been removed. This modification was requested from advisors. The reasoning being a lack of time and a complicated piece of hardware for the time available. Now the group will be looking into a more theoretical approach where the group should consider differences between the approaches and how the implementation could have been carried out.
  A list of acronyms, revision dates, and a clear appendix section has been added to the report structure.

Acronyms

FPGA Field-Programmable Gate Array

RARE Router for Academia, Research and Education

P4 Programming Protocol-independent Packet Processors

NAT Network Address Translation

IP Internet Protocol

IPv4 Internet Protocol version 4

SDN Software Defined Networking

DPDK Data Plane Development Kit

PISA Protocol Indendepent Switch Architecture

PSA Portable Switch Architecture

TNA Tofino Native Architecture

BMv2 Behavioral model 2

API Application Programming Interface

RPC Remote Procedure Calls

gRPC gRPC Remote Procedure Calls

Bfrt Barefoot runtime

ONL Open Network Linux

UCLI Unified Command Line Interface

PM Port Management

AN Auto Negotiation

LPM Longest Prefix Match

ICMP Internet Control Message Protocol

TTL Time To Live

PHV Packet Header Vector

Contents

1 Introduction

1.1 Background

Router for Academia, Research and Education (RARE), is a project under the European GÉANT research network to develop fully programmable network routers. The project uses FreeRouter for the control plane, and P4 for the data plane.

P4, short for Programming Protocol-independent Packet Processors, is a language to program the data plane of network devices, such as switches and routers. The language allows a programmer to quickly and easily specify exactly how a given device should parse, process, check, modify and transmit any given packet it receives. The programmer gets full control over the process, down to the individual bits, allowing them freedom to implement any given network function. The same physical device can alternatively act as a switch, router, firewall, NAT, gateway, load balancer or even more, depending only on what P4 code was uploaded. The language is designed with the ability to run on general-purpose hardware, without requiring any specialized hardware for existing protocols. New features can be designed, developed and implemented at any time, without having to rely on hardware manufacturers to design and distribute new chips.

In this project the goal is to program a Tofino router using the P4 programming language, and connect it to the GÉANT network. The router will receive and forward packets to and from the network, as well as act as a rudimentary firewall to stop TCP exchanges from being initiated from outside of the network.

1.2 Objectives

The specifications and protocols of the P4 programming language are sparse and not well-explained in many papers or journals. Therefore, the group will have to spend a lot of time testing and learning how to use the language. While there are many tutorials and exercises on the basic usages of forwarding and receiving packets, more complex features will be more challenging to understand. Due to the low availability of resources on this language, the members of the group hope to help clarify and explain how to use it. The results and testing of this project can hopefully assist students and readers in the future who might want to use P4.
P4 allows for many use cases and various implementations of router operations. One important element when working with networks and external users, who might not always have good intentions, is the implementation of a firewall. The group will implement a stateful firewall to enhance overall network security. The reason for implementing a firewall is that the router will be connected to a local network, where many tests can be run and many educational cases can be tested. The local network is ideal for testing and is, therefore, optimal for a research group like this one.

The overall system will therefore use firewalls, routing, a large open network, and other components like the FPGA-based backbone. To fully cover all these requirements and technologies used, the group members have outlined the following objectives:

- Describe the project & overall system
- Implement a router capable of forwarding and receiving packets
- Implement a stateful firewall
- Test the router & firewall using Scapy & other tools
- Describe and document the RARE/FreeRouter framework and setup. Verify that the P4-forwarding and -firewalling can be offloaded to different P4-programmable dataplane targets using RARE/FreeRouter.
- Describe how the router could be offloaded onto an FPGA
- Test the overall system & perform an assessment of the final state of the project & the corrosponding tests
- Perform an assessment of finished project & its components
- Provide a user guide of how to use the various technologies used in the project

2 System Overview

The expected outcome of the project is a functioning IP router/firewall with a programmable data plane. The router/firewall will use P4 for the data plane, and can alternatively use FreeRouter/FreeRtr for the control plane and RARE for the control-dataplane communication. RARE/FreeRtr provides all standard routing features and protocols, it can distribute computed routing tables to the data planes of target routers/switches and it can be integrated with various data plane targets, which can be both virtual- or hardware-targets. RARE/FreeRtr operates on the network-level control plane. Our P4-router should be able to do basic forwarding of IPv4-traffic and should also double as a stateful firewall. Rather than having one single overall system acting as a router and firewall, we want to aim for different targets to implement our P4-functionality on, including both virtual and hardware switches. As P4 is a target independent language, it is compatible with both hardware and software targets, and can run on general-purpose hardware.

2.1 High-level Overview

In our switch-targets, the control-plane (the control logic governing how network traffic data should be managed and forwarded) is decoupled from the data-plane (the actual forwarding of the network-traffic), which is the fundamental principle of the Software Defined Networking (SDN) paradigm. Traditionally, networking devices contained local control-planes, however with SDN the control-plane is abstracted from the data-plane and the control-plane logic governing the forwarding of traffic on all forwarding devices is contained and managed on one single platform, the SDN-controller, instead of on each individual device.

While SDN simplifies network configuration by segregating the control plane from the data plane on the device-level, and enables customizing of the control plane logic from a SDN-controller, the forwarding of network traffic still remains rather inflexible when using SDN with a predefined data plane, where the networking hardware vendor has defined the forwarding behavior and which protocols can be handled by their hardware; the chip embedded in the networking hardware devices provide fixed functionality (figure 1. However, on our system - be it a virtual switch or a hardware switch - the data-plane is also programmable, so that we can customize our own dataplane and embed more functionality into our target device(s), besides just the basic forwarding functions of a switch and/or router.
When provided with a compiled P4-program, our system containing a programmable dataplane can be programmed with dedicated packet processing besides it's initial functionalities, giving it functionality unique to that particular system. Thus, a switch/router with a programmable P4-dataplane can also be given firewall functionality, in addition to the forwarding.

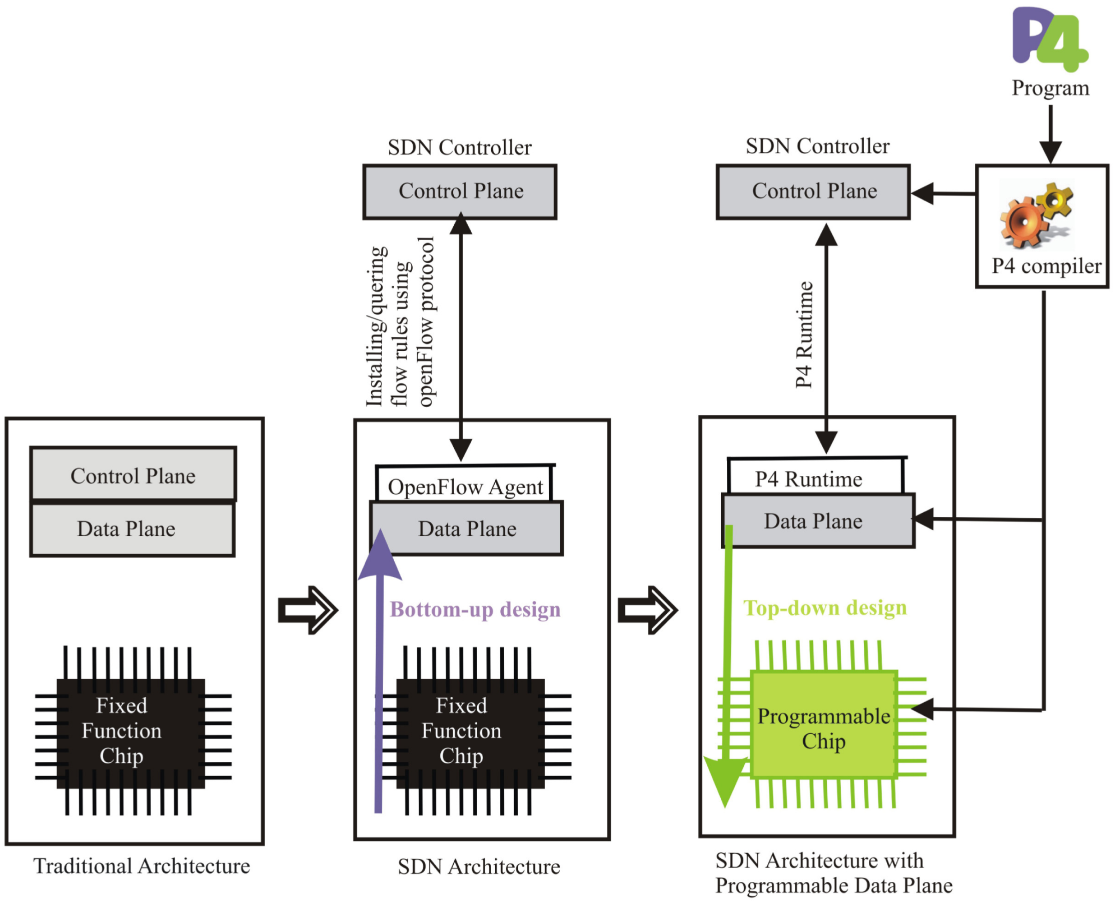


Figure 1: *From a "traditional" networking architecture to SDN-based networking using the OpenFlow protocol to SDN-based networking with a P4-programmable dataplane [8]*

2.2 System Implementation Platforms and Components

The following platforms will be used in this project:

- **RARE/FreeRtr:** RARE is an open-source routing platform created to provide support for multiple dataplanes, including DPDK, P4 and OpenFlow. RARE uses FreeRtr as a router operating system on the control plane, so we will refer to this platform as RARE/FreeRtr. FreeRtr is an open-source control-plane software providing many features within forwarding, routing and protocol security. Refer to [1] for a full summary of features. Furthermore, FreeRtr handles the packets themselves at the socket layer, meaning it is independent of the capabilities of the underlying operating system. The FreeRtr control-plane also populates the forwarding tables of our targets.

  RARE is an API which can be integrated with various different programmable target dataplanes, and their native communication APIs, as shown in figure 2. Thus, RARE/FreeRtr makes it easier to deploy the control-plane rules to different target dataplanes, due to its standardization.

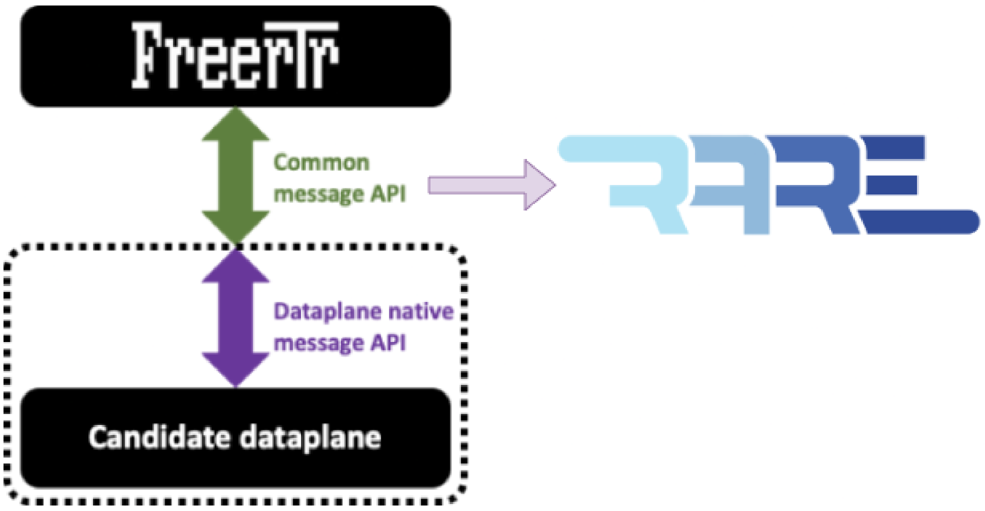


Figure 2: *RARE-FreeRouter provide a common way of integrating the control-plane with various different dataplanes and their native APIs.*

- **P4:** P4 is the language used to program the packet processing behavior of our router. The P4-language was revised in 2014 and 2016, leading to the versions denoted $P4_{14}$ and $P4_{16}$. The main difference is that $P4_{14}$ targets only devices with the Protocol Independent Switch Architecture (PISA), while $P4_{16}$ is based on a different architectural model, the Portable Switch Architecture (PSA), which is more flexible regarding the targets you can run your P4-program on. So, by opting for $P4_{16}$, we can target devices with different switching architectures than PISA. $P4_{16}$ is not backwards compatible with $P4_{14}$.

- **BMv2 and P4Runtime API:** Behavioral Model v2 (BMv2) is an open-source, PSA-based (Portable Switch Architecture) virtual P4-switch. The BMv2-platform is a good starting platform for learning the architecture of a programmable P4-switch, and how to use it, as the SDN programmable dataplane architechture shown in 1 overall is the same for the BMv2 and the Tofino switch. To deploy our P4-router functionality on the BMv2-switch, our P4-program (denoted program.p4 in figure 3) must first be compiled by a P4-compiler, creating a json-file (program.json) which is provided as input to the BMv2. The compiler we are using is called "p4c-bmv2" [4].

  In figure 3, "simple_switch_grpc" is a variation of BMv2, which exposes a gRPC P4Runtime server to the controller/control-plane. It receives and processes requests from them, e.g. to populate tables on the BMv2-switch. Other BMv2-versions exists, e.g. "simple_switch", using other frameworks like Apache Trift for implementing and managing remote procedure calls (RPC), however simple_switch_grpc have slightly lower performance overhead than simple_switch.

Besides the json-file, p4c-bmv2 also creates a "P4Info" file (program.p4info in figure 3 in txt-format containing metadata specifying the all the P4-entities defined in our P4-program (tables, actions and other P4-objects) that can be accessed via the P4Runtime API. Each object instance has an associated Identifier assigned to it by p4c-bmv2, which is used to refer to the objects in the API calls between the controller/control-plane and the P4Runtime API server on the BMv2-switch. The P4Info file is loaded by both the controller and the P4Runtime server. From the P4-entities declared in the P4Info file, the controller knows e.g. which tables on the switch it can access and populate, and the gRPC P4Runtime server uses the information in the file to translate Identifiers in received API-calls to the P4-objects, e.g. the tables instances, on the BMv2-switch. The "p4runtime.proto" is the specification file for the P4Runtime protocol defining which RPC-methods and messages that can be used between the control-plane and the P4Runtime server.

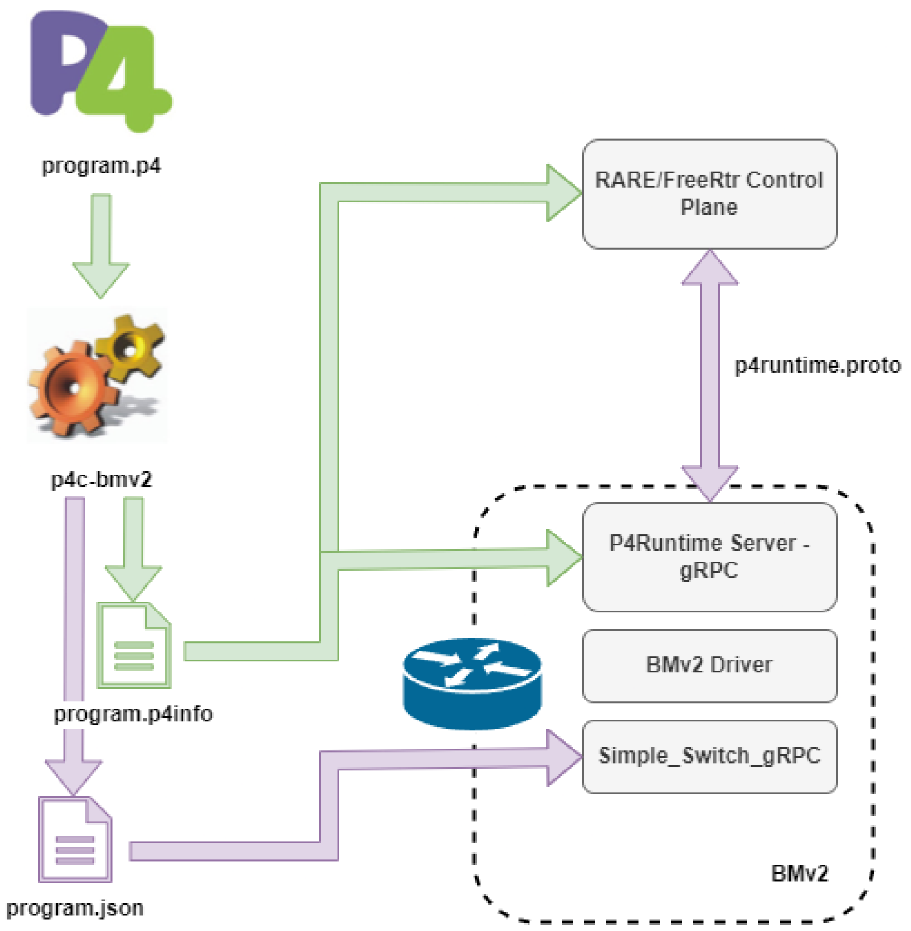


Figure 3: *Integration of control- and data-plane components on our virtual BMv2-switch, and deployment of P4-program.*

- **Tofino and BFRuntime API:** Tofino Native Architecture (TNA) is an alternative P4 target architecture developed by Intel's Barefoot Labs for their Tofino line of network switches [7]. In our project we have developed and compiled P4 on a *WEDGE-100BF-32X* model of hardware switch, which runs on the TNA architecture.

The Tofino-switch allows packet processing at higher speeds than the BMv2, allowing us to test high-performance use cases and simulate a real production environment. In the Barefoot-Tofino architecture (figure 4, BF Runtime is the native control-plane and communication to the dataplane is enabled by native Barefoot Runtime (Bfrt) Python APIs. For the Barefoot-Tofino switch, our P4-program is compiled by a "bf-p4c" compiler, generating files that e.g. define the packet processing pipeline structure of the switch, and these files are provided to the dataplane. The user executes a bf_switchd shell-script, while providing the program name as a paramenter, and that starts a process "bf_switchd" on the switch; the dataplane. The script also initializes a number of services, which besides the Bfrt-API, includes a BFRuntime gRPC server for receiving requests from the control-plane, and a bfshell, which allows us to interact with the data-plane and e.g. insert table rules using the Bfrt-API.

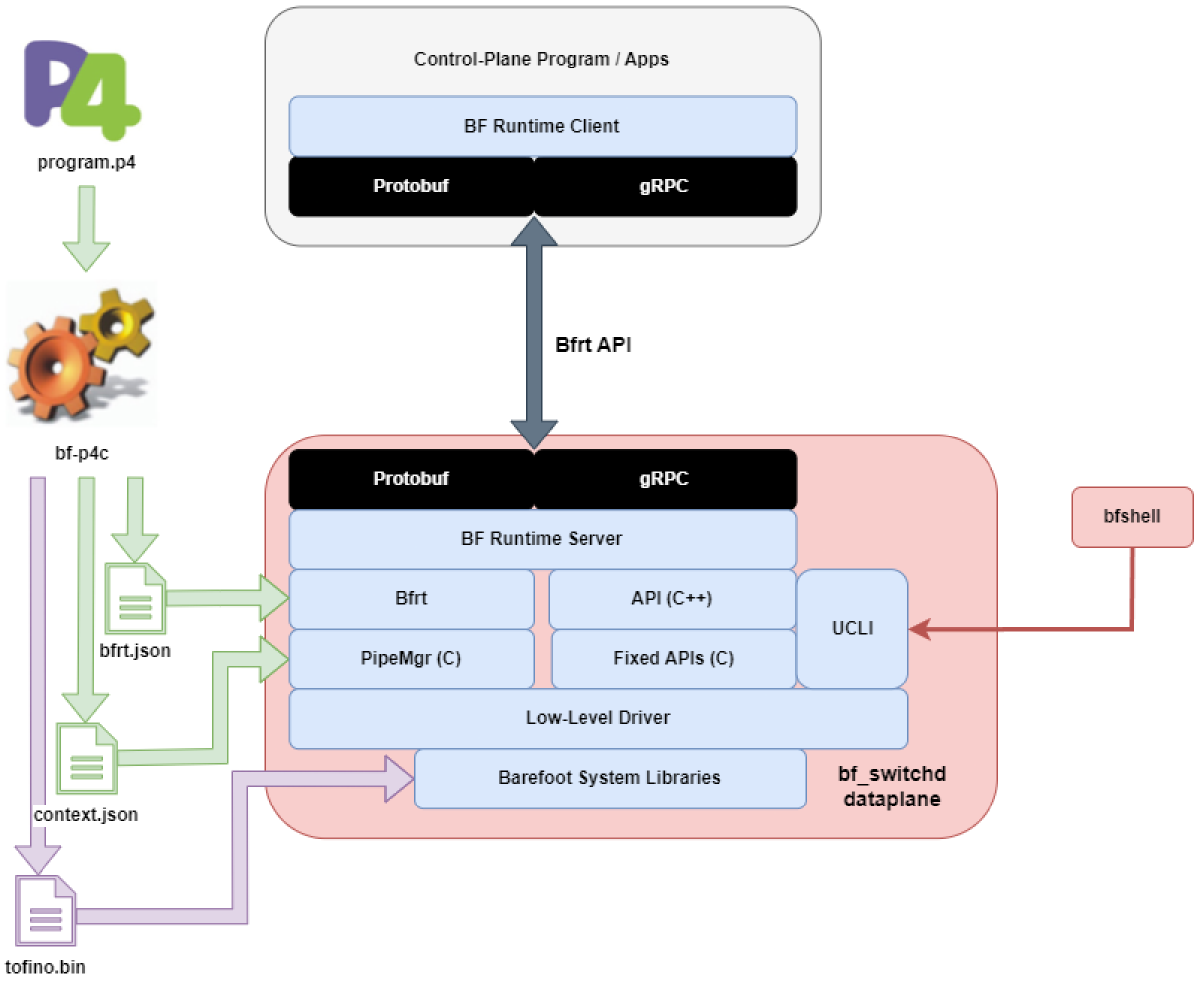


Figure 4: *Plane architecture of the Barefoot Tofino switch. Barefoot runtime is the native control-plane of this target and provides the communication API to the* `bf_switchd` *dataplane.*

3 Functional specification

In this section of the report a short description of some of the current features will be provided. The functionalities covered in this report wil firstly be on the router functionality which here is based on simple forward and receiving functionalities. Then the firewall setup used in this project will be covered. In the end the network must have a network in which it can exchange packets with. The final section will therefore describe how the GÉANT network will be used, and for which cases.

