Rule Verification in Software-Defined Networks by Passively Probing the Data Plane

Frode Brattensborg

Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2018

Rule Verification in Software-Defined Networks by Passively Probing the Data Plane

Frode Brattensborg

Rule Verification in Software-Defined Networks by Passively Probing the Data Plane

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Software-Defined Networking (SDN) is a new architectural approach in computer networking. Implementing the control plane into software enables dynamic network management through programmability. This new approach and virtualized architecture aim at providing network automation and multi-vendor interoperability through open standards.

In this new domain, researchers have uncovered various flaws such as faulty rules and forwarding logic caused by missing batch-update acknowledgements and faulty protocol implementations. In this thesis, we address the issue of how to verify the presence of various types of entries seen in SDN flow tables by actively probing the data plane.

The thesis further compares dedicated probe injection points used in other works with a direct injection point using OFPP_TABLE. Probing is performed by interacting directly with the controller and the communication with the data plane is taking place on the control channel.

Different scenarios are presented for probing different types of rules. We show that by using a linear search algorithm we can create distinct rule-matching probes or 500 rules in 1.6 seconds and 3000 rules in 11.5 seconds. By measuring the time probes spend on-wire we show that the overhead of sending probes into the network via a dedicated injection point instead of using OFPP_TABLE varies from a 3.5% increase to $\approx$ 8%.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem statement . . . 3
- 1.3 Thesis outline . . . 3

2 Background **5**
- 2.1 Software-Defined Networking (SDN) . . . 6
- 2.2 Controllers . . . 7
  - 2.2.1 Ryu . . . 7
- 2.3 OpenFlow . . . 9
  - 2.3.1 Switches . . . 9
  - 2.3.2 Flow tables . . . 10
  - 2.3.3 Match & action . . . 11
  - 2.3.4 Packet in/out . . . 13
- 2.4 Mininet . . . 14
- 2.5 Relevant work . . . 16
  - 2.5.1 CherryPick . . . 16
  - 2.5.2 NetSight . . . 17
  - 2.5.3 Monocle . . . 17
  - 2.5.4 RuleScope . . . 18
  - 2.5.5 SERVE . . . 19
  - 2.5.6 SDN traceroute . . . 19
  - 2.5.7 VeriFlow . . . 20

3 Approach **23**
- 3.1 Objectives . . . 23
- 3.2 Design . . . 24
- 3.3 Implementation . . . 28
  - 3.3.1 Topology . . . 28
  - 3.3.2 Controller . . . 29
  - 3.3.3 Gathering . . . 30
  - 3.3.4 Flow matching . . . 31
  - 3.3.5 Install/remove . . . 33
  - 3.3.6 Inject . . . 33
  - 3.3.7 Catch . . . 34

4 Experiments & results **37**

4.1 Rule generation . . . 37
4.2 Matching . . . 38
4.3 Benchmarking . . . 41
4.4 OFPP_TABLE . . . 43
4.5 False positive . . . 45
4.6 Drop rule . . . 45
4.7 Forwarding loop . . . 46
4.8 Dedicated point of entry . . . 48
4.9 Scaling . . . 50

5 Discussion **53**
5.1 Matching . . . 53
5.2 Project . . . 55
5.3 Experiments . . . 55

6 Conclusion **57**
6.1 Conclusion . . . 57
6.2 Future work . . . 58

Appendices **65**

A OpenFlow **67**
A.1 OFPT_FLOW_MOD . . . 67
A.2 OXM fields . . . 70

B Tests/output **73**
B.1 Benchmarks . . . 73
B.2 Comparison . . . 76
B.3 Hardware . . . 78
B.4 OFPP_TABLE . . . 80
B.5 Packetgen/match . . . 83
B.6 FP-test . . . 85
B.7 Drop-test . . . 87
B.8 Loop-test . . . 90

C Code **93**
C.1 Functions . . . 93
C.2 Controller . . . 95
C.3 Generator . . . 106
C.4 Scraper . . . 114
C.5 Simple topology . . . 117
C.6 Scaled topology . . . 118
C.7 Dedicated topoloy . . . 119

List of Figures

2.1 Traditional vertical network stack. 5
2.2 Architecture of SDN. 6
2.3 Ryu architecture 8
2.4 OpenFlow switch components [11] 9
2.5 OpenFlow pipeline processing [11]. 10
2.6 Anatomy of a flow table entry in OpenFlow [13] 11
2.7 Packet flowchart through an OpenFlow switch [14] 12
2.8 Mininet architecture [16] 15
2.9 CherryPicking links [17] where (x, y) equals link and detour. 16
2.10 Netsight arcitecture [19]. 17
2.11 Unicast-rule verification in Monocle. [20]. 18
2.12 RuleScope workflow and architecture [23] 18
2.13 SERVE architecture [24] 19
2.14 SDN Traceroute flow of operation [18] 20
2.15 VeriFlow architecture [28] 21