3.1 Tofino router

The router is a WEDGE-100BF-32X hardware switch, based on Intel's Tofino silicon. The router will be used for backing everything implemented by the group. Most code will be loaded to the router such as the forwarding and the firewall. In this section of the router the functionality will be explained. Firstly the IPv4 packet forwading will be explained and then an explanation of the firewall. The router will be hosting both of these featues, which is why they are both loacted under this section.

3.1.1 IPv4 packet forwarding

The router will have to be able to both receive and forward packets at all running times. The router must know what to do whenever there are packets to be received or ready for forwarding. The mechanism for receiving is done by setting up the router to a PC with an internet connection, and an active link directly to the GÉANT network. The router will then utilize the PC's connection and set up a port with an assigned NAT-addresses backed by an IPv4 address. From here the router will be recieving the packets sent to it. Once the packets are recieved the router will send them directly to the hardware based system where the packets will be examined and assessed by a firewall, before entering the system backbone. The system will then assess the packet and decide what should happen next. At some point the sender should get a reply, and therefore specified forwarding protocol is required.
When the packets has been assessed by the backbone of the system, the router must then be able to forward the next packet to the designated reciever or sender. This is done via the P4 programming language. The benefits of using this program is the enhanced possibilities for how an engineer might want a packet to be handled. Using this language the router will be effective and able to decide what should happen with each packet type that might be recived during testing.
Having these functionalities the router will function as a bridge between the backbone and the internet which in this case will be the GÉANT Network.

3.1.2 Stateful firewall

As described in the previous section the router will forward packets to the backbone from where the packets will be examined. There are however a step before we reach the backbone which is the firewall. In most cases we talk about two types of firewalls which is a stateful or stateless. In this project, the firewall will be stateful. The reason for choosing a stateful firewall over a stateless is due to its ability to provide more comprehensive and dynamic security. A stateful firewall might be more complex to implement due to the specific rules a programmer or designer can setup. Some of the features that are introduced to this type of firewall is connection tracking, performance efficiency, enhanced security, and application awareness, as well as range of other features that can be implemented by the engineers. One of the key technologies utilized by statefulness is the state tables from which the firewall keeps track of the connections currently interacting with the firewall. From here the firewall can keep a constant surveillance of connections even though they might be let through to the backbone and is able to cut the connection if something fishy happens during the exchange.

4 Theory

4.1 P4 Architecture

A P4 program is in charge of everything that happens to a packet, from the moment it enters the port on the PHY layer to the moment it leaves again. The process can be seen as a pipeline, logically divided into *blocks* with different purposes.
In standard P4, the overall steps of the pipeline are parsing, ingress processing, egress processing, and deparsing.

4.1.1 Parsing

The packet arrives to the switch as a stream of binary, and must first be parsed into some predefined data structures before the program can read, understand and modify the relevant fields.
Because P4 is protocol-independent, it has no native support for any existing network protocols like IPv4 or TCP. The programmer must therefore define what these headers are supposed to look like, describing exactly which fields belong to a header, how many bits go into each field, and in which order they are expected.
As an example, an Ethernet header (after the switch port has stripped the preamble, SFD and FCS) consists of only three fields: a destination MAC address, a source MAC address, and an ethertype field denoting which protocol is carried by the frame. A programmer defining this header could write it as

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}
```

When the programmer later calls `packet.extract(hdr.ethernet)`, the program extracts the first 48 bits of `packet`, and saves them in the `dstAddr` field of `hdr.ethernet`. It saves the next 48 bits as the source address, and the next 16 bits as the ethertype. It is now possible for the rest of the program to refer to these objects and act upon them. The first step would naturally be to read the ethertype field, to know whether to parse the next series of bits as IPv4, IPv6 or something different entirely.
The parsing process can thus be viewed as a state machine, travelling from layer to layer up the protocols in the packet. The Ethernet header can lead to IPv4 or IPv6, those can in turn lead to TCP, UDP, etc., which in turn can lead to application layer protocols like HTTPS, if parsing those is desired.

4.1.2 Ingress processing

After all necessary headers are parsed into manageable data structures, ingress processing is where these fields are read, acted upon and changed.
Fields like the `ttl` field and checksum of an IPv4 header, or the source and destination addresses of an ethernet header must be updated at each switch. This happens in ingress processing.
This block is also where decisions are made regarding routing, using *match-action tables*. These tables are defined by the P4 program, and then populated with entries by the control plane. An example of a match-action table could be

```
    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
}
```

The above code defines a table with space for 1024 entries using the destination address of the IPv4 header as a key, allowing for one out of three *actions* to be called with parameters supplied by the control plane. The control plane can add entries of IPv4 addresses and subnets, defining which of the three actions they should trigger, and which parameters to pass to the action. In this case, the control plane would define a number of different IP addresses and ranges, then pair them with the correct exit port and MAC addresses for the next hop, and pass this to the `ipv4_forward` function.
This particular table uses *longest prefix matching* (LPM). As IP ranges can encompass each other, LPM makes sure only the most specific range containing the address is selected. Otherwise, matching against `0.0.0.0\0` (the entire IPv4 address space) as a default route would overrule all other entries.

4.1.3 Egress processing

Between ingress and egress, the packet is added to the packet buffer and handled by the traffic manager, which is responsible for scheduling packets to be output by the switch. During egress, it is no longer possible to change the output port of the packet, but other last-nanosecond changes are still permitted. This can be useful for latency measurements and the like.

4.1.4 Deparsing

The last step is deparsing, which is where the fields of the packet are assembled back into a bitstream, and emitted by the exit port.

4.2 Bloom filter

A Bloom filter (named after Burton Howard Bloom) is a *probabilistic* data structure, which trades some correctness for a big increase in performance, both in space and in time.
A Bloom filter describes a *set* of elements. Elements can be added to the set, and the set can later be tested against to check whether an element has been added previously. It is, however, not possible to remove elements from the set again.
Because of the probabilistic nature of the algorithm, when testing whether an element is part of the set or not, false positives may occur, but not false negatives. In essence, the response will either be "probably in the set" or "definitely not in the set".

The Bloom filter is made up of an array of m bits, all initialized to 0, as well as k different hash functions. The hash functions are each applied to the input element to compute k different array indices. Ideally the output indices should be uniformly distributed between 0 and m.
Adding an element entails setting each of these k array positions to 1, testing for the presence of an element is simply testing whether all relevant bits are 1. All k hash functions must be computed in either case, so the time complexity of insertion and access are both $\mathrm{O}(k)$. The space complexity is $\mathrm{O}(m)$.
Because of the limited amount of array positions compared to the output space of most hash functions, hash collisions are likely to occur. This is how false positives occur when testing for the presence of an element, especially when the Bloom filter fills up with more elements. The error probability ε of receiving a false positive, based on the number of inserted elements n, can be calculated as:

$$\varepsilon = (1 - e^{-kn/m})^k \tag{1}$$

In our project, we use a Bloom filter to keep track of established TCP connections, for the stateful firewall. To add a connection to the filter, we concatenate the IP source and destination addresses, as well as the TCP source and destination ports, and hash them all together.

4.3 Tofino Native Architecture

P4 programs are able to be run on a variety of different packet-forwarding hardware and software, denoted as *targets*. The manufacturers of P4 targets are responsible for providing the compiler that turns any valid P4 program into code that can run on the target. Furthermore, they must provide an API to facilitate communications between the P4 data plane and any kind of control plane. As long as the API is used correctly, the control plane can be anything: Any kind of hardware or software, even a human manually entering commands in a terminal.

In this project, we intend to run our P4 code on a Tofino switch, designed and built by Intel. To do that, we'll need to rewrite some of the code to work with Tofino Native Architecture (TNA), as opposed to the default Portable Switch Architecture (PSA). We will also have to use the BFRuntime API for the connection between the data plane and the control plane. The perk of using TNA is the ability to make use of specialized hardware features of the Tofino switch ASICs, that are not by themselves defined in the standard P4 specification. [7]

One exclusive feature of this architecture is the fact that Tofino switches are divided into multiple parallel *pipes*, that can optionally be configured with separate P4 code.
TNA also provides access to a variety of *externs*, hardware features of the Tofino switch that are encapsulated into program objects with methods. Externs include things like registers, counters, meters, a 16-bit ones' complement checksum calculator (used for IPv4 packets), a variety of hash algorithms, a number of Arithmetic Logic Units (ALU) for quick mathematical operations, and more.

We use a *register* extern to hold our Bloom filter. This is necessary to keep the data persistent between the processing of multiple packets, as saving it in the P4 program itself would mean that the filter was reset for each new packet.

4.4 AMD FPGA target

Initially one of the targets for the project was the Alveo U55C High Performance Compute Card, with two QSPF28 network interfaces. This is an FPGA board developed by Xilinx (now part of AMD), made to excel in applications with high computational requirements as well as low latency. It has high bandwidth memory and would be an interresting target for the firewall program.
Due to limited time and unexpected complexity to the setup of the card, advisor Mingyuan recommended simply researching the card and related components.
One of the components is provided by Xilinx and is an FPGA-based NIC platform called OpenNIC. OpenNIC consists of multiple subcomponents like a NIC shell, Linux kernel drivers and a Data Plane Development Kit (DPDK). The OpenNIC Shell provides the hardware design for the specific card (Alveo U55C in this case) and is programmed via a bitstream generation from Vivado. A guide on how to build, simulate and program the FPGA is found via. [2].
Another more complete platform is the ESnet SmartNIC platform, comprised of multiple seperate components, including the OpenNIC shell. This has a tutorial, linked in [6].
This project is more focused on the fundamental differences in the architecture, going from the TNA architecture to Xilinx's architecture known as XSA.
ESnet provides simple example P4 source code for eg. layer 2 forwarding, see [5]. It is similar to the TNA architecture, but simplified in the example with a single parser, match-action

implementation and a deparser. No ingress and egress separation. Furthermore it requires a structure `smartnic_metadata` with 152 bits for different purposes like timestamp in ns, 16 bit packet ID and more. Other than that, it is just another P4 architecture that the developer has to learn. The API for the control plane as well as getting the card and interfaces up and running may prove to be the most difficult part of the project. This was not researched further, as the group never got to play with either of the components and research became abstract.

5 Implementation

5.1 RARE/FreeRtr-BMv2 Integration and Verification

This section explains how to install a FreeRtr environment on a Virtual Machine and deploy an instance of a RARE/FreeRtr control-plane on the VM, provide external connectivity between the FreeRtr-instance and the LAN-network of the VM-host, and finally integrate RARE/FreeRtr with the BMv2-dataplane and install a P4-program on the BMv2 virtual P4-switch. For a full guide on commands used for integration, refer to appendix D. The VM/OS used for the RARE/FreeRtr-BMv2 implementation is Linux Ubuntu (64-bit) v22.04. Automatic deployments of FreeRtr exists for Ubuntu v18.04 and v20.04, however v22.04 was chosen, instead for manual installation for reasons explained later. The reason for first doing an integration of the RARE/FreeRtr controlplane with the BMv2 dataplane, is because it is a great way of learning the SDN architecture of a switch with a programmable dataplane, the components of the architecture (P4-compiler, Runtime-API, runtime server, etc.) and how the control- and data-planes communicate, because there are several architectural similarities between the BMv2 and the Tofino-switch. Besides the BMv2 was the P4-programmable dataplane target first introduced early in the project for learning purposes.

As FreeRouter is written in Java, Java Runtime Environment (which was missing on the Linux VM) was first installed on the VM. Then a FreeRtr directory environment was created to store binary-, library-, log- and configuration-files for FreeRtr. The FreeRouter pulled as a .jar-file was compiled using Java 11, thus Java 11 or later was required, and launching a freeRtr environment with an older JRE version would give an "Unsupported Class Version Error".

A deployed FreeRouter-instance naturally provide much more utility when it has connectivity out of the VM-scope, to the LAN of the VM-host. To make the FreeRouter accessible from the LAN, the FreeRtr net-tools are installed. Our FreeRouter-instance requires two configuration txt-files: a hardware specification file and a software specification file. In the hardware specification file, the network interfaces for the FreeRouter-instance are declared, including their type (e.g. Ethernet), MAC-address, and enabling remote reachability on a tcp-port. The software specification file contains information, e.g. regarding the default IPv4-route of the FreeRouter-instance and configuring a static IP-address for it (or specify it as dynamic). From FreeRtr net-tools, PcapInt binary is used to bind a FreeRouter socket specified in the hardware configuration file to an available physical network interface of the VM-host (figure 5). `pcapInt.bin` requires glibc version 2.34 or later, and the glibc version is pretty much fixed by the Linux Distribution/Version. Hence, Ubuntu v22.04 was used as it comes with glibc v2.35. In the software specification file of freerouter, the interface denoted `eth1` is configured to act as an IPv4-client, thus 192.168.0.105 - provided by the DHCP-server, i.e the default gateway - becomes the IPv4-address of the freerouter-instance (figure 5).

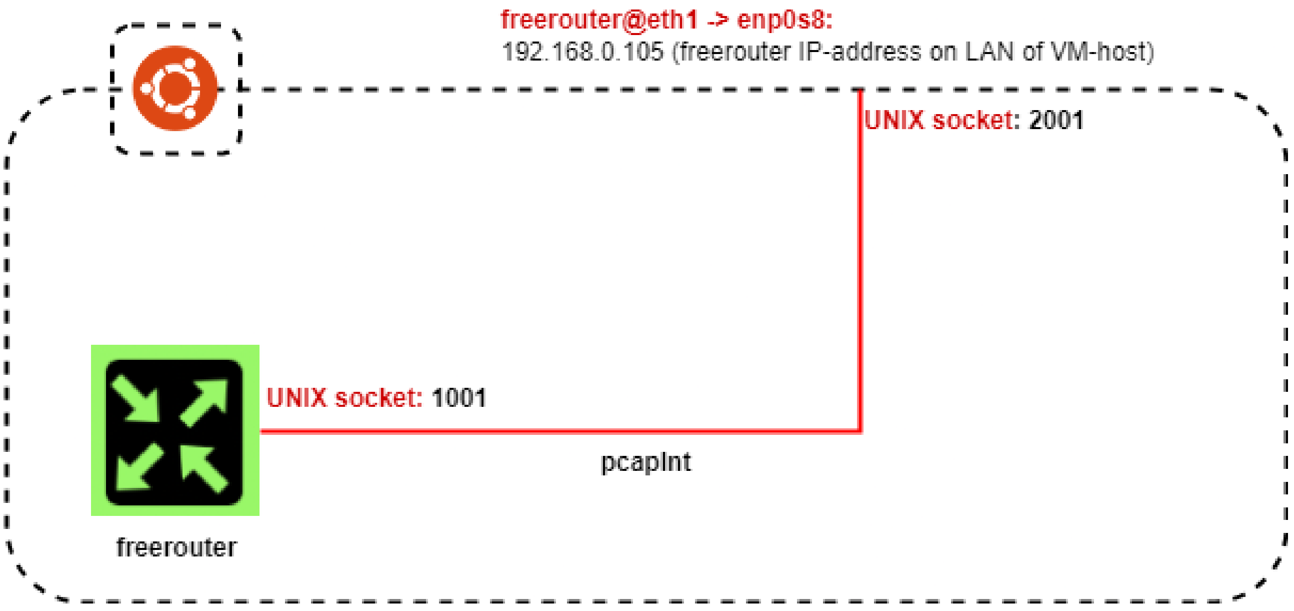
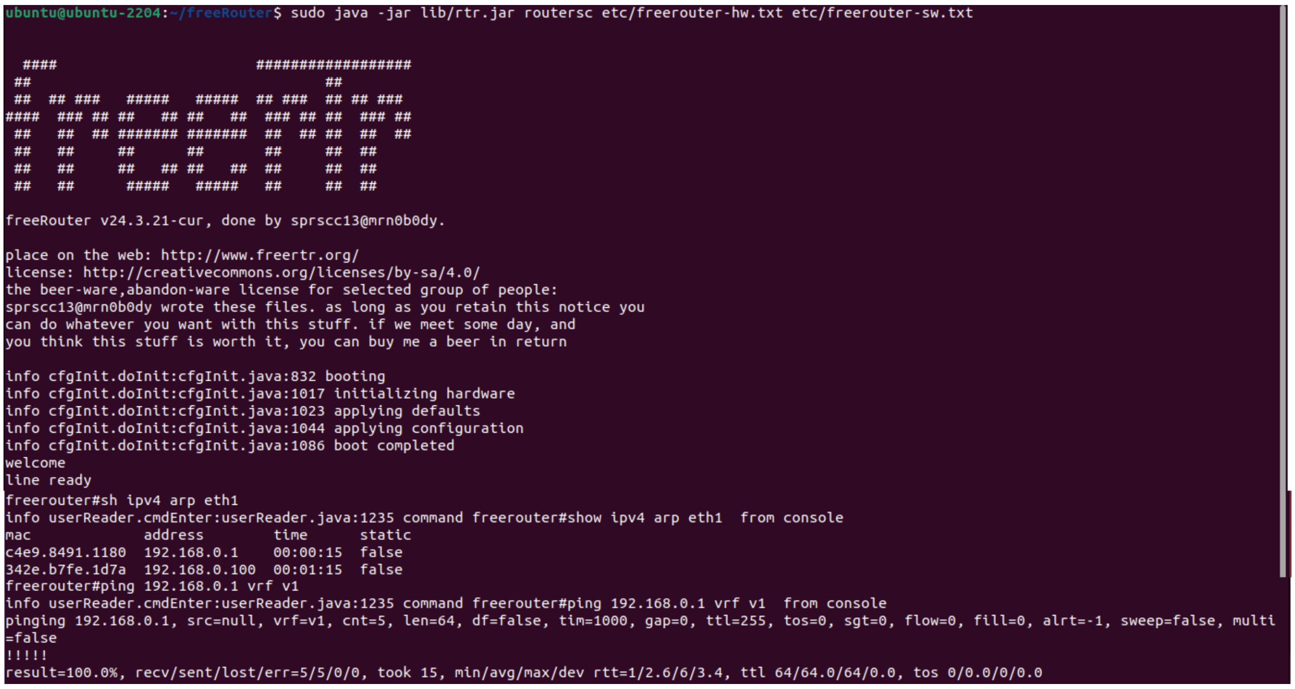

freerouter@eth1 -> enp0s8:
192.168.0.105 (freerouter IP-address on LAN of VM-host)
UNIX socket: 2001
UNIX socket: 1001
pcapInt
freerouter

Figure 5: *Unix sockets are used for communication between the running freerouter-instance process and the outside. pcapInt maps Unix socket 2001 to the available physical network interface* ***`enp0s8`****, and stitches socket 2001 with 1001.*



```
ubuntu@ubuntu-2204:~/freeRouter$ sudo java -jar lib/rtr.jar routersc etc/freerouter-hw.txt etc/freerouter-sw.txt


  ####                       ##################
 ##                                  ##
 ##  ## ###   #####   #####  ## ###  ## ## ###
####  ### ## ##   ## ##   ##  ### ## ##  ### ##
 ##   ##  ## ####### #######  ##  ## ##  ##  ##
 ##   ##     ##      ##       ##     ##  ##
 ##   ##     ##   ## ##   ##  ##     ##  ##
 ##   ##      #####   #####   ##     ##  ##

freeRouter v24.3.21-cur, done by sprscc13@mrn0b0dy.

place on the web: http://www.freertr.org/
license: http://creativecommons.org/licenses/by-sa/4.0/
the beer-ware,abandon-ware license for selected group of people:
sprscc13@mrn0b0dy wrote these files. as long as you retain this notice you
can do whatever you want with this stuff. if we meet some day, and
you think this stuff is worth it, you can buy me a beer in return

info cfgInit.doInit:cfgInit.java:832 booting
info cfgInit.doInit:cfgInit.java:1017 initializing hardware
info cfgInit.doInit:cfgInit.java:1023 applying defaults
info cfgInit.doInit:cfgInit.java:1044 applying configuration
info cfgInit.doInit:cfgInit.java:1086 boot completed
welcome
line ready
freerouter#sh ipv4 arp eth1
info userReader.cmdEnter:userReader.java:1235 command freerouter#show ipv4 arp eth1  from console
mac             address        time      static
c4e9.8491.1180  192.168.0.1    00:00:15  false
342e.b7fe.1d7a  192.168.0.100  00:01:15  false
freerouter#ping 192.168.0.1 vrf v1
info userReader.cmdEnter:userReader.java:1235 command freerouter#ping 192.168.0.1 vrf v1  from console
pinging 192.168.0.1, src=null, vrf=v1, cnt=5, len=64, df=false, tim=1000, gap=0, ttl=255, tos=0, sgt=0, flow=0, fill=0, alrt=-1, sweep=false, multi
=false
!!!!!
result=100.0%, recv/sent/lost/err=5/5/0/0, took 15, min/avg/max/dev rtt=1/2.6/6/3.4, ttl 64/64.0/64/0.0, tos 0/0.0/0/0.0
```

Figure 6: *The FreeRouter CLI after launching freerouter. The command* **`sh ipv4 arp eth1`** *show the arp cache of the interface* **`eth1`** *specified for the freerouter.* `192.168.0.1` *is IP-address of the default gateway of the LAN of the VM-host,* `192.168.0.100` *is the VM-host itself. Notice that pinging the default gateway is successful, meaning we have external connectivity for our freerouter instance.*

Proceeding with the integration of the Rare/Freerouter control-plane with a BMv2 P4-dataplane, a P4-environment is first installed on the Ubuntu VM, as per the instruction listed in D.0.3, providing us with the P4Runtime API, the P4C compiler and the BMv2-switch needed to implement the setup in figure 3. Communication between the FreeRtr control-plane and the BMv2 data-plane is enabled by RARE forwarder.py - a Python script based on the gRPC P4Runtime Python library implementing the P4Runtime API; the script and other utilities are cloned into the environment from another Github repository (refer to D.0.3 step 2). Our P4-program, denoted "router.p4" in D.0.3 step 2, is compiled with the p4c-compiler, generating two files "router.txt" (the P4info-file) and "router.json", as described in subsection 2.2. Again, hardware- and software configuration files are made for our freerouter instance, which is to be connected to the BMv2 data-plane, before the instance is launched (step 5). The FreeRtr control-plane and the BMv2-dataplane are connected with a virtual Ethernet link (D.0.3 step 4). The pcapInt-binary from the FreeRtr net-tools binds a UNIX-socket of the FreeRtr-instance (22710) to a virtual network interface (veth250) of our BMv2-dataplane D.0.3 step 6), and veth250 is in turn connected to CPU-port 64 which is specified in D.0.3 step 7. When the BMv2-dataplane is launched, a gRPC runtime server is also started along with it (D.0.3 step 6). The last implementation step is where the P4Info- and json-files generated from our compiled P4-program "router.p4" are provided to the running FreeRtr control-plane, the gRPC P4Runtime server and the simple_switch_grpc, as described in subsection 2.2. When the RARE script `forwarder.p4` is run, P4Runtime is enabled between the control- and data-plane (figure 8: freerouter notifies that a neighbor is up). Figure 8 confirms that packets start being transmitted out the interface `ethernet0`, connecting to the simple_switch_grpc BMv2 data-plane, and interface `sd1` providing external connectivity for our Rare/freeRtr-BMv2 setup, indicating that Rare/freeRtr is communicating with the BMv2 data-plane.

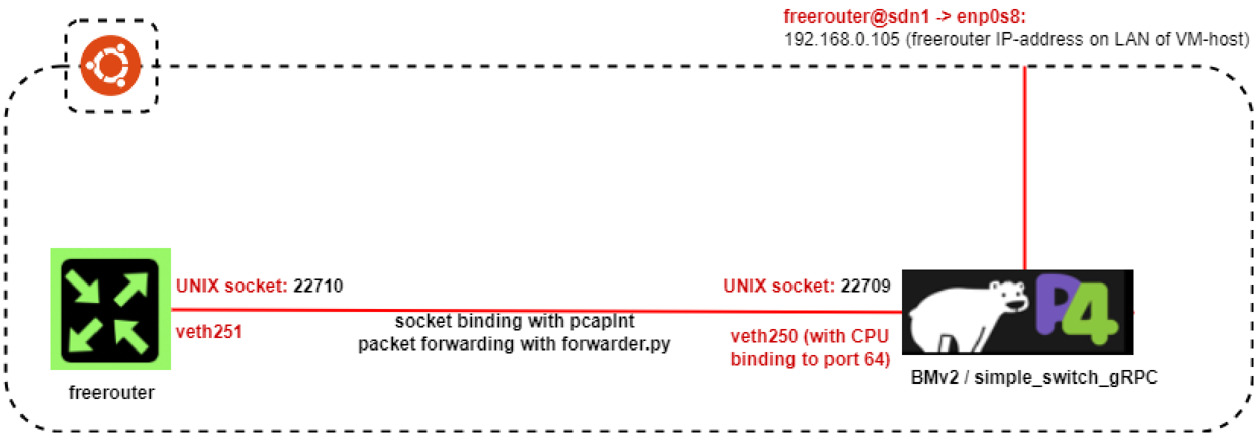
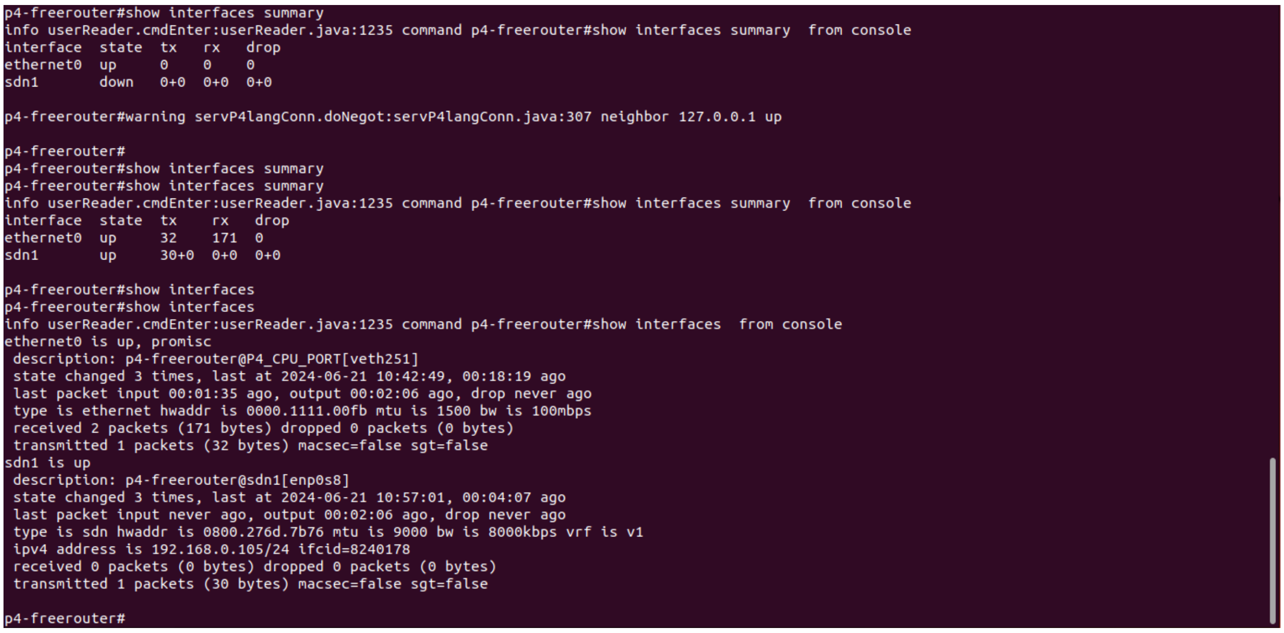

freerouter@sdn1 -> enp0s8:
192.168.0.105 (freerouter IP-address on LAN of VM-host)
UNIX socket: 22710
veth251
socket binding with pcapInt
packet forwarding with forwarder.py
UNIX socket: 22709
veth250 (with CPU binding to port 64)
freerouter
BMv2 / simple_switch_gRPC

Figure 7: *Overview of Rare/freeRtr-BMv2 setup on Ubuntu VM.*



```
p4-freerouter#show interfaces summary
info userReader.cmdEnter:userReader.java:1235 command p4-freerouter#show interfaces summary  from console
interface  state  tx    rx   drop
ethernet0  up     0     0    0
sdn1       down   0+0   0+0  0+0

p4-freerouter#warning servP4langConn.doNegot:servP4langConn.java:307 neighbor 127.0.0.1 up

p4-freerouter#
p4-freerouter#show interfaces summary
p4-freerouter#show interfaces summary
info userReader.cmdEnter:userReader.java:1235 command p4-freerouter#show interfaces summary  from console
interface  state  tx     rx   drop
ethernet0  up     32     171  0
sdn1       up     30+0   0+0  0+0

p4-freerouter#show interfaces
p4-freerouter#show interfaces
info userReader.cmdEnter:userReader.java:1235 command p4-freerouter#show interfaces  from console
ethernet0 is up, promisc
 description: p4-freerouter@P4_CPU_PORT[veth251]
 state changed 3 times, last at 2024-06-21 10:42:49, 00:18:19 ago
 last packet input 00:01:35 ago, output 00:02:06 ago, drop never ago
 type is ethernet hwaddr is 0000.1111.00fb mtu is 1500 bw is 100mbps
 received 2 packets (171 bytes) dropped 0 packets (0 bytes)
 transmitted 1 packets (32 bytes) macsec=false sgt=false
sdn1 is up
 description: p4-freerouter@sdn1[enp0s8]
 state changed 3 times, last at 2024-06-21 10:57:01, 00:04:07 ago
 last packet input never ago, output 00:02:06 ago, drop never ago
 type is sdn hwaddr is 0800.276d.7b76 mtu is 9000 bw is 8000kbps vrf is v1
 ipv4 address is 192.168.0.105/24 ifcid=8240178
 received 0 packets (0 bytes) dropped 0 packets (0 bytes)
 transmitted 1 packets (30 bytes) macsec=false sgt=false

p4-freerouter#
```

Figure 8: *Verification that the Rare/freeRtr control-plane is communicating with the BMv2 data-plane, on the `ethernet0` interface.*

5.2 RARE/FreeRtr Implementation on Tofino-Switch and Test

In summary, a successful RARE/FreeRtr-BMv2 was done, and traffic dumps from the interfaces of the FreeRouter control-plane verifies that it is communicating with the BMv2-switch. This section documents the implementation and result of RARE/FreeRouter on our other programmable dataplane target - the Barefoot-Tofino switch.

Referring to the achitecture of 4, the Barefoot-Tofino switch already contains a P4-environment, with a P4-compiler bf-p4c, and a native control-plane and communication API provided as/by Barefoot Runtime. The dataplane is `bf_switchd`.

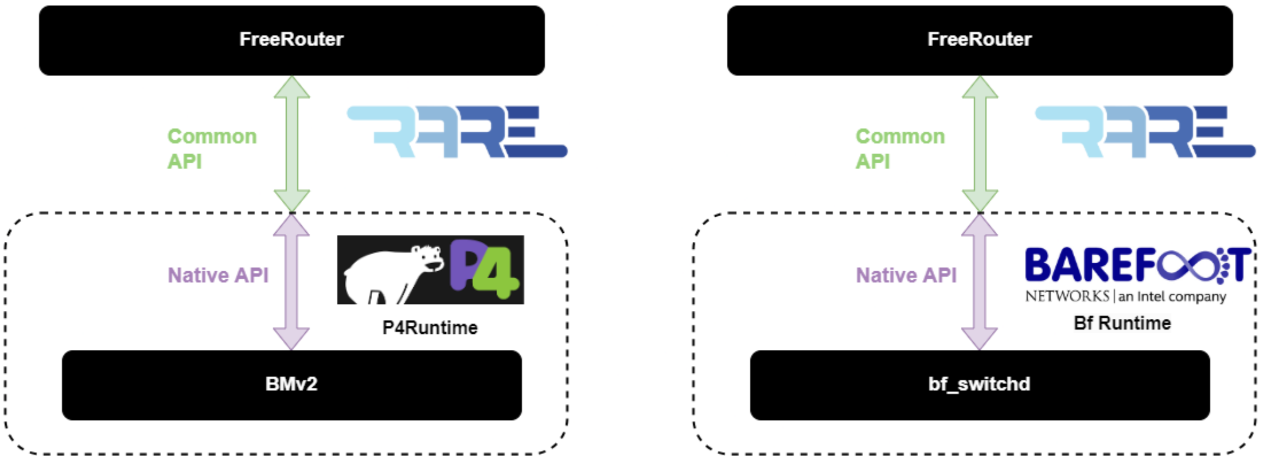
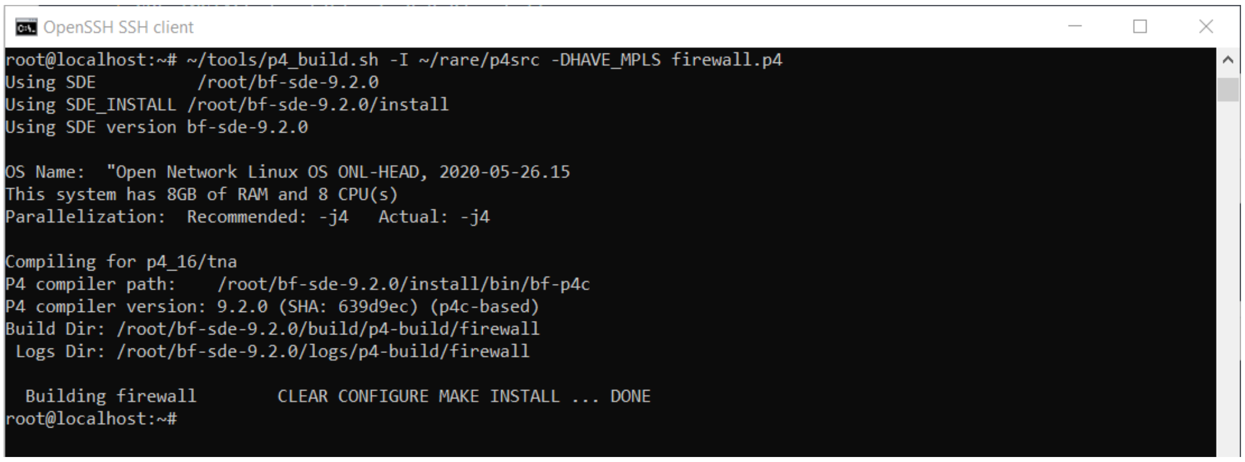


Figure 9: *High-level overview of RARE/FreeRouter implementation with our two target dataplanes: BMv2 and bf_ switchd*

However, rather than using the native Barefoot Runtime, FreeRouter is implemented as a control-plane onto the Barefoot-Tofino switch, along with RARE providing the runtime interface to the `bf_switchd`. First, a FreeRouter environment was set up on the Barefoot-Tofino swtich in the same manner as for BMv2 (refer to D.0.4, step 1 and 3). Communication between the FreeRtr control-plane and the bf_switchd data-plane is enabled by RARE bf_forwarder.py, which implements and executes a gRPC-client that communicates the the BF Runtime gRPC server in the bf_switchd dataplane (refer to figure 4 for the BareFoot-Tofino architecture).

Again, hardware- and software configuration files are made for the freerouter instance, which is to be connected to the bf_switchd dataplane (D.0.4, step 4). Our final P4-program `firewall.p4` is compiled with the bf-p4c compiler

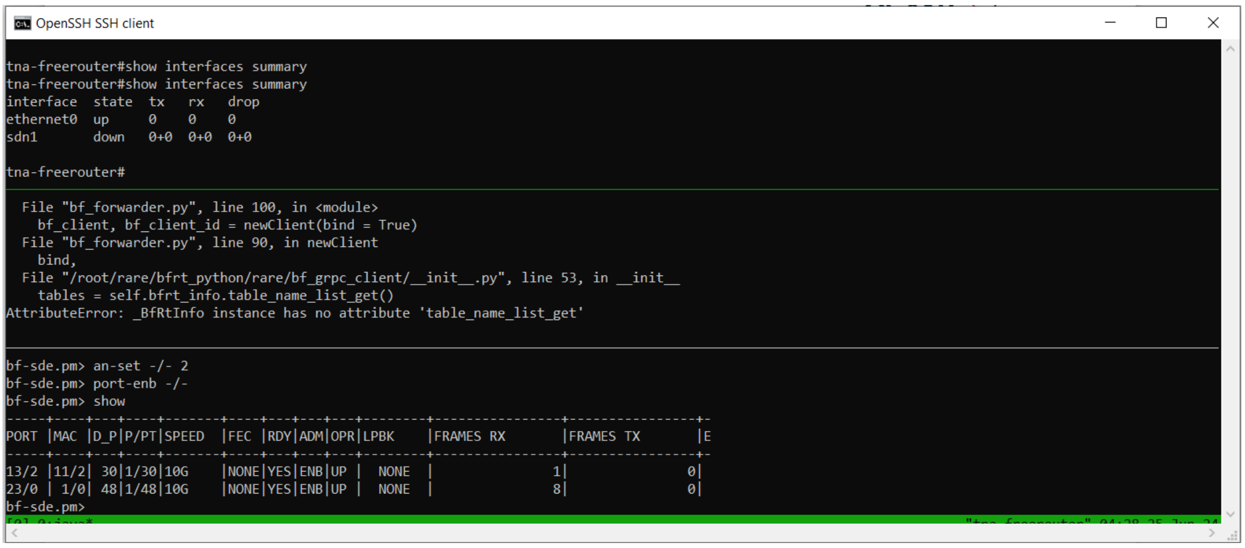


```
root@localhost:~# ~/tools/p4_build.sh -I ~/rare/p4src -DHAVE_MPLS firewall.p4
Using SDE          /root/bf-sde-9.2.0
Using SDE_INSTALL /root/bf-sde-9.2.0/install
Using SDE version bf-sde-9.2.0

OS Name:  "Open Network Linux OS ONL-HEAD, 2020-05-26.15
This system has 8GB of RAM and 8 CPU(s)
Parallelization:  Recommended: -j4   Actual: -j4

Compiling for p4_16/tna
P4 compiler path:    /root/bf-sde-9.2.0/install/bin/bf-p4c
P4 compiler version: 9.2.0 (SHA: 639d9ec) (p4c-based)
Build Dir: /root/bf-sde-9.2.0/build/p4-build/firewall
 Logs Dir: /root/bf-sde-9.2.0/logs/p4-build/firewall

  Building firewall        CLEAR CONFIGURE MAKE INSTALL ... DONE
root@localhost:~#
```

Figure 10: *Successful compilation of our firewall p4-program on the Barefoot-Tofino switch.*

The bf_switch dataplane is started (D.0.4, step 6), and the name of the P4-program is provided as parameter to the shell-script. That process provides the .json-files and tofino.bin file generated from the P4-program compilation to the bf_switchd dataplane drivers 4. After starting the dataplane, the physical TOFINO ports can be configured through the UCLI stated from the bf-shell, as per C.4. In the FreeRouter, an interface `sdn1` is specified and mapped to one of the TOFINO ports using its identifier D_P. In the FreeRouter configuration files, an interface `ethernet0` is also specified as the interconnection port with the bf_switchd dataplane, and for that interface, `enp4s0f1` is specified to enable access to the P4 CPU port no. 64. Next step of the implementation is then the setup of the communication channel between the FreeRouter control-plane and the bf_switchd dataplane (D.0.4, step 8), where e.g. CPU port 64 is enabled by bringing interface `enp4s0f1` up. The FreeRouter instance is started (D.0.4, step 9).

With a running FreeRouter control-plane and bf_switchd P4-dataplane, the last thing is to initiate the RARE-interface between them through the RARE bf_forwarder where the P4-program name is provided as parameter. Figure 11 displays the running FreeRouter-instance and bf_switchd dataplane, however the RARE-interface fails to initiate the communication between them. At the time of writing, the issue was not yet resolved. It could be just Python-errors, or maybe something more in the implementation. Figure 12 shows what the terminal output should be, when the RARE-interface is started, where the address of the gRPC-client started by RARE is displayed, the P4-program ("bf_router" in that case) is identified, and the status saying that `bf_forwarder.py` is running.

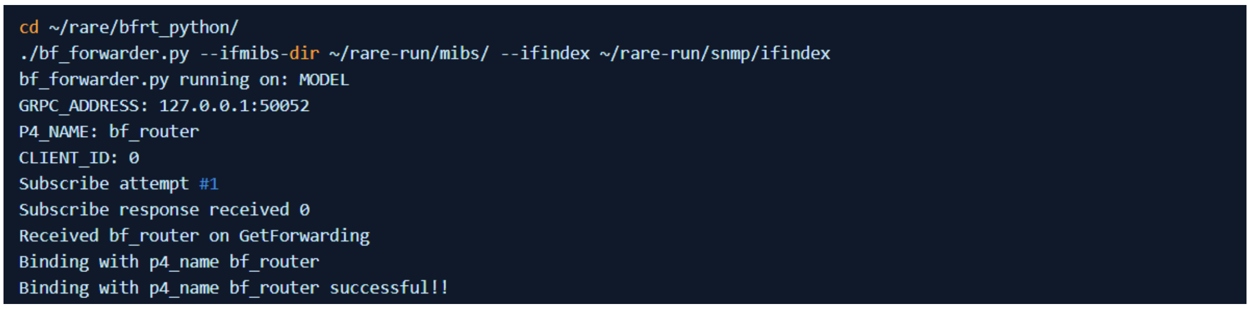


```
tna-freerouter#show interfaces summary
tna-freerouter#show interfaces summary
interface  state  tx   rx   drop
ethernet0  up     0    0    0
sdn1       down   0+0  0+0  0+0

tna-freerouter#
```

```
  File "bf_forwarder.py", line 100, in <module>
    bf_client, bf_client_id = newClient(bind = True)
  File "bf_forwarder.py", line 90, in newClient
    bind,
  File "/root/rare/bfrt_python/rare/bf_grpc_client/__init__.py", line 53, in __init__
    tables = self.bfrt_info.table_name_list_get()
AttributeError: _BfRtInfo instance has no attribute 'table_name_list_get'
```

```
bf-sde.pm> an-set -/- 2
bf-sde.pm> port-enb -/-
bf-sde.pm> show
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |RDY|ADM|OPR|LPBK    |FRAMES RX       |FRAMES TX       |E
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
13/2 |11/2| 30|1/30|10G    |NONE|YES|ENB|UP |  NONE  |               1|               0|
23/0 | 1/0| 48|1/48|10G    |NONE|YES|ENB|UP |  NONE  |               8|               0|
bf-sde.pm>
```

Figure 11: *In the topmost SSH-pane, the FreeRouter-instance is running. The bottommost pane has the bf_switchd dataplane running. In the middle pane, the RARE-interface is started, however communication between the control- and data-planes fail. Cause unknown.*