3.1 Verification flow in a simple topology 24
3.2 IPv4 DSCP field 25
3.3 Shadow rule [31] 26
3.4 L3 and L4 values comprising the 5-tuple set 26
3.5 Architectural flowchart diagram 27
3.6 Ryu application workflow [35] 30
3.7 Venn diagram depicting overlapping rules 32
3.8 Storing packets in a FIFO qeueu 35

4.1 Match field value distribution. 38
4.2 Match field value/header distribution. 39
4.3 Overlapping rule 39
4.4 Short term storage structure. 41
4.5 Time to generate distinct probes. 42
4.6 Verifying OFPP_TABLE 43
4.7 Time spent on link. 44
4.8 False positive testing 45
4.9 Testing drop rule 46
4.10 Forwarding loop 47
4.11 Time to check for loops 47
4.12 Dedicated point of entry 48
4.13 Time spent on link. 49

4.14 Scaled topology 51

5.1 Rule generation boxplot (500 to 1500 rules) 54
5.2 Rule generation boxplot (2000 to 3000 rules) 54

B.1 OFPT_PACKET_OUT capture showing the OFPP_TABLE set. 81
B.2 OFPT_PACKET_IN capture showing the incoming packet. . 82

List of Tables

3.1 5-tuple value combination 32

4.2 Time to REST 42
4.1 Packet generation 42
4.3 Time spent on link 44
4.4 Time spent on link (in seconds) 49

A.1 OXM Flow match field types. 71

B.1 Packet generation 74
B.2 Time spent on link 74
B.3 Linktime with dedicated point of entry 75
B.4 Hardware specifications 79

C.1 Support-functions 94

Preface

This master thesis was written at the Faculty of Mathematics and Natural Sciences, at the University of Oslo (UiO) in the spring of 2018. The thesis is a product of 30 ECTS at the Network and System Administration (NSA) programme which is a collaboration between UiO and the faculty of Technology, Art and Design at Oslo Metropolitan University (OsloMet).

I would like to thank associate professor Anis Yazidi and assistant professor Ramtin Aryan for supervising the thesis, their guidance, discussions and for their ideas and valuable feedback throughout this semester.

Thanks to my girlfriend Marielle R. Øverby for being supportive and patient during the past few weeks.

Finally, I would like to thank my fellow students for two great years at the Network and System Administration (NSA) programme.

Chapter 1

Introduction

In this chapter, the reader will find a brief introduction to computer networks and its growth followed by the evolution of Software-Defined Networks (SDN). Challenges in the troubleshooting arena are discussed and cost calculation of network downtime in datacenters and cloud infrastructures is presented. The reader will be introduced to terminology. The problem statement and its focus area are outlined after. The project layout is provided at the end of this chapter.

1.1 Motivation

Computer networks have experienced a steady growth since its creation in the late 1960s, but in last decade this growth has rapidly increased. More devices get interconnected, datacenters are expanding, businesses encourage bring-your-own-device (BYOD) policies, Internet of things (IoT) is on the rise and end users are spanning multiple consumer-devices. Vendors are following this growth offering network appliances which shuffle more packets at faster rates – increase in overall bandwidth[1]. By interconnecting branch offices and content providers serving higher quality data on demand – this growth has been quite rapid.

As a result of this growth, IP networks have become complex, hard to manage, prone to errors and logical flaws [1] resulting in time-consuming management and fault handling [2]. Troubleshooting of computer networks tends to be labour intensive and intricate work. Network outages might seem almost unavoidable and may be caused by various reasons. Typical reasons for an outage include, but are not limited to, human errors like faulty configurations or malicious activities, equipment malfunction and force majeure events.

In addition to troubleshooting, the widespread increase in the use of server virtualization and containerization technologies is in addition to

[1] Bandwidth: data transmission capacity of a communications channel.

the troubleshooting arena another factor pushing for a more flexible approach for traffic control and provisioning. Traditional networking architectures [3] are having trouble keeping up as virtual machines and software containers dynamically move around. An effect of this new and emerging traffic picture is a need for dynamic configuration and allocation of network resources.

Businesses and service providers are striving to ensure the reliability of their network. Sitting between users and applications, the network alone is a critical component on which both parties heavily rely upon. In a survey done by H. Zeng et al. [4] they found that 35 percent of networks generate more than 100 trouble tickets per month on average. In another survey Mota E. S. & Fonseca, P. C. (2017) [5] looked into application and infrastructure failures at 28 cloud providers from 2007 to 2013 and estimated around 1.600 disruptive hours with a cost frame of approximately $273M in total. Downtime in information and communications technology (ICT) services imposes a huge cost for both businesses and the society. IHS Markit found in one of their surveys [6] from 2016 that the aggregated cost of downtime in ICT is costing organizations in North America $700Bn per year. The survey found that network interruptions were the biggest culprit of the recorded downtime.