```
cd ~/rare/bfrt_python/
./bf_forwarder.py --ifmibs-dir ~/rare-run/mibs/ --ifindex ~/rare-run/snmp/ifindex
bf_forwarder.py running on: MODEL
GRPC_ADDRESS: 127.0.0.1:50052
P4_NAME: bf_router
CLIENT_ID: 0
Subscribe attempt #1
Subscribe response received 0
Received bf_router on GetForwarding
Binding with p4_name bf_router
Binding with p4_name bf_router successful!!
```

Figure 12: *Expected outcome when initiating the RARE-interface between the FreeRouter and the bf_switchd.*

5.3 Stateful Firewall Implementation using Bloom filters

This section describes how the stateful firewalling functionality is implemented in the P4-dataplane. Rather than giving a description of the entire P4-code, from parser to deparser (refer to appendix E for the full code), the description focuses on the specific componenents that enable the stateful functionality, allowing the firewall to monitor individual connections and keep track of their state; namely the Bloom filters and the hashing of incoming packets. In the project, P4-code for the firewall have been based on the Portable Switch Architecture and Tofino Native Architecture, which differ in their architectures, however the implementation using Bloom filters is the same in both cases. The code presented here are for the TNA, though.

As mentioned in subsection 4.2, the Bloom filters stores information about a set of elements and checks whether and element is (possibly) a member of the set. With the two Bloom filters used to implement the stateful firewall, a set of established tcp-connections is maintained. The Bloom filters are implemented as two different register arrays in P4:

```
    Register<bit<1>,_>(Bloom_Filter_Entries) bloom_filter_1;
    Register<bit<1>,_>(Bloom_Filter_Entries) bloom_filter_2;
    bit<1> direction;
```

Above, `Bloom_Filter_Entries` specifies the size of the Bloom filters, thus it determines the number of tcp-connections that can be monitored. The variable `direction` specifies the direction of the packet flow, i.e. is it coming from the internal or the external network. When a tcp-packet is received, the direction of the packet is first checked, as it determines whether to update the Bloom filters or run a packet-check on them. The Bloom filters must only be updated with a new tcp-connection when a host on the internal network established a connection to the outside. External hosts cannot initate connections to the internal network - that is our security policy.

When a new tcp-connection is to be added to the set, that is when a SYN-packet is received, the filters are updated by two different hash-functions, crc16 and crc32; one for each filter. Adding a new tcp-connection is done by hashing a 5-tuple of information extracted from the IP- and TCP-headers (e.g. source- and destination IP-addresses) of a TCP SYN-packet, to two register positions/indices:

```
if (direction == 0) {

    if (hdr.tcp.syn == 1) {

        Hash<bit<32>>(HashAlgorithm_t.CRC16) hash_10;
        Hash<bit<32>>(HashAlgorithm_t.CRC32) hash_20;

        index1 = (bit<32>)(hash_10.get({packet hdr info}));
        index2 = (bit<32>)(hash_20.get({packet hdr info}));

        bloom_filter1_set.execute(index1);
        bloom_filter2_set.execute(index2);
    }
}
```

Above, `hash.get` computes the hash values of the packet header information. Generally, the crc-algoithms enable very fast computation of hashes. `bloom_filter_set.execute(index)` triggers a register action that updates the Bloom filters with 1's at the indices positions (figure 13).

When a tcp-packet comes from the external network, a packet-check is run on the Bloom filters to determine whether it is part of an established and allowed tcp-connection. Below, `bloom_filter_get.execute(index)` triggers a register action that reads the Bloom filters at the computed indices positions, and the packet will be dropped if it is not in either Bloom filter, i.e. if either values read from the filters are not 1. The order of the hashed packet header information is changed, i.e. IPv4 destination addresses come before source addresses, and tcp destination ports come before source ports, to account for the opposite directional flow of the packet - otherwise, the hash indices computed when reading the Bloom filters would not correspond to the hash indices computed when updating the filters, meaning a tcp-packet coming from the external network wrongly would be determined to not be part of an established tcp-connection even though it actually is.

```
if (direction == 1) {

    index1 = (bit<32>)(hash_10.get({packet hdr info}));
    index2 = (bit<32>)(hash_20.get({packet hdr info}));

    meta.bloom_read_1 = bloom_filter1_get.execute(index1);
    meta.bloom_read_2 = bloom_filter2_get.execute(index2);

    if (meta.bloom_read_1 != 1 || meta.bloom_read_2 != 1) {
        drop();
    }

}
```

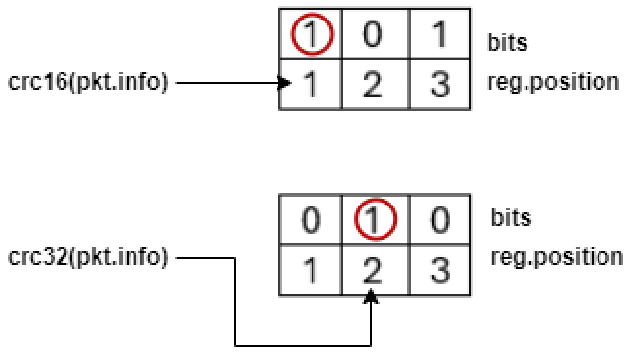


Figure 13: *Hashing TCP SYN-packet information and updating Bloom filters with the new TCP-connection. Initially, the filters contains only 0's.*

6 Testing and Validation

6.1 Testing in the BMv2/PSA enviroment

This subsection covers the initial process of learning the P4 language. Advisor Mingyuan provided a Ubuntu test enviroment with all the necessary tools to compile and run tests in the Behavioral model 2 (BMv2) software enviroment. All the tutorials are provided by the p4lang team, see [3]. The process of doing the tutorials provided by the p4lang won't be covered here, as there are many small tutorials demonstrating different features like forwarding, load-balancing, and even a calculator.
This is a great start to get a taste of running P4 programs on a software target and quickly gave us the confidence to move on to the TNA architecture.

This rest of section 6 will showcase setups that test the functionality of forwarding, firewall features and performance of the program. All the tests are carried out on the Tofino switch located in the basement of building 340 at DTU - together with the two hosts; eoza and minza. The setup is illustrated in figure 14.

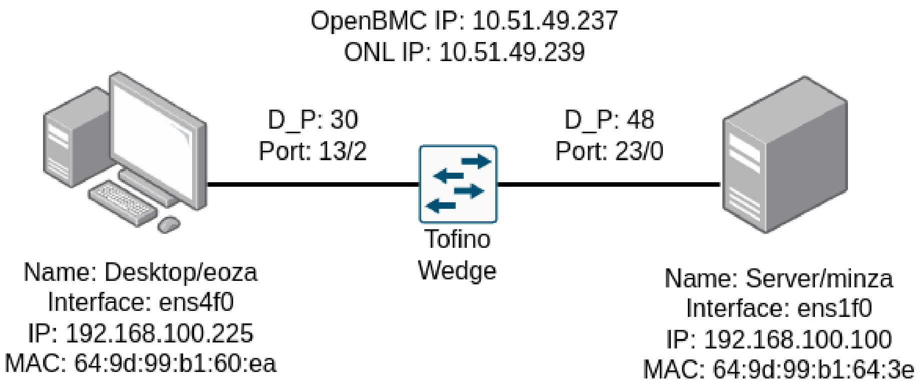
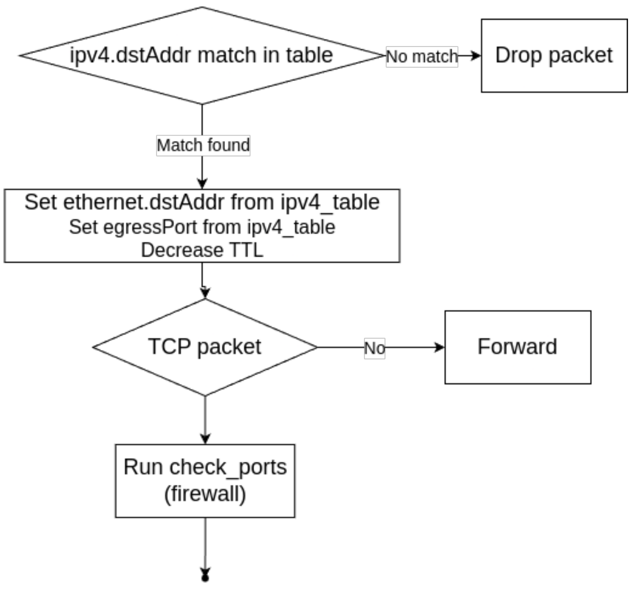


Figure 14: *Enviroment setup with Tofino switch and two hosts.*

6.2 Simple forwarding

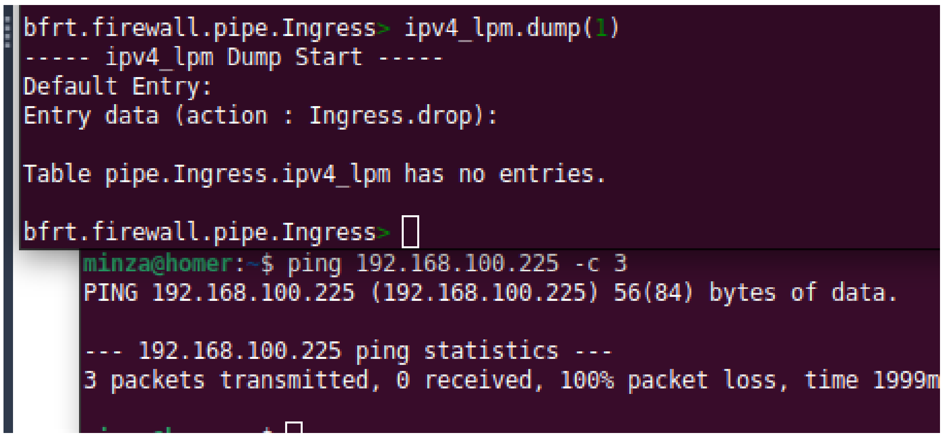
The simple forwarding happens to all non-TCP packets, as figure 15 illustrates.
The Internet Control Message Protocol (ICMP) Ping command will be used to test the forwarding. The P4 program checks if a destination Internet Protocol (IP) address is found in the Internet Protocol version 4 (IPv4) table. If found, the IPv4 header field Time To Live (TTL) is decremented, and the packet is forwarded to the respective ethernet destination address and corresponding egress port - as configured in the control plane.
This flow of simple forwarding is illustrated in figure 15.



ipv4.dstAddr match in table
No match
Drop packet
Match found
Set ethernet.dstAddr from ipv4_table
Set egressPort from ipv4_table
Decrease TTL
TCP packet
No
Forward
Run check_ports (firewall)

Figure 15: *Simple forwarding flow diagram*

To test this, only the IPv4 forwarding table has to be configured. Before configuring the ping test does not work:



```
bfrt.firewall.pipe.Ingress> ipv4_lpm.dump(1)
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

Table pipe.Ingress.ipv4_lpm has no entries.

bfrt.firewall.pipe.Ingress>
```

```
minza@homer:~$ ping 192.168.100.225 -c 3
PING 192.168.100.225 (192.168.100.225) 56(84) bytes of data.

--- 192.168.100.225 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 1999m
```

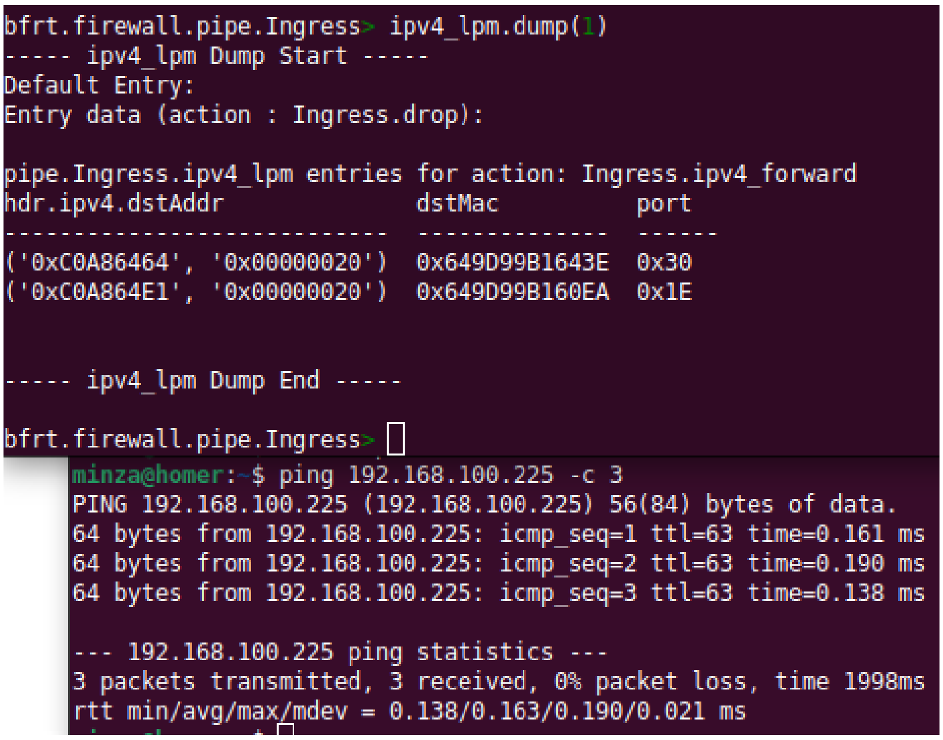
Figure 16: *No ping connection*

After adding the two clients to the table, the ping is successful. A guide on how to add the two clients, is found in appendix C. The forwarding table looks like so:

Key	dst MAC	dst Port
192.168.100.100	64:9d:99:b1:64:3e	48
192.168.100.225	64:9d:99:b1:60:ea	30

Table 1: *Forwarding table for simple forwarding*

Figure 17 shows the Longest Prefix Match (LPM) table configuration as well as the ping result.



```
bfrt.firewall.pipe.Ingress> ipv4_lpm.dump(1)
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

pipe.Ingress.ipv4_lpm entries for action: Ingress.ipv4_forward
hdr.ipv4.dstAddr               dstMac          port
----------------------------   --------------  ------
('0xC0A86464', '0x00000020')   0x649D99B1643E  0x30
('0xC0A864E1', '0x00000020')   0x649D99B160EA  0x1E


----- ipv4_lpm Dump End -----

bfrt.firewall.pipe.Ingress>
```

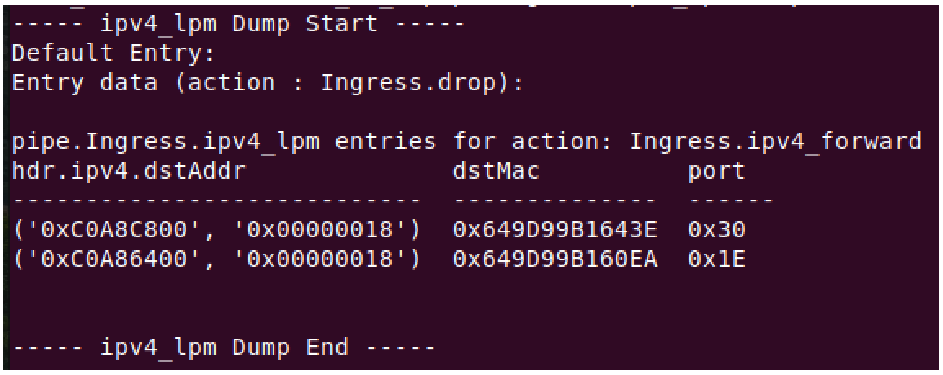
```
minza@homer:~$ ping 192.168.100.225 -c 3
PING 192.168.100.225 (192.168.100.225) 56(84) bytes of data.
64 bytes from 192.168.100.225: icmp_seq=1 ttl=63 time=0.161 ms
64 bytes from 192.168.100.225: icmp_seq=2 ttl=63 time=0.190 ms
64 bytes from 192.168.100.225: icmp_seq=3 ttl=63 time=0.138 ms

--- 192.168.100.225 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.138/0.163/0.190/0.021 ms
```

Figure 17: *Ping success*

This is simple forwarding of ICPM (ping) packets, works as expected. The user guide in Appendix C shows how to configure the control plane through Barefoot Runtime.

6.3 IPv4 subnet forwarding

In this test, the ability of the switch to forward packets to the correct IP subnet will be demonstrated. For this purpose, client minza will use IP 192.168.200.100, while eoza will use 192.168.100.225. Two different subnets, 192.168.200.0/24 and 192.168.100.0/24.
Figure 18 shows the following in a hexadecimal representation; IP target address, address match length, destination MAC address, destination port.

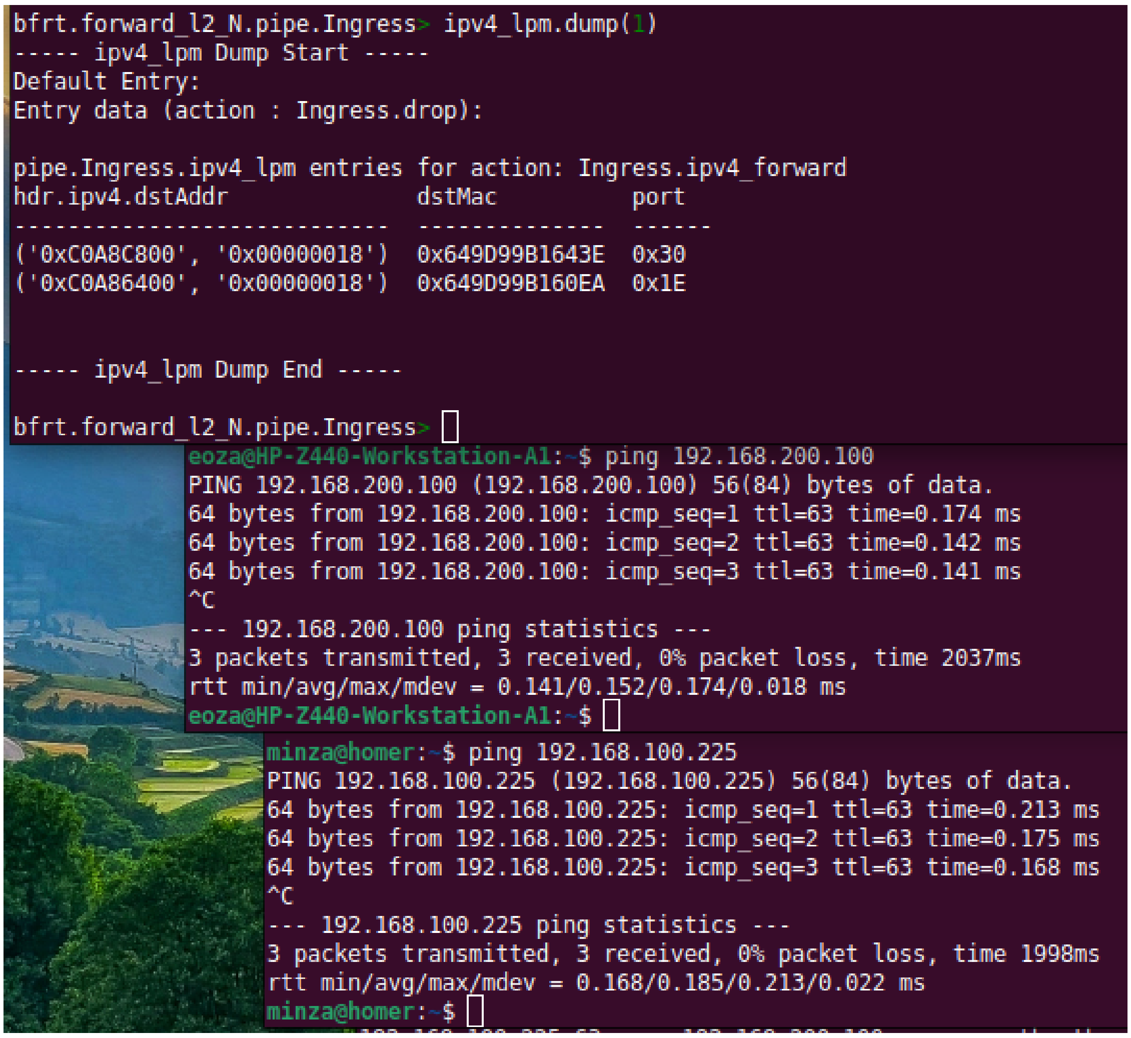


```
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

pipe.Ingress.ipv4_lpm entries for action: Ingress.ipv4_forward
hdr.ipv4.dstAddr               dstMac          port
----------------------------   --------------  ------
('0xC0A8C800', '0x00000018')   0x649D99B1643E  0x30
('0xC0A86400', '0x00000018')   0x649D99B160EA  0x1E


----- ipv4_lpm Dump End -----
```

Figure 18: *Dump displaying the /24 LPM setting.*

The dump shows how packets for 192.168.200.0/24 will be forwarded to port 48, while packets

for 192.168.100.0/24 will be forwarded to the client on port 30. To test the connection, a simple ping request will be sent, showcasing the forwarding capability of the Tofino switch and the compiled P4 program.
The ping is successful, as the screenshot in figure 19 illustrates:



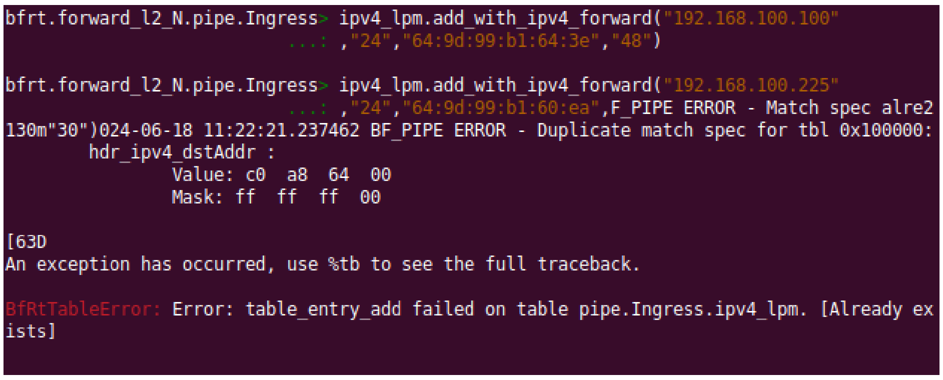
```
bfrt.forward_l2_N.pipe.Ingress> ipv4_lpm.dump(1)
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

pipe.Ingress.ipv4_lpm entries for action: Ingress.ipv4_forward
hdr.ipv4.dstAddr              dstMac          port
----------------------------  --------------  ------
('0xC0A8C800', '0x00000018')  0x649D99B1643E  0x30
('0xC0A86400', '0x00000018')  0x649D99B160EA  0x1E


----- ipv4_lpm Dump End -----

bfrt.forward_l2_N.pipe.Ingress>
```

```
eoza@HP-Z440-Workstation-A1:~$ ping 192.168.200.100
PING 192.168.200.100 (192.168.200.100) 56(84) bytes of data.
64 bytes from 192.168.200.100: icmp_seq=1 ttl=63 time=0.174 ms
64 bytes from 192.168.200.100: icmp_seq=2 ttl=63 time=0.142 ms
64 bytes from 192.168.200.100: icmp_seq=3 ttl=63 time=0.141 ms
^C
--- 192.168.200.100 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.141/0.152/0.174/0.018 ms
eoza@HP-Z440-Workstation-A1:~$
```

```
minza@homer:~$ ping 192.168.100.225
PING 192.168.100.225 (192.168.100.225) 56(84) bytes of data.
64 bytes from 192.168.100.225: icmp_seq=1 ttl=63 time=0.213 ms
64 bytes from 192.168.100.225: icmp_seq=2 ttl=63 time=0.175 ms
64 bytes from 192.168.100.225: icmp_seq=3 ttl=63 time=0.168 ms
^C
--- 192.168.100.225 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.168/0.185/0.213/0.022 ms
minza@homer:~$
```

Figure 19: *Successful ping between clients.*

The Barefoot SDE does not allow multiple addresses under the same subnet, ie. it is not possible to add 192.168.100.100 as well as 192.168.100.225, with the IP address length of 24, as they are part of the same subnet. This results in an error:



```
bfrt.forward_l2_N.pipe.Ingress> ipv4_lpm.add_with_ipv4_forward("192.168.100.100"
                           ...: ,"24","64:9d:99:b1:64:3e","48")

bfrt.forward_l2_N.pipe.Ingress> ipv4_lpm.add_with_ipv4_forward("192.168.100.225"
                           ...: ,"24","64:9d:99:b1:60:ea",F_PIPE ERROR - Match spec alre2
130m"30")024-06-18 11:22:21.237462 BF_PIPE ERROR - Duplicate match spec for tbl 0x100000:
        hdr_ipv4_dstAddr :
                Value: c0  a8  64  00
                Mask: ff  ff  ff  00

[63D
An exception has occurred, use %tb to see the full traceback.

BfRtTableError: Error: table_entry_add failed on table pipe.Ingress.ipv4_lpm. [Already ex
ists]
```

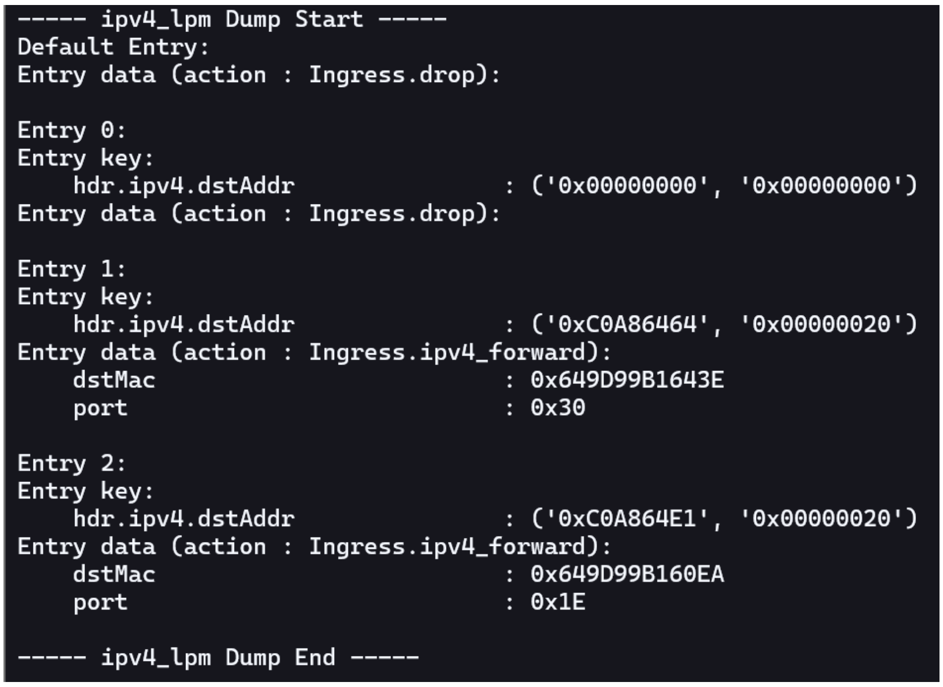
Figure 20: *Error adding rule.*

Thanks to the clever integration of checking the subnet of the entries in the table, there is no confusion and/or redundant entries in the `ipv4_lpm` table.

6.4 IPv4 LPM

LPM (Longest Prefix Matching) is used in IP forwarding to make sure the most specific rule is used. This allows one to have general rules for large subnets, without having them overrule the more specific rules for smaller subnets or exact matches.
For example, one might add a rule for the subnet 0.0.0.0/0, which is the entire IPv4 address space. This would act as a *default* rule, as it matches with every address that doesn't already have a more specific rule available to it. It also has the lowest priority in an LPM system, as the prefix is 0, and therefore overruled by any other rule with a longer prefix.

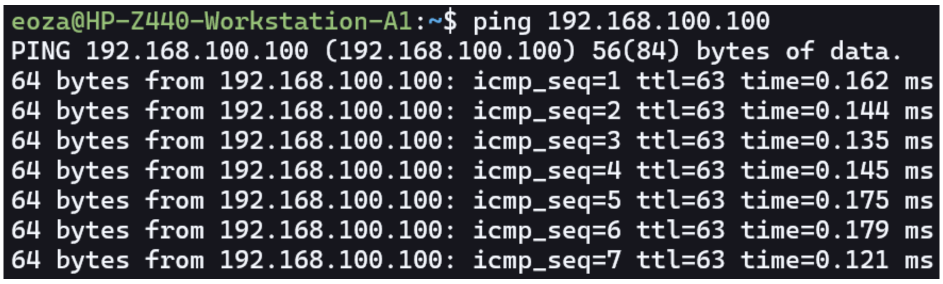
In this test, we've added a default rule to drop any packet matching 0.0.0.0/0.
Additionally, we've added forwarding rules for the addresses 192.168.100.100/32 (to minza) and 192.168.100.225/32 (to eoza).



```
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

Entry 0:
Entry key:
    hdr.ipv4.dstAddr              : ('0x00000000', '0x00000000')
Entry data (action : Ingress.drop):

Entry 1:
Entry key:
    hdr.ipv4.dstAddr              : ('0xC0A86464', '0x00000020')
Entry data (action : Ingress.ipv4_forward):
    dstMac                        : 0x649D99B1643E
    port                          : 0x30

Entry 2:
Entry key:
    hdr.ipv4.dstAddr              : ('0xC0A864E1', '0x00000020')
Entry data (action : Ingress.ipv4_forward):
    dstMac                        : 0x649D99B160EA
    port                          : 0x1E

----- ipv4_lpm Dump End -----
```

Figure 21: *Forwarding table of the switch.*

A packet addressed to 192.168.100.100 would trigger both entry 0 and entry 1, but since entry 1 has the longest prefix, that one will be used.



```
eoza@HP-Z440-Workstation-A1:~$ ping 192.168.100.100
PING 192.168.100.100 (192.168.100.100) 56(84) bytes of data.
64 bytes from 192.168.100.100: icmp_seq=1 ttl=63 time=0.162 ms
64 bytes from 192.168.100.100: icmp_seq=2 ttl=63 time=0.144 ms
64 bytes from 192.168.100.100: icmp_seq=3 ttl=63 time=0.135 ms
64 bytes from 192.168.100.100: icmp_seq=4 ttl=63 time=0.145 ms
64 bytes from 192.168.100.100: icmp_seq=5 ttl=63 time=0.175 ms
64 bytes from 192.168.100.100: icmp_seq=6 ttl=63 time=0.179 ms
64 bytes from 192.168.100.100: icmp_seq=7 ttl=63 time=0.121 ms
```

Figure 22: *Successful ping from eoza to minza.*

The ability to successfully ping between the two hosts shows that the 0.0.0.0/0 default rule is not triggered, despite being the first entry in the forwarding table.

6.5 Firewall functionality

We have implemented a firewall intended to block all TCP traffic that isn't part of an already-established exchange initiated by a client behind the firewall.
Unlike the IPv4 table, the `check_ports` table has two keys - one for verifying ingress port and one for verifying egress port.
The current setup uses the `range` match type, meaning the ingress and egress port is configured in a range of ports - not an exact match or lpm match. Furthermore, when configuring the

table, a direction for the match is set to either 0 or 1. 0 indicating it is from the internal network and 1 being outside the network.
The flow of any TCP packet is illustrated in figure 23. DOT graphs on the Ingress parser, Ingress and Ingress power, generated by Barefoot is found in appendix F. These illustrates how the firwall.p4 is handled by the different components of the program.
The default setup is to drop any packets that dont match a rule in the table.

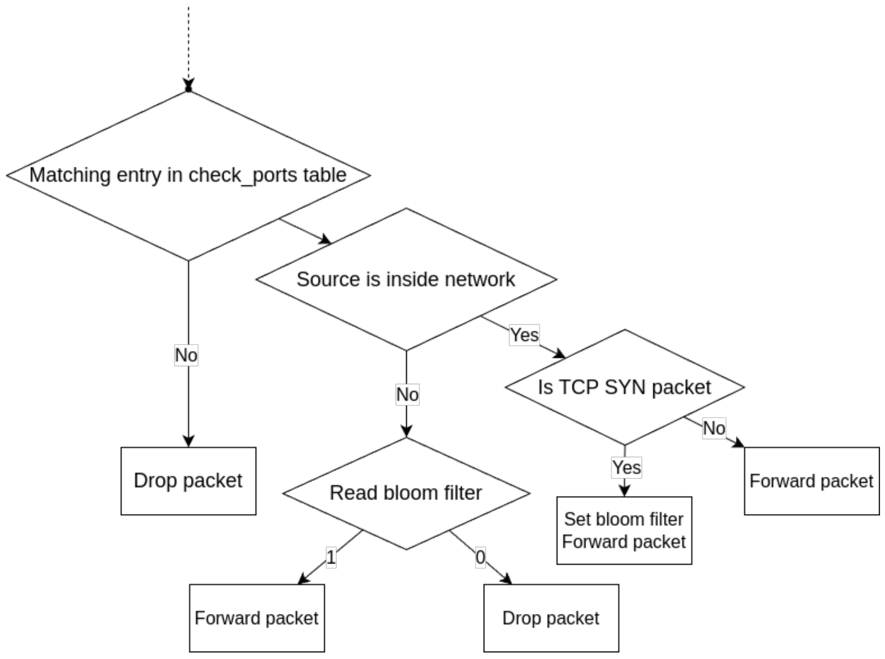


Figure 23: *TCP forwarding*

For the two clients, eoza and minza to be configured inside the network, the complete table for TCP rules is shown in table 2. The table below shows the configuration of the table, where the asterisk is a wildcard for any port and a lower priority is prioritized higher.

Table 2: *check_ports configuration table*

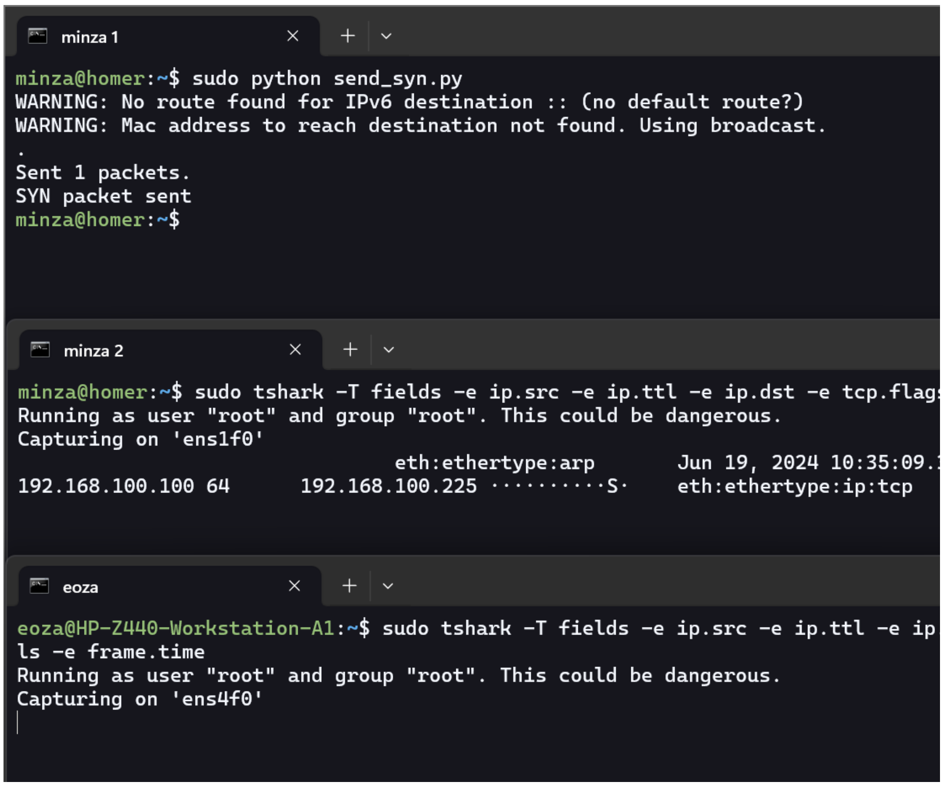
Rule #	Ingress port	Egress port	Direction	Priority
1	30	*	0	0
2	48	*	0	0
3	*	*	1	1

6.5.1 Firewall test 1 - external initialization

In this test, we've set up eoza to be inside the area protected by the firewall, while minza is outside of it. This equates to only rule #1 and #3 being added to the table - allowing traffic to and from port 30 if TCP handshake is initialized from inside the network.

To generate TCP SYN packets, the Python library Scapy is handy. To monitor network activity on the respective network interfaces TShark is the network analyzer of choice. Tshark shows which packets are forwarded and implicitly which are blocked.

The first test has minza (outside network), send a SYN packet to eoza (inside).



```
minza@homer:~$ sudo python send_syn.py
WARNING: No route found for IPv6 destination :: (no default route?)
WARNING: Mac address to reach destination not found. Using broadcast.
.
Sent 1 packets.
SYN packet sent
minza@homer:~$
```

```
minza@homer:~$ sudo tshark -T fields -e ip.src -e ip.ttl -e ip.dst -e tcp.flag
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens1f0'
                                eth:ethertype:arp       Jun 19, 2024 10:35:09.
192.168.100.100 64      192.168.100.225 ··········S·     eth:ethertype:ip:tcp
```

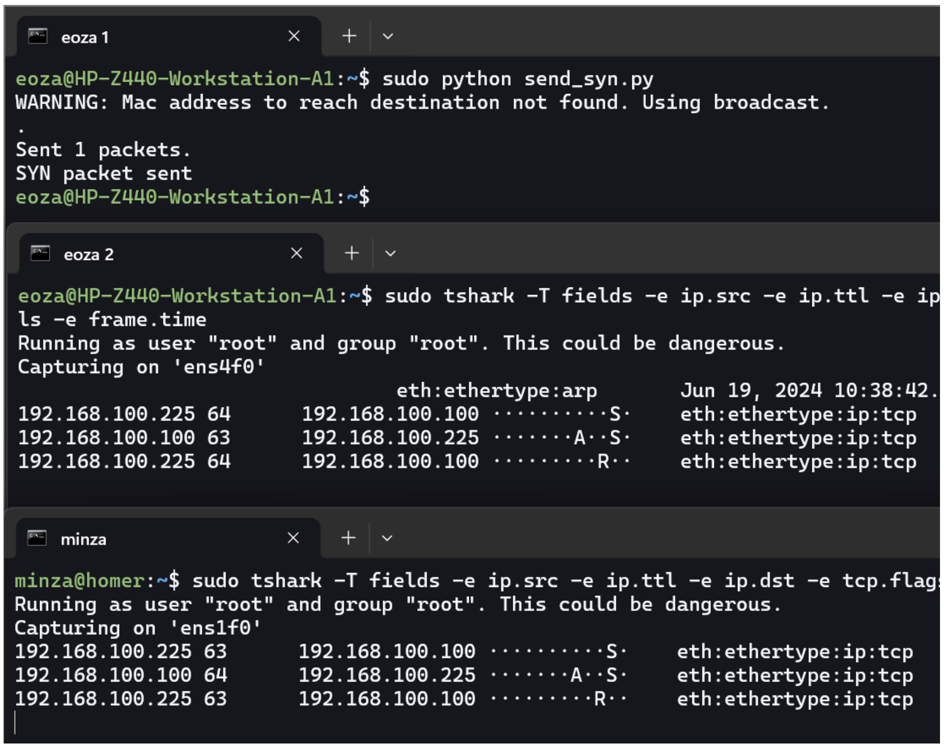
```
eoza@HP-Z440-Workstation-A1:~$ sudo tshark -T fields -e ip.src -e ip.ttl -e ip
ls -e frame.time
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens4f0'
```

Figure 24: *TCP SYN from minza to eoza.*

As can be seen on the two TShark outputs, the interface on minza sees the sent SYN, but no response. Meanwhile eoza does not capture anything as the switch has dropped the packet, as expected.

6.5.2 Firewall test 2 - internal initialization

This tests sends a SYN from eoza to minza - from inside, to outside the network.



```
eoza 1
eoza@HP-Z440-Workstation-A1:~$ sudo python send_syn.py
WARNING: Mac address to reach destination not found. Using broadcast.
.
Sent 1 packets.
SYN packet sent
eoza@HP-Z440-Workstation-A1:~$
```

```
eoza 2
eoza@HP-Z440-Workstation-A1:~$ sudo tshark -T fields -e ip.src -e ip.ttl -e ip
ls -e frame.time
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens4f0'
                                eth:ethertype:arp      Jun 19, 2024 10:38:42.
192.168.100.225 64      192.168.100.100 ··········S·   eth:ethertype:ip:tcp
192.168.100.100 63      192.168.100.225 ·······A··S·   eth:ethertype:ip:tcp
192.168.100.225 64      192.168.100.100 ·········R··   eth:ethertype:ip:tcp
```

```
minza
minza@homer:~$ sudo tshark -T fields -e ip.src -e ip.ttl -e ip.dst -e tcp.flag
Running as user "root" and group "root". This could be dangerous.
Capturing on 'ens1f0'
192.168.100.225 63      192.168.100.100 ··········S·   eth:ethertype:ip:tcp
192.168.100.100 64      192.168.100.225 ·······A··S·   eth:ethertype:ip:tcp
192.168.100.225 63      192.168.100.100 ·········R··   eth:ethertype:ip:tcp
```

Figure 25: *TCP SYN from eoza to minza.*

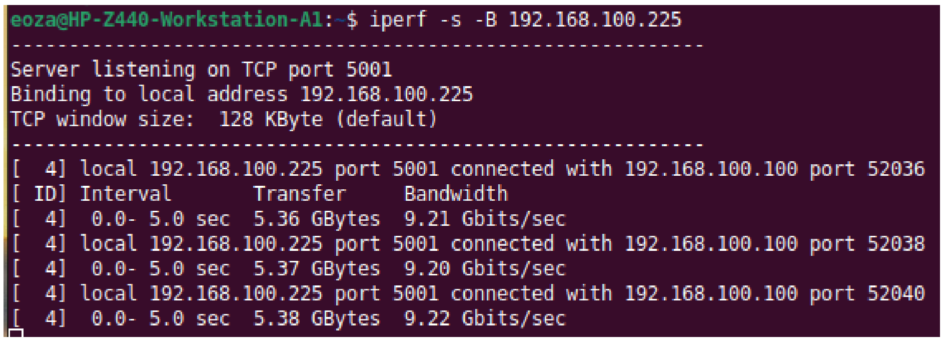
Here we see a different story. The entire exchange of SYN, SYN-ACK and RST is visible to both parties. For the SYN-ACK to go through from minza to eoza, the details of that TCP exchange must have successfully been added to the Bloom filter by the initial SYN sent from eoza. Furthermore, the TTL decrements as expected from 64 to 63, when received on either end.

The forwarding, bloom filter and firewall works as expected, only allowing a TCP handshake to be initialized from inside the network.

6.6 Performance

To test the performance of the firewall P4 program, we will compare it to the simple layer 2 forwarding with no firewall capabilities. The network bandwidth test of choice is the iperf test, measuring the throughput of the network.

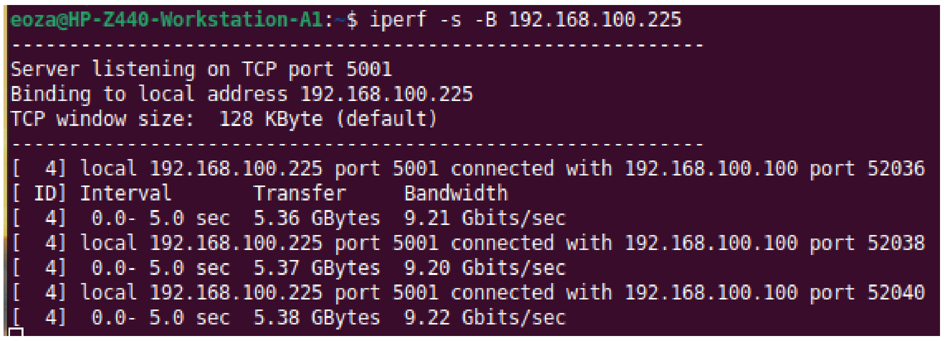
Running the iperf test with the firewall program, results in an averate bandwidth of 9.21 Gbits/sec, documented in figure 26(eoza as server, minza as client). The link is naturally limited to max 10Gbits/sec, due to the ethernet connection.