The evolved complexity of computer networks is outpacing the traditional tools that IT personnel have at their disposal [4] when troubleshooting various outages. Technicians and operators are still constrained to the layered architectures of the OSI-model[2] and TCP/IP[3] protocol stack and a handful of manual tools for troubleshooting.

Two usually tightly bundled together and central pieces in today's network devices are the forwarding plane and control plane. The control plane is responsible for making decisions on how packets should be forwarded i.e. fine-tuning the forwarding table, and the forwarding plane (or data plane) is responsible for handling packets based on the instructions received from the control plane i.e. forward, drop or change packets.

SDN and its layered architecture decouple these two planes. This opens up for a dynamic approach to managing and configure networks. Centralizing the control functionality enables programmability of network control functions and elements. With the rise of SDN and its high level of abstraction, it is expected to achieve optimized dataflow, smarter automation and added flexibility. By utilizing the capabilities of the OpenFlow protocol, verification of the overall network state is achievable from the controllers central point of view.

This new architecture however is prone to errors [7] and bugs [4]. D Kreutz et al. [1] (2015) categorized in a broad survey common errors into controller logic bugs, race conditions and software bugs & performance disturbance. The categories are dealing with problems like reachability

[2]OSI-model: 7-layered reference model and framework for network communications.
[3]TCP/IP: the non-proprietary protocol suite for network communications.

issues, events being processed in different orders and transient failures caused by CPU spikes. The architectural change and the programmability provided by SDN both allows and urges the emerging of new approaches for troubleshooting, re-thinking of the workflow and development of new tools for troubleshooting.

The aim of this project is to look at how we can verify the presence of rules of different types installed on SDN enabled network devices residing in the data plane.

1.2 Problem statement

Software-Defined Networking is a popular research domain within academia. Numerous publications improving on traditional tools and techniques for network troubleshooting have been published. But researchers have also discovered various flaws [8] ranging from faulty protocol implementations to switches prematurely reporting rules as installed [9]. Firmware errors [10], loss of rule-update messages or batch-level rule update acknowledgements can result in missing rule faults [8] where rules in the data plane are reported as installed, or updated, but proven to not be active or working. The following research questions are proposed:

1. How can we ensure network policies installed work as intended?
2. How can we insert probes into the data plane in an effective manner?
3. How can we probe policies using a least-amount of catch-rules?
4. How can we verify different types of rules e.g. unicast, drop and loop-causing rules?

1.3 Thesis outline

The report is outlined as follows:

1. Chapter 2 provides an overview of Software-Defined Networks, controllers and the underlying protocol enabling the control plane separation. OpenFlow is discussed in some detail along with the emulator software used.
2. In chapter 3 the projects design phase is outlined with the intended flow of operation and the implementation of the supporting architecture.
3. The experiments performed during this project is found in chapter 4 along with their results. The experiments range from rule generation to packet matching and various types of rule-verification.

4. Chapter 5 presents a brief discussion of the project and overall achievements during the project.
5. In chapter 6 a reflection on the project, the experiments performed and their results are given. The chapter also discusses limitations, strengths and weaknesses.
6. In chapter 7 the conclusion of the research performed in this task is given along with proposed future work.

Chapter 2

Background

A computer network is a collection of various devices communicating and exchanging data with each other. Communication is achieved using various protocols, relies on transit devices and often different transportation methods for carrying the data packets are used. Traditional computer networks typically comprise of individual pieces of networking equipment and dedicated appliances, which contain everything from the hardware and its configuration to the intelligence of that appliance.

These devices are vendor specific and their software is vertically bundled in with the hardware, as shown in figure 2.1, effectively making them rather inflexible. With the distributed control plane, each device has its own "brain". Networks as a whole is often configured semi-indirectly and these appliances usually rely on distributed protocols to converge e.g. STP [1] or OSPF [2] in order to achieve optimal solutions or intended operational states.

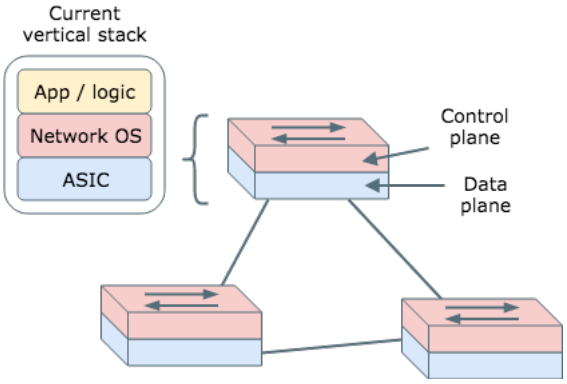

Figure 2.1: Traditional vertical network stack.