```
eoza@HP-Z440-Workstation-A1:~$ iperf -s -B 192.168.100.225
------------------------------------------------------------
Server listening on TCP port 5001
Binding to local address 192.168.100.225
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52036
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 5.0 sec  5.36 GBytes  9.21 Gbits/sec
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52038
[  4]  0.0- 5.0 sec  5.37 GBytes  9.20 Gbits/sec
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52040
[  4]  0.0- 5.0 sec  5.38 GBytes  9.22 Gbits/sec
```

Figure 26: *iperf Firwall test*

Running the same command on the `forward_l2.p4` program, results in a very similar average bandwidth of 9.17 Gbits/sec, within the margin of error.



```
eoza@HP-Z440-Workstation-A1:~$ iperf -s -B 192.168.100.225
------------------------------------------------------------
Server listening on TCP port 5001
Binding to local address 192.168.100.225
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52036
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 5.0 sec  5.36 GBytes  9.21 Gbits/sec
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52038
[  4]  0.0- 5.0 sec  5.37 GBytes  9.20 Gbits/sec
[  4] local 192.168.100.225 port 5001 connected with 192.168.100.100 port 52040
[  4]  0.0- 5.0 sec  5.38 GBytes  9.22 Gbits/sec
```

Figure 27: *iperf Forward_l2 test*

The slightly more complicated firwall program does not affect the iperf bandwidth performance test. iperf uses TCP ACK's as well as some TCP data packets to test the performance - not triggering hash calculations. A more interresting test would be to continuously send SYN packets that would trigger a hash calculations upon every SYN packet and measure the bandwidth. This would test if the hash calculations impact performance in a stress test.
nping is capable of testing this, but it did not show any significant change in the average RTT (round-trip-time).
For eoza it changed from an average of 0.008ms, to an average of 0.009ms in the firewall. Minzas average kept put at 0.012ms. Complete nping results are documented in appendix G.

6.7 Utilization

Naturally the Tofino chip does not have unlimited resources. Upon compilation, a summary of the resources used is generated. For the firewall program, the overall number of bits in the Tagalong Collection are just 12.9%, while the overall PHV usage is at 9.28% - both distributed on 8, 16 and 32b containers. The Tofino chip has plenty of memory to spare, allowing lots of further development.
Tagalong collections are used for managing additional metadata about a packet while Packet

Header Vector (PHV) usage refers to the collection of header fields extracted from a packets header. Ie. the more header and metadata definitions, the greater the memory usage. Complete reports on the memory allocation are found under "logs" on the GitHub repo of the project[1].
In summary, the chip has plenty of memory to spare and is far from fully utilized. It would be interresting to get access to Barefoot P4 Insight, showcased in "Tech Field Day" in 2019, see [9]. P4 Insight does require a license DTU does not possess.

[1]`https://github.com/McQueen24/P4`

7 Conclusion

As seen throughout the report, the group has been working intensively on understanding the P4 programming language and how to use the Tofino router. The functionality and theory behind all features and technologies have been thoroughly examined by the group to give the reader a clear understanding of the different technologies used.
When examining the group's work, a wide array of different technologies has been covered and looked into. One of the goals of the introduction was to provide readers with an understanding of how to use the P4 programming language themselves. Further, the group has provided in the appendix a walkthrough guide on how to access and use the Tofino switch and the RARE/FreeRtr. These walkthroughs are meant for students for future work and to help clarify how these components can be used.

In the testing section, there is a wide range of tests being described. Firstly, the Tofino router was tested, and we see that the router can successfully receive and forward packets. Then we moved on to the firewall, where a wide range of tests was carried out. The first couple of tests were successful, and as seen, the group managed to solve the previous firewall problems. When assessing all these tests, we see that the Tofino router is working as intended and is fully operational. When accessing the P4 language, it is highly capable of managing advanced routing properties and is a good approach when implementing advanced routers. As for the user guides, they can be found in the appendix and have been developed in such a manner that they give new programmers a good starting point, hopefully helping them save some time.

In the project we also wanted validate the RARE/FreeRouter framework in practice, and deploy the P4-forwarding and -firewalling rules to two different P4-programmable dataplane targets using FreeRouter as the control-plane and RARE as a common interface for the targets. While RARE/FreeRouter integration with the BMv2-target was successful, the implementation on the BareFoot-Tofino target was not fully completed. So, a proof-of-concept for using RARE/FreeRouter as a standarized control-plane and communication API for different P4-targets, was not achieved.

7.1 Future work

For the future of this project, developers could go in multiple directions. One of the features that could be looked into is the firewall; though it is functioning, more features and security measures could be added. Further, a look into connecting to the GEANT network for extensive testing could be an option when testing the router fully. Finally, more perspective on the user guide could be added if more interest from the public arises.

Appendices

References

[1] *freerTr*. URL: `http://www.freertr.org/`.

[2] *GitHub-repo for OpenNIC shell guide*. URL: `https://github.com/esnet/open-nic-shell`.

[3] *GitHub-repo for P4 tutorials in test-enviroment*. URL: `https://github.com/p4lang/tutorials`.

[4] *GitHub-repo for P4-compiler*. URL: `https://github.com/p4lang/p4c`.

[5] *GitLab-repo for ESnet SmartNIC example*. URL: `https://github.com/esnet/esnet-smartnic-hw/tree/main/examples/p4_only`.

[6] *GitLab-repo for ESnet SmartNIC tutorial*. URL: `https://gitlab.com/d-r-r/release/esnet-smartnic-tutorial`.

[7] Intel. P4_{16} *Intel Tofino Native Architecture - Public Version*. 2021.

[8] Sukhveer Kaur, Krishan Kumar, and Naveen Aggarwal. *A review on P4-Programmable data planes: Architecture, research efforts, and future directions*. March 2021. URL: `https://www.sciencedirect.com/science/article/pii/S0140366421000487`.

[9] *Tech Field Day youtube video on P4 Insight*.

A - Work Distribution

The work distribution of the report can be seen in the table below. All parts of the report has been reviewed and edited by all members of the group, and the table is only meant to show the primary author of each sections.

Section	**Primary Author(s)/Contributor(s)**
Abstract	Rasmus
Revisions	Joachim
Introduction	Nikolai & Joachim
System Overview	Lasse
Functional Specification	Joachim
Theory	Nikolai
Implementation	Lasse
Testing and Validation	Rasmus & Nikolai
Conclusion	Joachim
P4 code Development	Rasmus
RARE/FreeRtr Implementation	Lasse
Tofino User Guide	Rasmus
Guide for RARE/FreeRtr Implementation	Lasse

B - Peer Feedback Valuable Points

In this section a presentation of the given feed back from the groups peers can be seen. Here each section in the feedback schema will be presented and underlined is the most significant and interesting points for each section.

1. Structure & formalities

Overall rating: 4.5/5

"The overall structure is very complete and clear, there are only a few small problems, the paragraphs are not indented, and some code can be centred if it would be clearer, such as the code snippet on page 10"

"Overall very good structure, the entire project is explained well and each part of the report correlates well with the others."

2. Readability & language

Overall rating: 4.5/5

"Overall (like mentioned in the previous comment), the flow of information given is very good, since all the main "building blocks" of the system are explained in depth. I know that this report is of very technical nature and there are a lot of intricacies, but as a suggestion maybe there could be a few simpler explaination of them, since perhaps someone who does not know how registries work might have some trouble understanding about what is actually going on. Else than that, the information given is very thorough, good job!"

"While the language is easy to follow, it is long in several places and repeats itself"

"The introduction section is to the long side and could maybe be cut down a bit and some of the project details get mentioned more than once. This isn't a problem but something to keep in mind if looking for things to improve."

3. Explain the project

"The project is very well explained and I especially like the "System Overview" section. I would love to hear a little bit more about why this subject was chosen."

4. Scoping

Overall rating: 4.6/5

"The scope is very clear and well-defined. The objectives section is a good idea, which I didn't see in other reports and helps to understand the goals of the report."

"This project is wide but the authors managed to scope it clearly. The theory explains how the system works and especially the P4 architecture, the bloom filter and the Tofino Architecture. It is clearly linked to what was introduced in the introduction. Maybe the Bloom filter could have been introduced in the introduction to understand a bit before how it is important in the overall project."

5. Theory

Overall rating: 4.7/5

"The theory is very well written. The choices in what to include and what not to include make the level quite high but it also manages to not make the theory section too long."

6. Validation of theory

Overall rating: 4.4/5

"Good reference and figures!"

"The theory is well explained but it seems like only one source was used in this part. Is the example of the match action from you or were you inspired by some literature?"

7. Choise of tools & platforms

Overall rating: 4.6/5

"All the things explained in the theory are used in the project. I am missing an explanation as to why you use FreeRtr-BMv2 in the implementation part"

"The choice of the used tools for the platform, the programming language and the switch have been well explained and justified by the authors."

8. Implementation

Overall rating: 4.4/5

"Regarding the implementation, I think that some parts could be added in an appendix instead of in the report (the code lines regarding the update of the system or the cloning from GitHub).
For the bloom filter, the figure is relevant and interesting."

9. Implementation quality

Overall rating: 4.5/5

"Looks very well implemented and all the steps in the process are well described."

"It is very well documented, commented and easy to follow."

10. Commenting implementation

"I'd recommend the authors to focus primarily on the implementation of the firewall functionality, especially addressing the issues encountered with the bloom filter."

"I would very much like to hear what challenges the project has brought so far."

11. Testing scenarios

Overall rating: 4.7/5

"A lot of tests have already been performed. They are well documented and we can easily understand the process of their testing especially with the firewalls testing. The results of the firewalls testing will be continued in the 3 weeks period."

"It's interesting that you start with Ubuntu to become familiar with P4 and quickly transition to BFRT/TNA for a more practical test. You have simulated most of the possible scenario (I suppose) from packet loss to the firewall configuration to the transmission time. However maybe you may like to propose the throughput of each device in the network? :)"

12. Ambition level & amount

Overall rating: 4.8/5

"This project is ambitious, the group seems to have put a lot of effort on the P4 programming. There are a lot of tests that have been performed."

13. Clear who did what

Overall rating: 4.6/5

"It would maybe be nice to split the theory and some of the other sections so every member of the group shows that they work on all the taxonomic levels and understand the theory and implementation."

14. Suggestions

"looking through the code and removing the unnecessary out commented code blocks might be a good idea to improve code readability."

"A more precise and to the point language throughout the report would make it easier to read. Details given as to why the firewall did not work would have helped"

"In conclusion, I like your report very much and I appreciate the added goals section for good clarity. I did find some small grammar mistakes and typos for which I suggest using a grammar checker after everyone finished their parts. I also felt like the report was a bit long, but in general most sections very justified for the report"

C User Guide for Tofino switch

This section contains a complete user guide on how to transfer a P4 program to the Tofino switch onsite (INCOM lab in building 340 at DTU), and compile and test the connection between the connected clients. Everything but passwords will be shared. The guide is based on an Ubuntu 22.04.4 LTS build, June 2024.
Optionally, the DTU VPN can be used, enabling work from home. Command for connecting to the DTU VPN:

```
sudo openconnect --useragent=AnyConnect vpn.dtu.dk
```

The VPN is not required when connected to eg. eduroam or DTUsecure.

C.1 Transferring files for compilation

The management of the server happens via. the management port of the switch. For this project we have used remote management. If a physical connection is preferred, use serial port COM5, download driver and set the baud rate to 9600 (8 data bits, no parity, 1 stop bit).
For remote management there are two options:

- Use SCP in terminal
- Use Filezilla to transfer files

To transfer files via. the SCP command, the file has to be directly placed in the root folder on the ONL (Open Network Linux) server.

```
scp <filename>.p4 root@10.51.49.239:/root/<filename>.p4
```

Alternatively, a more flexible and interresting option is to use FileZilla. FileZilla gives a quick overview of the files and structure of the BMC/ONL. Use host: 10.51.49.239, username: root, password: , port: 22.
Go to the root folder and simply drag and drop the relevant P4 file to root/.

C.2 Connecting to ONL

As mentioned previously, this project is based on ssh connections. To connect to the BMC, use the following command:

```
ssh root@10.51.49.237 -p 22
```

Password: `OpenBmc`
When connected to the BMC, run `sol.sh` to connect the ONL server.
Alternatively, if the server is running, connect directly to the ONL server via:

```
ssh root@10.51.49.239 -p 22
```

Password: `root`

C.3 Compiling a P4 program

When connected to `root@localhost`, compilation of the P4 program on the Tofino switch happens through a shell script using the `bf-p4c` compiler. To compile run:

```
root@localhost:~# ~/tools/p4_build.sh <filename>.p4
```

Compilation is successful when the script writes
`CLEAR CONFIGURE MAKE INSTALL ... DONE`
After a successfull compilation, the `switchd` daemon is started through another shell script. This is provided by the Barefoot SDE. The `switchd` daemon is responsible for initializing and programming the pipeline of the switch and will be used to configure forwarding tables and firewall configurations. To start the daemon, run:

```
root@localhost:~# cd $SDE && ./run_switchd.sh -p <filename>
```

Note, the switchd runs a `<filename>.conf` file, so avoid writing `<filename>.p4` when running the `switchd` script.
After starting the daemon, start the Unified Command Line Interface (UCLI) through the command `ucli` This is where the current setup is shown. Especially the Port Management (PM) is used, to monitor the active ports. After entering the `ucli`, write `pm` to show the port management.

```
bfshell> ucli
Starting UCLI from bf-shell
Cannot read termcap database;
using dumb terminal settings.
bf-sde> pm
bf-sde.pm> show
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |RDY|ADM|OPR|LPBK    |FRAMES RX       |FRAMES TX       |E
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
bf-sde.pm>
```

The code above shows an empty port configuration.

C.4 Configuring ports

Figure 14 shows eoza connected to the switch via physical port 13/2, while minza is connected to physical port 23/0. Both ports are added:

```
bf-sde.pm> port-add 13/2 10G NONE
bf-sde.pm> port-add 23/0 10G NONE
bf-sde.pm> show
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |RDY|ADM|OPR|LPBK    |FRAMES RX       |FRAMES TX       |E
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
13/2 |11/2| 30|1/30|10G    |NONE|YES|DIS|DWN|  NONE  |               0|               0|
23/0 | 1/0| 48|1/48|10G    |NONE|YES|DIS|DWN|  NONE  |               0|               0|
```

After adding the ports, the Auto Negotiation (AN) has to be set to option 2 = disabled. As the `OPR` column shows in the previous `show`, both ports are disabled. To set `an` and enable ports, run:

```
bf-sde.pm> an-set -/- 2
bf-sde.pm> port-enb -/-
bf-sde.pm> show
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |RDY|ADM|OPR|LPBK    |FRAMES RX       |FRAMES TX       |E
-----+----+---+----+-------+----+---+---+---+--------+----------------+----------------+-
13/2 |11/2| 30|1/30|10G    |NONE|YES|ENB|DWN|  NONE  |               0|               0|
23/0 | 1/0| 48|1/48|10G    |NONE|YES|ENB|DWN|  NONE  |               0|               0|
```

The `-/-` means any port and simplifies the process of writing `an-set 13/2 2`, `an-set 23/0 2`... To further simplify the setup, simply write the following. Beware this adds and enables all the ports on the switch.

```
bf-sde.pm> port-add -/- 10G NONE
bf-sde.pm> an-set -/- 2
bf-sde.pm> port-enb -/-
```

C.5 Entering Barefoot API and configuring tables

The Barefoot API is the control plane of the program and controls the forwarding as well as firewall rules through tables.
To enter the API, exit the UCLI and run `bfrt_python`.

```
bf-sde.pm> exit
bfshell> bfrt_python
cwd : /root/bf-sde-9.2.0

We've found 1 p4 programs:
<filename>
```

```
Loading the tables ...
```

From here, the setup depends on the P4 program, but in this case LPM forwarding to the two clients will be configured through the IPv4 LPM table. This table is part of the `Ingress` section, encapsulated in a final package called `pipe`.
The program is called `firewall.p4` and matches on the IPv4 address with a LPM match. In the example below, the IP match has to match all 32 bits.
If a match is found, the packet is forwarded to the MAC address on the respective port. Eg. if the destination IP is 192.168.100.100, forward to MAC address ...64:3e on port 48.
This is layer 2 forwarding, based on layer 3 (IP) headers.

```
bfrt_root>bfrt.firewall.pipe.Ingress
--------> bfrt.firewall.pipe.Ingress()
Available symbols:
bloom_filter_1          - Table Object
bloom_filter_2          - Table Object
check_ports             - Table Object
dump                    - Command
hash_10                 - Command Object
hash_11                 - Command Object
hash_20                 - Command Object
hash_21                 - Command Object
info                    - Command
ipv4_lpm                - Table Object


bfrt.firewall.pipe.Ingress> ipv4_lpm.add_with_ipv4_forward("192.168.100.100","32
                       ...: ","64:9d:99:b1:64:3e","48")

bfrt.firewall.pipe.Ingress> ipv4_lpm.add_with_ipv4_forward("192.168.100.225","32
                       ...: ","64:9d:99:b1:60:ea","30")

bfrt.firewall.pipe.Ingress> ipv4_lpm.dump(table=1)
----- ipv4_lpm Dump Start -----
Default Entry:
Entry data (action : Ingress.drop):

pipe.Ingress.ipv4_lpm entries for action: Ingress.ipv4_forward
hdr.ipv4.dstAddr              dstMac          port
----------------------------  --------------  ------
('0xC0A86464', '0x00000020')  0x649D99B1643E  0x30
('0xC0A864E1', '0x00000020')  0x649D99B160EA  0x1E
```

```
----- ipv4_lpm Dump End -----
```

At this point simple forwarding based on IP addresses is implemented and is ready for testing.
To clear the entire table, write `ipv4_lpm.clear`.
To modify an entry, write eg. `ipv4_lpm.mod_with_ipv4_forward("192.168.100.100","32 ...:  ","64:9d:99:b1:64:3e","49")` to change the port from "48" to "49", without clearing the entire table.

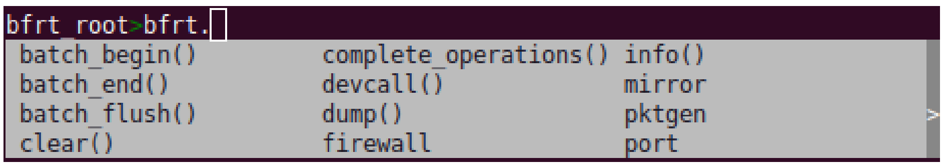
C.5.1 Tip

A helpfull tip is to use questionmark "?" after a function/command/table/method eg. `ipv4_lpm.add_with_ipv4_forward?`. This displays all the options for the requested table, with the default option listed.

```
bfrt.firewall.pipe.Ingress> ipv4_lpm.add_with_ipv4_forward?
Signature: ipv4_lpm.add_with_ipv4_forward(dstaddr=None, dstaddr_p_length=None,
↪  dstmac=None, port=None, pipe=None, gress_dir=None, prsr_id=None)
Docstring:
Add entry to ipv4_lpm table with action: Ingress.ipv4_forward

Parameters:
['dstaddr', 'dstaddr_p_length'] type=LPM        size=32 default=0
dstmac                         type=BYTE_STREAM size=48 default=0
port                           type=BYTE_STREAM size=9  default=0
File:      Dynamically generated function. No source code available.
Type:      method
```

C.5.2 Tip 2

After writing eg. `bfrt.`, press the TAB key do display all options for the next part of the command.
Writing `bfrt.` will in this case provide the option of selecting `firewall` as the next part of the command, shown in figure 28.



```
bfrt_root>bfrt.
 batch_begin()          complete_operations() info()
 batch_end()            devcall()             mirror
 batch_flush()          dump()                pktgen
 clear()                firewall              port
```

Figure 28: *Barefoot help menu*

C.6 Configuring a range-match key

Besides the ipv4_lpm table, the firewall program has a table called check_ports designed for allowing specific traffic to/from the correct ports. This uses the range-match key, defining a range of ports both in the source and destination.
For configuring a range-match key the start and end range has to be defined as well as a match priority (5th argument). 0 is the highest priority.

```
bfrt.firewall.pipe.Ingress> check_ports.add_with_set_direction("30","30","0","51
                       ...: 1","0","0")

bfrt.firewall.pipe.Ingress> check_ports.add_with_set_direction("0","511","30","3
                       ...: 0","1","1")

bfrt.firewall.pipe.Ingress> check_ports.dump(1)
----- check_ports Dump Start -----
Default Entry:
Entry data (action : NoAction):

pipe.Ingress.check_ports entries for action: Ingress.set_direction
ig_intr_md.ingress_port    ig_tm_md.ucast_egress_port    $MATCH_PRIORITY  dir
-------------------------  ----------------------------  -----------------  -----
('0x1E', '0x1E')           ('0x00', '0x1FF')                            0  0x0
('0x00', '0x1FF')          ('0x1E', '0x1E')                             1  0x1



----- check_ports Dump End -----
```

The dir column indicates whether the traffic comes from outside the network or inside the network. 0 is inside. Ie. traffic from port 30 is always allowed and prioritized as the dir = 0 and $MATCH_PRIORIRY = 0.
If the source port is not port 30, but targeted to port 30, dir = 1 and a connection has to be initialized from port 30 before the packet is allowed. This is how the stateful firewall operates, see further explanation in previous sections.

C.7 Inspecting bloom filter entries

To inspect the entries in a Register, the dump function is used. To avoid dumping all entries, use the option .dump(print_zero=0). Furthermore, the register dump does not update the dump output before calling .operation_register_sync(). An example of how to update a register below:

```
bfrt.firewall.pipe.Ingress> bloom_filter_1.dump(print_zero=0)
----- bloom_filter_1 Dump Start -----
----- bloom_filter_1 Dump End -----


bfrt.firewall.pipe.Ingress> bloom_filter_1.operation_register_sync
-------------------------> bloom_filter_1.operation_register_sync()

operation complete - dev_id: 0, pipe_id: 65535, direction: 0, parser_id: 0
bfrt.firewall.pipe.Ingress> bloom_filter_1.dump(print_zero=0)
----- bloom_filter_1 Dump Start -----
Entry 1568:
Entry key:
    $REGISTER_INDEX                : 0x00000620
Entry data:
    Ingress.bloom_filter_1.f1      : [1, 0]

----- bloom_filter_1 Dump End -----


bfrt.firewall.pipe.Ingress>
```

Note how the dump does not update before calling `operation_register_sync()`.

C.8 Connecting to clients

Currently the two clients connected to the switch are called eoza and minza, and are both accessible through ssh. Connect to eoza:

```
ssh eoza@10.51.49.225
```

Connect to minza:

```
ssh minza@10.51.49.233
```

C.9 Configuring IP, routing tables and arp tables

To test whether the network interface has been configured properly, run `ifconfig` on either device and inspect whether the IPv4 address is as expected on the network interface. On minza, add the IP 192.168.100.100 to ens1f0:

```
minza@homer:~$ sudo ip addr add 192.168.100.100/24 dev ens1f0
```

When running ifconfig, the ens1f0 interface should display:

```
ens1f0    Link encap:Ethernet  HWaddr 64:9d:99:b1:64:3e
          inet addr:192.168.100.100  Bcast:0.0.0.0  Mask:255.255.255.0
```

To delete or reconfigure an interface, run `del` instead of `add`, eg. `sudo ip addr del 192.168.100.100/24 dev ens1f0` to delete the IP and subnet mask. This is handy when reconfiguring the interface.
After configuring the interface, run the `route -n` command to display the routing table. In order to communicate to the other client, the respective IP address should be known by the routing table. To add an entry to the routing table, eg. for minza to know about eoza, add:

```
sudo route add -host 192.168.100.225 metric 100 dev ens1f0
```

To map the IP address to the correct MAC address, run the following command on minza:

```
sudo arp -s 192.168.100.225 64:9d:99:b1:60:ea
```

To test the arp table, run `arp -a` and find the added IP in the list.
The network interface, routing table and arp table is now setup correctly and the two clients can ping each other, showed below:

```
minza@homer:~$ sudo ip addr add 192.168.100.100/24 dev ens1f0
minza@homer:~$ sudo route add -host 192.168.100.225 metric 100 dev ens1f0
minza@homer:~$ sudo arp -s 192.168.100.225 64:9d:99:b1:60:ea
minza@homer:~$ ping 192.168.100.225 -c 3
PING 192.168.100.225 (192.168.100.225) 56(84) bytes of data.
64 bytes from 192.168.100.225: icmp_seq=1 ttl=63 time=0.170 ms
64 bytes from 192.168.100.225: icmp_seq=2 ttl=63 time=0.171 ms
64 bytes from 192.168.100.225: icmp_seq=3 ttl=63 time=0.148 ms

--- 192.168.100.225 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.148/0.163/0.171/0.010 ms
```

Now make a mirrored setup eoza, eg. using figure 14.

C.9.1 Tip

The client minza seems to reset the interface, routing table and arp table very often. It might be necessary to run the three mentioned commands multiple times. This is not the case for client eoza.

C.10 Inspecting network traffic

To capture network traffic, the network protocol analyser, tshark, is used. Tshark is installed on both clients. For firewall testing we created the custom output with IP source, TTL, IP destination, TCP flags, frame protocol as well as date/time. During development, the TTL field is a convenient field to modify, to inspect how far into the P4 program, any packet reaches. This is the reason to include the TTL as one of the very first fields.

```
sudo tshark -T fields -e ip.src -e ip.ttl -e ip.dst -e tcp.flags.str -e
↪  frame.protocols -e frame.time
```

This is fully customizeable depending on the usecase. Wireshark was not an option due to the ssh connection in a terminal.
Eoza should inspect on interface ens4f0 while minza should inspect on ens1f0.

C.11 Firewall test packets using scapy

On either client it is possible to use the python library scapy and run python scripts.
To create a new script run and edit it through the nano text editor on the terminal, use:

```
sudo nano scriptname.py
```

To test the firewall a simple script called `send_syn.py`, sending a SYN packet, is created and sent. The script imports the scapy library and defines the destination IP and port:

```python
from scapy.all import *

# Define target IP and port
target_ip = "192.168.100.225"
target_port = 22

# Send SYN packet
syn_packet = IP(dst=target_ip)/TCP(dport=target_port, flags="S", seq=1000)
send(syn_packet)
print("SYN packet sent")
```

To run the script, use `sudo python send_syn.py`.

C.11.1 Tip

Open two terminals on each client, one on each running tshark and another one sending packets/doing network configuration.

C.12 Inspecting log files

The log files may be interresting to inspect, to view resource utilization etc.
The path for the log files are found in the SDE path:

```
/root/bf-sde-9.2.0/build/p4-build/<filename>/tofino/<filename>/pipe/
```

Here, DOT graphs of the program are available as well. They can eg. be rendered on an online platform like `https://dreampuf.github.io/GraphvizOnline`.

D Guide for RARE/FreeRtr Implementation

D.0.1 Installation of FreeRtr environment on Ubuntu v22.04

Sections D.0.1-D.0.3 provides documentation and guidance on how RARE and FreeRtr is installed on the Linux Ubuntu (64-bit) v22.04 VM, how it is integrated with the BMv2-dataplane, and how our P4 firewall program is deployed on the BMv2-switch.

1. Update Linux Ubuntu system:

   ```
   sudo apt-get update
   sudo apt-get upgrade
   ```

2. Install Java Runtime Environment, if missing on your system:

   ```
   sudo apt-get install default-jre-headless --no-install-recommends
   ```

   OBS!: FreeRtr shipped as the .jar-file was compiled using Java 11, thus Java 11 or later is required. Launching freeRtr environment with earlier version will give an "Unsupported Class Version Error".

3. Create FreeRtr directory environment:

   ```
   sudo mkdir -p ~/freeRouter/bin ~/freeRouter/lib ~/freeRouter/etc
    ↪  ~/freeRouter/log
   cd ~/freeRouter/lib
   sudo wget http://freerouter.nop.hu/rtr.jar
   ```

D.0.2 Deploy FreeRtr-instance with external connectivity

Documentation for connecting a deployed FreeRtr instance to the LAN of the VM-host. Requries an installed FreeRtr environment.

1. Install freeRouter net-tools:

   ```
   sudo wget http://www.freertr.net/rtr-`uname -m`.tar -O rtr.tar
   sudo tar xvf rtr.tar -C ~/freeRouter/bin/
   ```

2. Create hardware- & software- txt-files. They are used to configure the FreeRtr-instance. Store them in /freeRouter/etc. Contents of the configuration files can be found at Géant RARE/FreeRouter-101 Tutorial.

3. Launch FreeRtr environment (with txt-configuration files as parameters):

```
sudo java -jar lib/rtr.jar routersc etc/hw-config.txt
↪  etc/sw-config.txt
```

4. In `~/freeRouter/bin`, use pcapInt binary from freeRouter net-tools to bind a freeRouter socket to an available network hardware interface (enp0s8):

```
sudo ./pcapInt.bin enp0s8 2001 127.0.0.1 1001 127.0.0.1
```

   OBS!: pcapInt.bin requires glibc version 2.34 or later. The glibc vesion is pretty much fixed by the Linux Distribution/Version you have, and attempted upgrading could damage your system.

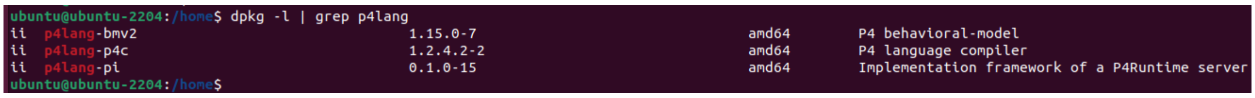
D.0.3 RARE/FreeRtr-BMv2 Integration on Ubuntu v22.04

Documentation for deploying a FreeRouter instance as a control-plane for the BMv2 virtual P4-switch.

1. Install P4 environment. Installing p4lang-p4c from the P4lang Project Repository, will install the P4Runtime API, the P4C compiler and BMv2 on your system:

```
echo
↪  'deb http://download.opensuse.org/repositories/home:/p4lang/xUbuntu_22.04/ /'
↪  | sudo tee /etc/apt/sources.list.d/home:p4lang.list
curl -fsSL https://download.opensuse.⌋
↪  org/repositories/home:p4lang/xUbuntu_22.04/Release.key | gpg
↪  --dearmor | sudo tee /etc/apt/trusted.gpg.d/home_p4lang.gpg >
↪  /dev/null
sudo apt update
sudo apt install p4lang-p4c
```

   `dpkg -l | grep p4lang` should verify the installation of the p4lang packages:



```
ubuntu@ubuntu-2204:/home$ dpkg -l | grep p4lang
ii  p4lang-bmv2                     1.15.0-7                          amd64        P4 behavioral-model
ii  p4lang-p4c                      1.2.4.2-2                         amd64        P4 language compiler
ii  p4lang-pi                       0.1.0-15                          amd64        Implementation framework of a P4Runtime server
ubuntu@ubuntu-2204:/home$
```

Figure 29: *Installed P4 packages.*

   For installing on Debian, commands can be found at Software Opensuse - p4lang-p4c Installation.

2. Clone RARE code from repository to your system:

    ```
    sudo git clone https://github.com/rare-freertr/RARE-bmv2.git
    ```

    Then go to `~/RARE-bmv2/02-PE-labs/p4src` and compile the RARE router.p4 (or any other chosen p4-program) with the P4C compiler:

    ```
    mkdir -p ../build ../run/log
    sudo p4c --std p4-16 --target bmv2 --arch v1model -I ./ -o
     ↪  ../build --p4runtime-files ../build/router.txt router.p4
    ```

3. Create hardware- & software- txt-files for the FreeRtr-instance and store them in /freeRouter/etc. Contents of the configuration files can be found at Géant RARE/FreeRouter-101 Tutorial.

4. The RARE/FreeRtr-BMv2 communication is enabled using a virtual Ethernet link:

    ```
    sudo ip link add veth251 type veth peer name veth250
    sudo ip link set veth250 up
    sudo ip link set veth251 up
    ```

5. Launch FreeRtr environment (with txt-configuration files as parameters):

    ```
    sudo java -jar lib/rtr.jar routersc etc/hw-config.txt
     ↪  etc/sw-config.txt
    ```

6. In `~/freeRouter/bin`, launch freeRouter pcapInt in order to stitch the control plane and P4 BMv2 dataplane communication:

    ```
    sudo ./pcapInt.bin veth251 22709 127.0.0.1 22710 127.0.0.1
    ```

7. Start the RARE P4-dataplane - the simple_switch_grpc version of BMv2. This will also start the gRPC runtime server on BMv2:

    ```
    export P4_RARE_ROOT=RARE-bmv2/02-PE-labs
    sudo simple_switch_grpc --log-file
     ↪  $P4_RARE_ROOT/run/log/p4-freerouter.log -i 1@enp0s8 -i
     ↪  64@veth250 --thrift-port 9090 --nanolog ipc://
     ↪  $P4_RARE_ROOT/run/bm-0-log.ipc --device-id 0
     ↪  $P4_RARE_ROOT/build/simple_switch_grpc.json --
     ↪  --grpc-server-addr 127.0.0.1:50051 >
     ↪  $P4_RARE_ROOT/run/log/p4-freerouter.out 2>&1 &
    ```

8. In `~/RARE-bmv2/02-PE-labs`, launch forwarder.p4 to enable the P4Runtime communication between the FreeRtr control-plane and the BMv2 P4-dataplane:

```
sudo python3 p4src/forwarder.py --p4info build/router.txt
 ↪  --bmv2-json build/router.json --p4runtime_address
 ↪  127.0.0.1:50051 --freerouter_address 127.0.0.1
 ↪  --freerouter_port 9080
```

D.0.4 RARE/FreeRouter Implementation on Barefoot-Tofino switch

Documentation for deploying a FreeRouter control-plane instance and RARE-API on Barefoot-Tofino switch, and integrating with the bf-switch dataplane. A guide can also be found here: Freerouter installation guide on WEDGE-100BF-32X. For remotely connecting to the Tofino-switch, refer to the User Guide for Tofino switch.

1. Install latest version of JDK on the Barefoot-Tofino to support FreeRouter (was done during the implementation). Set the JAVA environment variable in ~/.bashrc: (gets reset with each reboot):

```
export JAVA_HOME=/root/jdk-22.0.1
export PATH=$JAVA_HOME/bin:$PATH
```

2. Clone RARE software from GÉANT Bitbucket into root directory (was done during the implementation):

```
git clone https://bitbucket.software.geant.org/scm/rare/rare.git
```

3. Create FreeRtr environment, from root-directory (was done during the implementation):

```
mkdir -p ~/freeRouter/bin ~/freeRouter/lib ~/freeRouter/etc
 ↪  ~/freeRouter/log
cd ~/freeRouter/lib
sudo wget http://freerouter.nop.hu/rtr.jar
```

4. Create FreeRouter hardware- and software- configuration files, store in /freeRouter/etc. Refer to Freerouter installation guide on WEDGE-100BF-32X, freeRouter control-plane configuration for the files.

5. Compile the P4-program (from root-directory):

```
~/tools/p4_build.sh -I ~/rare/p4src/ -DHAVE_MPLS
↪  /path/to/p4-file/program.p4
```

6. Start the bf_switch dataplane:

```
cd $SDE
./run_switchd.sh -p firewall
```

7. Configure the ports of the bf_switchd dataplane. Refer to C.4 for guidance on port configuration.

8. Setup the communication channel to the bf_switch dataplane: (to be done after each reboot)

```
echo 1 > /proc/sys/net/ipv6/conf/all/disable_ipv6
echo 1 > /proc/sys/net/ipv6/conf/default/disable_ipv6

# set CPU_PORT UP, promiscuous mode and jumbo MTU
# (enp4s0f1 interfaces to the P4_CPU port)
ip link set enp4s0f1 up
ifconfig enp4s0f1 promisc
ip link set dev enp4s0f1 up mtu 8192

# Disable TCP offload
export TOE_OPTIONS=
↪  "rx tx sg tso ufo gso gro lro rxvlan txvlan rxhash"

for TOE_OPTION in $TOE_OPTIONS; do
    /sbin/ethtool --offload enp4s0f1 "$TOE_OPTION" off &>
     ↪  /dev/null
done
```

9. Launch freerouter instance (in freeRouter directory):

```
java -jar lib/rtr.jar routersc etc/freerouter-hw.txt
↪  etc/freerouter-sw.txt
```

10. Start RARE interface between freeRouter and P4 dataplane:

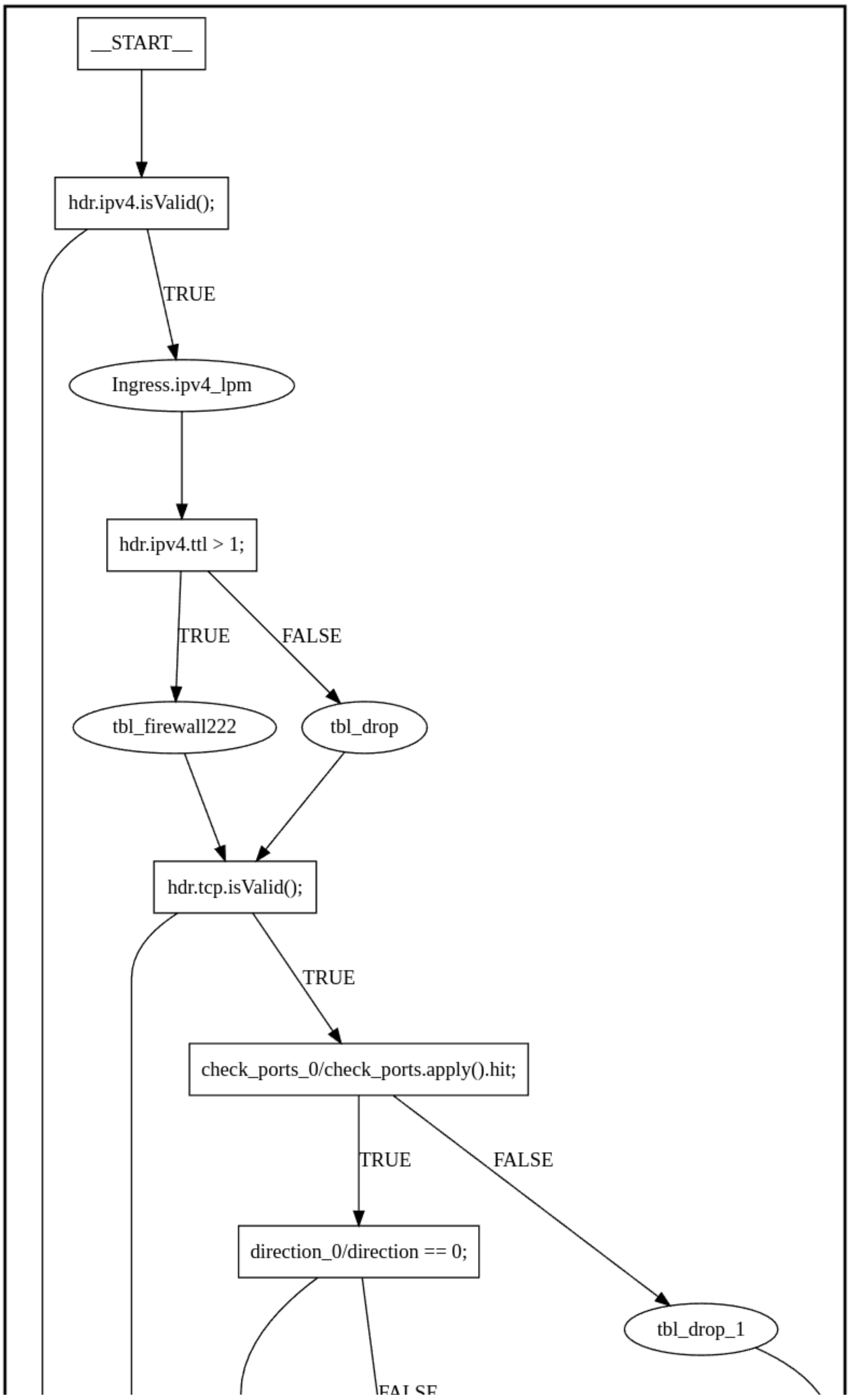
```
export PYTHONPATH='/usr/local/lib/python2.7/dist-packages'
cd /root/rare/bfrt_python
python bf_forwarder.py --p4-program-name firewall
```

E - firewall.p4

Link to public github repo (username McQueen24: Rasmus):
https://github.com/McQueen24/P4

```
/* -*- P4_16 -*- */

#include <core.p4>
#include <tna.p4>

/*************************************************************************
 ************* C O N S T A N T S    A N D   T Y P E S  *******************
*************************************************************************/
const bit<16> ETHERTYPE_TPID = 0x8100;
const bit<16> ETHERTYPE_IPV4 = 0x0800;
const bit<8>  TYPE_TCP = 6;

/* Table Sizes */
const int IPV4_HOST_SIZE = 65536;
const int IPV4_LPM_SIZE  = 12288;

#define BLOOM_FILTER_ENTRIES 4096

//typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

struct user_metadata_t {
    bit<1> bf_tmp;
}

/*************************************************************************
 ***********************  H E A D E R S  *********************************
 *************************************************************************/

header ethernet_h {
    bit<48>   dstAddr;
    bit<48>   srcAddr;
    bit<16>   ether_type;
}

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;
}

header tcp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<32> seqNo;
    bit<32> ackNo;
    bit<4>  dataOffset;
    bit<4>  res;
    bit<1>  cwr;
    bit<1>  ece;
    bit<1>  urg;
    bit<1>  ack;
    bit<1>  psh;
    bit<1>  rst;
    bit<1>  syn;
    bit<1>  fin;
    bit<16> window;
    bit<16> checksum;
    bit<16> urgentPtr;
}

/*************************************************************************
 **************  I N G R E S S   P R O C E S S I N G   *******************
 *************************************************************************/

    /***********************  H E A D E R S  ************************/

struct my_ingress_headers_t {
    ethernet_h   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;

}

    /****** G L O B A L   I N G R E S S   M E T A D A T A  *********/

struct my_ingress_metadata_t {
    user_metadata_t md;
    bit<1> bloom_read_1;
    bit<1> bloom_read_2;
}

    /*********************** P A R S E R  *************************/
parser IngressParser(packet_in        pkt,
    /* User */
    out my_ingress_headers_t          hdr,
    out my_ingress_metadata_t         meta,
    /* Intrinsic */
    out ingress_intrinsic_metadata_t  ig_intr_md)
{
    Checksum() ipv4_checksum;
    Checksum() tcp_checksum;

    /* This is a mandatory state, required by Tofino Architecture */
     state start {
        pkt.extract(ig_intr_md);
        pkt.advance(PORT_METADATA_SIZE);
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.ether_type) {
            ETHERTYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            TYPE_TCP: parse_tcp;
            default: accept;
        }
    }

    state parse_tcp {
        pkt.extract(hdr.tcp);
        transition accept;
    }
}

    /***************** M A T C H - A C T I O N  ********************/

control Ingress(
    /* User */
    inout my_ingress_headers_t                       hdr,
    inout my_ingress_metadata_t                      meta,
    /* Intrinsic */
    in    ingress_intrinsic_metadata_t               ig_intr_md,
    in    ingress_intrinsic_metadata_from_parser_t   ig_prsr_md,
    inout ingress_intrinsic_metadata_for_deparser_t  ig_dprsr_md,
    inout ingress_intrinsic_metadata_for_tm_t        ig_tm_md)
{

    // define two registers to hold the bloom filters
    Register<bit<1>,_>(BLOOM_FILTER_ENTRIES) bloom_filter_1;
    ↪  // [format of individual entries], [index size (DONTCARE)], [number of entries], [na
    Register<bit<1>,_>(BLOOM_FILTER_ENTRIES) bloom_filter_2;
    bit<1> direction;

    // hash functions used by the bloom filter
    Hash<bit<32>>(HashAlgorithm_t.CRC16) hash_10;
    Hash<bit<32>>(HashAlgorithm_t.CRC16) hash_11;
    Hash<bit<32>>(HashAlgorithm_t.CRC32) hash_20;
    Hash<bit<32>>(HashAlgorithm_t.CRC32) hash_21;

    @name(".bloom_filter1_get") RegisterAction<bit<1>,bit<32>,bit<1>>(
    ↪  bloom_filter_1) bloom_filter1_get = {
        void apply(inout bit<1> value, out bit<1> ret) {
            ret = value;
        }
    };

    @name(".bloom_filter2_get") RegisterAction<bit<1>,bit<32>,bit<1>>(
    ↪  bloom_filter_2) bloom_filter2_get = {
        void apply(inout bit<1> value, out bit<1> ret) {
            ret = value;
        }
    };

    @name(".bloom_filter1_set") RegisterAction<bit<1>,bit<32>,bit<1>>(
    ↪   bloom_filter_1) bloom_filter1_set = {
        void apply(inout bit<1> value, out bit<1> ret) {
            value = 1;
            ret = 0;
        }
    };

    @name(".bloom_filter2_set") RegisterAction<bit<1>,bit<32>,bit<1>>(
    ↪   bloom_filter_2) bloom_filter2_set = {
       void apply(inout bit<1> value, out bit<1> ret) {
            value = 1;
            ret = 0;
        }
    };

    action drop() {
        ig_dprsr_md.drop_ctl = 1;
    }

    action ipv4_forward(macAddr_t dstMac, PortId_t port) {
        hdr.ethernet.dstAddr = dstMac;
        ig_tm_md.ucast_egress_port = port;
    }

    action set_direction(bit<1> dir) {
        direction = dir;
    }

    table ipv4_lpm {
        key     =  { hdr.ipv4.dstAddr : lpm; }
        actions =  { ipv4_forward; drop; NoAction; }
        const default_action = drop;
        size = 1024;
    }


    ↪   /* Table to determine whether the packet is incoming or outgoing, based on ports use
    table check_ports {
        key = {
            ig_intr_md.ingress_port : range;
            ig_tm_md.ucast_egress_port : range;
        }
        actions = {
            set_direction;
            NoAction;
        }
        const default_action = NoAction();
        size = 1024;
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();

            if (hdr.ipv4.ttl > 1) {
                hdr.ipv4.ttl = hdr.ipv4.ttl - 1; // decrease TTL
            } else {
                drop();
            }

            if (hdr.tcp.isValid()) {

                if (check_ports.apply().hit) {


                    if (direction == 0) { // Packet comes from internal network

                        if (hdr.tcp.syn == 1) {
                         ↪  // if syn update bloom filter and add entry
                            bit<32> index1 = (bit<32>)(hash_10.get({hdr.ipv4.
                             ↪  srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort, hdr.
                             ↪  tcp.dstPort, hdr.ipv4.protocol })[11:0]);
                            bit<32> index2 = (bit<32>)(hash_20.get({hdr.ipv4.
                             ↪  srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort, hdr.
                             ↪  tcp.dstPort, hdr.ipv4.protocol })[11:0]);
                            bloom_filter1_set.execute(index1);
                            bloom_filter2_set.execute(index2);
                        }
                    }
                    else if (direction == 1) { // Packet comes from outside

                        // Read bloom filters to check for 1's
                        bit<32> index1 = (bit<32>)(hash_11.get({hdr.ipv4.dstAddr,
                         ↪   hdr.ipv4.srcAddr, hdr.tcp.dstPort, hdr.tcp.srcPort,
                         ↪   hdr.ipv4.protocol })[11:0]);
                        bit<32> index2 = (bit<32>)(hash_21.get({hdr.ipv4.dstAddr,
                         ↪   hdr.ipv4.srcAddr, hdr.tcp.dstPort, hdr.tcp.srcPort,
                         ↪   hdr.ipv4.protocol })[11:0]);
                        meta.bloom_read_1 = bloom_filter1_get.execute(index1);
                        meta.bloom_read_2 = bloom_filter2_get.execute(index2);

                        if (meta.bloom_read_1 != 1 || meta.bloom_read_2 != 1) {
                            drop();
                             ↪   // drop packet if it's not in the bloom filter
                        }
                    }
                } else {
                  drop(); // Drop if no match in check_ports
                }
            }
        }
    }
}

    /*********************  D E P A R S E R  ***********************/

control IngressDeparser(packet_out pkt,
    /* User */
    inout my_ingress_headers_t                      hdr,
    in    my_ingress_metadata_t                     meta,
    /* Intrinsic */
    in    ingress_intrinsic_metadata_for_deparser_t  ig_dprsr_md)
{
    Checksum() ipv4_checksum; // Initialize checksum func from tofino.p4
    apply { // Update checksum with new ttl
        hdr.ipv4.hdrChecksum = ipv4_checksum.update(
                 {hdr.ipv4.version,
                 hdr.ipv4.ihl,
                 hdr.ipv4.diffserv,
                 hdr.ipv4.totalLen,
                 hdr.ipv4.identification,
                 hdr.ipv4.flags,
                 hdr.ipv4.fragOffset,
                 hdr.ipv4.ttl,
                 hdr.ipv4.protocol,
                 hdr.ipv4.srcAddr,
                 hdr.ipv4.dstAddr});

        pkt.emit(hdr);
    }
}


/*************************************************************************
 ****************  E G R E S S   P R O C E S S I N G   *******************
 *************************************************************************/

    /***********************  H E A D E R S  ************************/
/*
struct my_egress_headers_t {
}
*/

struct my_egress_headers_t {
    ethernet_h   ethernet;
    ipv4_t       ipv4;
    tcp_t        tcp;
}

    /********  G L O B A L   E G R E S S   M E T A D A T A  *********/

struct my_egress_metadata_t {}

    /***********************  P A R S E R  **************************/

parser EgressParser(packet_in        pkt,
    /* User */
    out my_egress_headers_t          hdr,
    out my_egress_metadata_t         meta,
    /* Intrinsic */
    out egress_intrinsic_metadata_t  eg_intr_md)
{
    /* This is a mandatory state, required by Tofino Architecture */
    state start {
        pkt.extract(eg_intr_md);
        transition accept;
    }
}

    /***************** M A T C H - A C T I O N  ********************/

control Egress(
    /* User */
    inout my_egress_headers_t                          hdr,
    inout my_egress_metadata_t                         meta,
    /* Intrinsic */
    in    egress_intrinsic_metadata_t                  eg_intr_md,
    in    egress_intrinsic_metadata_from_parser_t      eg_prsr_md,
    inout egress_intrinsic_metadata_for_deparser_t     eg_dprsr_md,
    inout egress_intrinsic_metadata_for_output_port_t  eg_oport_md)
{
    apply {
    }
}

    /*********************  D E P A R S E R  ***********************/

control EgressDeparser(packet_out pkt,
    /* User */
    inout my_egress_headers_t                       hdr,
    in    my_egress_metadata_t                      meta,
    /* Intrinsic */
    in    egress_intrinsic_metadata_for_deparser_t  eg_dprsr_md)
{
    apply {
        pkt.emit(hdr);
    }
}


/************ F I N A L   P A C K A G E *****************************/
Pipeline(
 ↪  // define a pipeline consisting of the individual control blocks and in which order
    IngressParser(),
    Ingress(),
    IngressDeparser(),
    EgressParser(),
    Egress(),
    EgressDeparser()
) pipe;

Switch(pipe) main; // make sure the switch uses the defined pipeline
```

F - dot graphs


__START__
hdr.ipv4.isValid();
TRUE
Ingress.ipv4_lpm
hdr.ipv4.ttl > 1;
TRUE
FALSE
tbl_firewall222
tbl_drop
hdr.tcp.isValid();
TRUE
check_ports_0/check_ports.apply().hit;
TRUE
FALSE
direction_0/direction == 0;
tbl_drop_1
FALSE

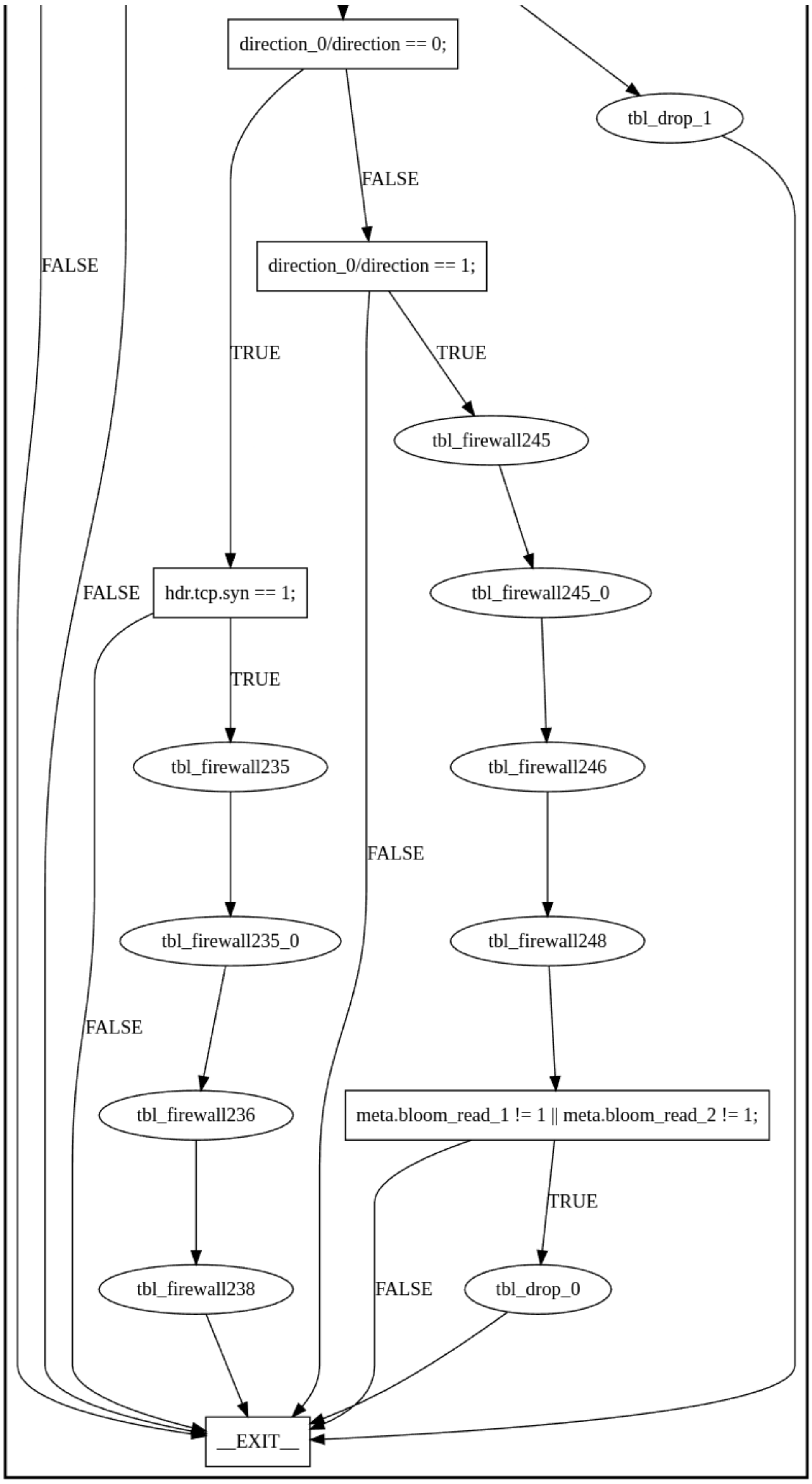


Figure 31: *Ingress DOT graph*

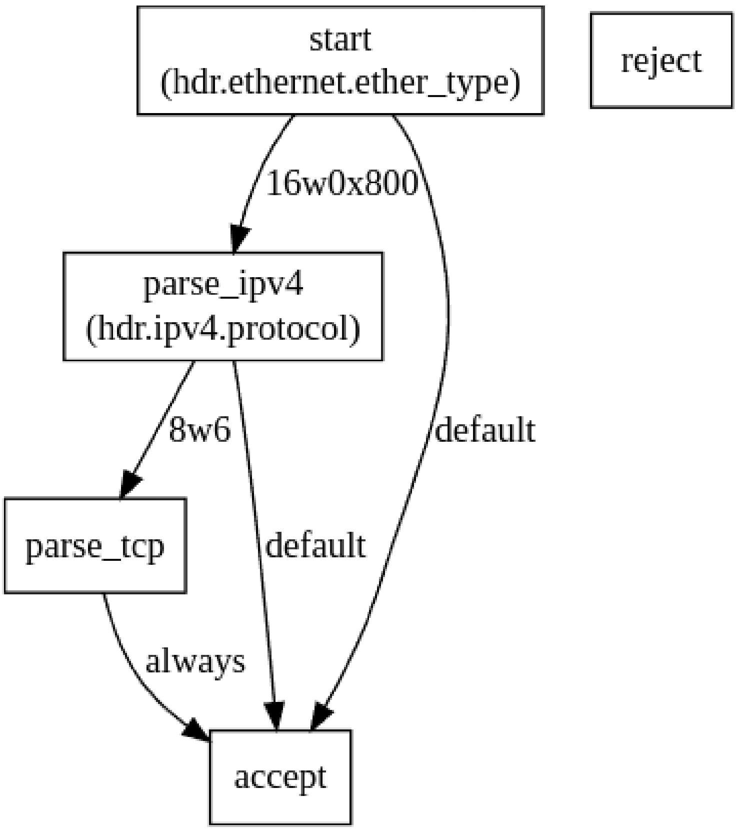


Figure 32: *Ingress Parser DOT graph*

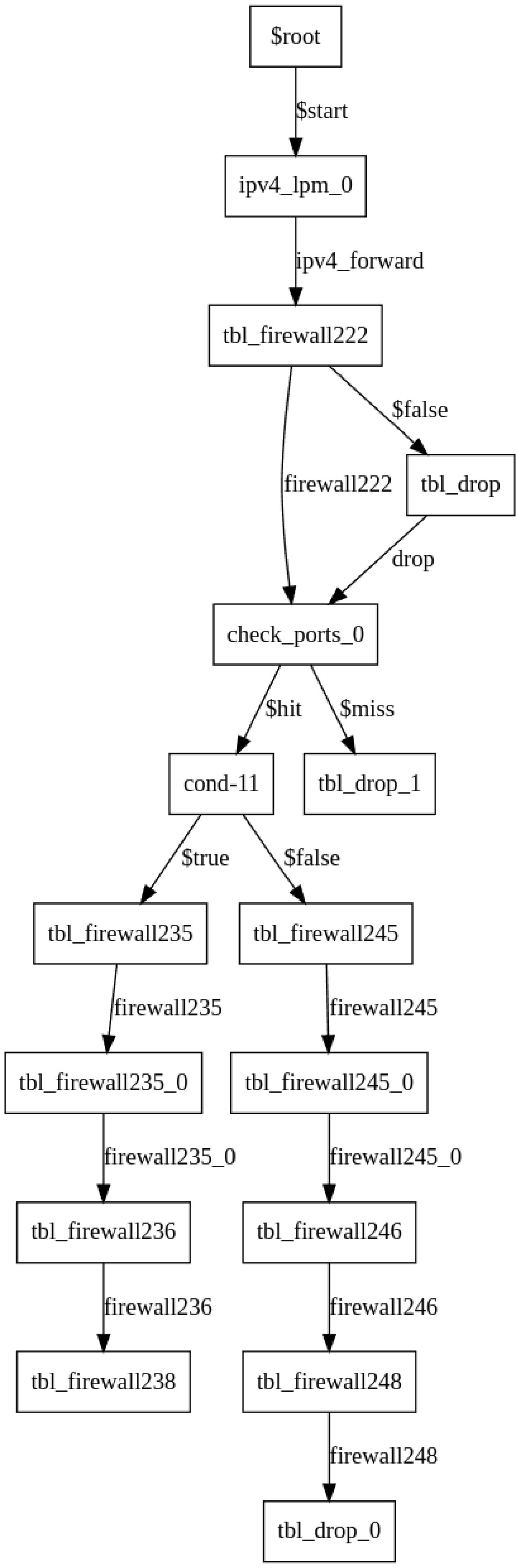
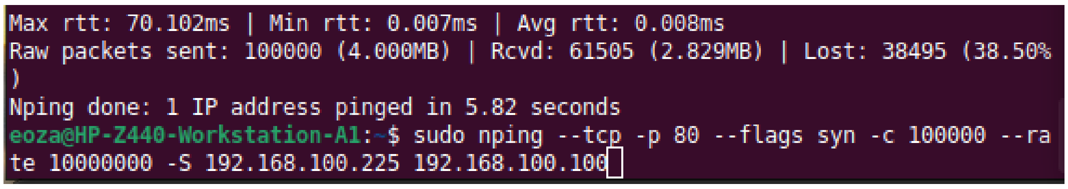


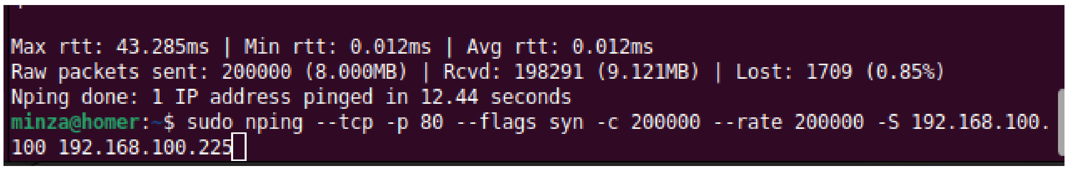
Figure 33: *Ingress power DOT graph*

G - nping results



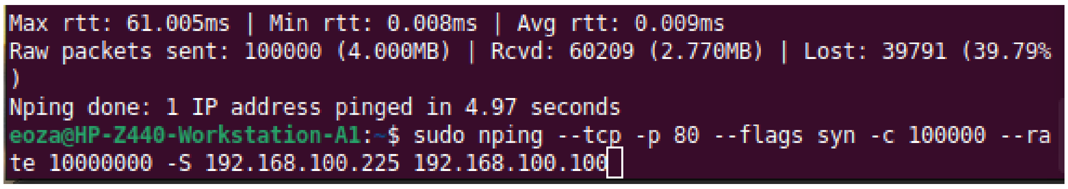
```
Max rtt: 70.102ms | Min rtt: 0.007ms | Avg rtt: 0.008ms
Raw packets sent: 100000 (4.000MB) | Rcvd: 61505 (2.829MB) | Lost: 38495 (38.50%
)
Nping done: 1 IP address pinged in 5.82 seconds
eoza@HP-Z440-Workstation-A1:~$ sudo nping --tcp -p 80 --flags syn -c 100000 --ra
te 10000000 -S 192.168.100.225 192.168.100.100
```

Figure 34: *nping forward_l2 eoza*



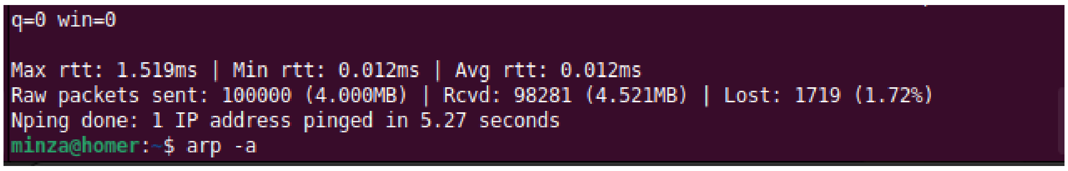
```
Max rtt: 43.285ms | Min rtt: 0.012ms | Avg rtt: 0.012ms
Raw packets sent: 200000 (8.000MB) | Rcvd: 198291 (9.121MB) | Lost: 1709 (0.85%)
Nping done: 1 IP address pinged in 12.44 seconds
minza@homer:~$ sudo nping --tcp -p 80 --flags syn -c 200000 --rate 200000 -S 192.168.100.
100 192.168.100.225
```

Figure 35: *nping forward_l2 minza*



```
Max rtt: 61.005ms | Min rtt: 0.008ms | Avg rtt: 0.009ms
Raw packets sent: 100000 (4.000MB) | Rcvd: 60209 (2.770MB) | Lost: 39791 (39.79%
)
Nping done: 1 IP address pinged in 4.97 seconds
eoza@HP-Z440-Workstation-A1:~$ sudo nping --tcp -p 80 --flags syn -c 100000 --ra
te 10000000 -S 192.168.100.225 192.168.100.100
```

Figure 36: *nping firwall eoza*



```
q=0 win=0

Max rtt: 1.519ms | Min rtt: 0.012ms | Avg rtt: 0.012ms
Raw packets sent: 100000 (4.000MB) | Rcvd: 98281 (4.521MB) | Lost: 1719 (1.72%)
Nping done: 1 IP address pinged in 5.27 seconds
minza@homer:~$ arp -a
```

Figure 37: *nping firwall minza*