[1] STP: Spanning-tree protocol for building loop-free network topologies
[2] OSPF: Open Shortest Path First, a routing protocol for IP networks.

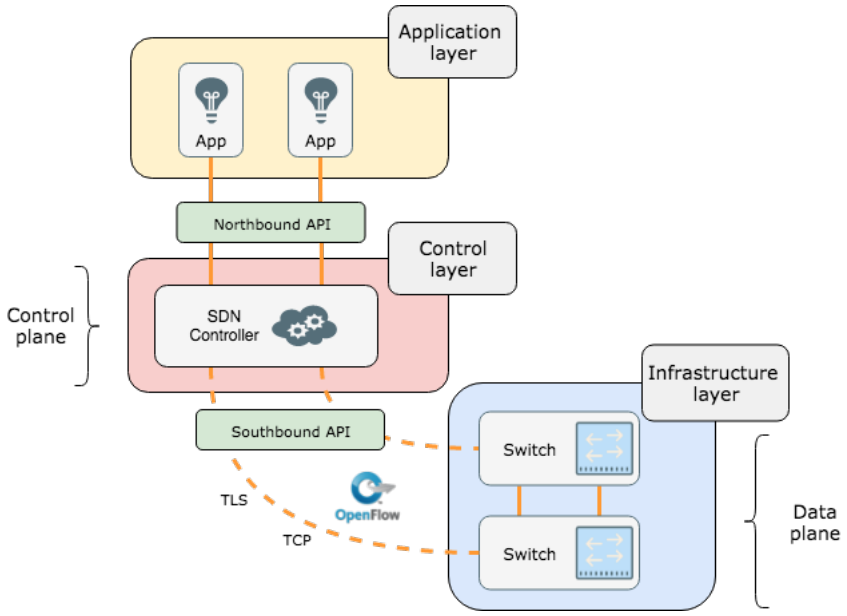

Figure 2.2: Architecture of SDN.

2.1 Software-Defined Networking (SDN)

Software-defined networking (SDN) offers centralized network management by logically separating the control plane from the data plane. For a typical network device the data, or forwarding, plane is responsible for moving packets and the control plane is where the logic resides – the brain, who makes decisions about how and where to forward packets. This decoupling opens up to implementing the control plane into software. Most traditional network devices today are directly and manually managed. With the logical separation, they can now be managed and configured securely from a central controller.

The main goal of SDN is for the network to be open and programmable. This paves way for faster deployment, scaling, automation and building applications communicating with the various network devices. If an organization requires a specific type of network behaviour it can develop or install an application to do what it needs. These applications may be for common network functions such as traffic engineering, security, QoS [3], routing, switching, virtualization, monitoring, load balancing or other new innovations.

At the core of SDN is the creation of an environment where this kind of flexibility exists and the network can evolve at the speed of software. The principle of SDN can be introduced into traditional networks as the open standards are getting traction from various top technology conglomerates. A simple SDN architecture can be seen in figure 2.2.

[3]Quality of Service: mechanisms for prioritizing specific types of traffic – ensure/reserve resources

2.2 Controllers

In SDN networks the controller is the brain of operation. The network components residing in the data plane rely on the controller to send information downwards via the southbound APIs telling them how to manage flows, or relay application business logic all the way from the northbound APIs and down. Controllers can be clustered for higher availability, stronger reliability and more flexible scalability.

A typical SDN controller can do basic CRUD operations on an OpenFlow switch's flow table entries, i.e. create, read, update and delete. This is done either reactively or proactively. Proactive flow entries are entries which are programmed/installed beforehand – before traffic arrives. Reactive flow entries are installed as a reaction to a PacketIn from a switch to the controller. Figure 2.5 depicts the flow as ingress packets arrive at the switch, gets processed by the entries in the flow table and egress out either a physical or logical port.

Typically a controller might contain pluggable modules for different network tasks like inventorying, host tracking, statistics gathering or topology services. Typically modules provide REST [4] APIs, internal APIs, gather and use network and topology information to implement traditional routing protocols or enable quality of service mechanisms. The two most well-known protocols for southbound communication with the switches and routers in the data plane is OpenFlow and OVSDB. Northbound communication depends on the controller software and what language it is written in. The choice of languages varies and range from C to Java and Python.

2.2.1 Ryu

Ryu is a network framework for Software-Defined Networking. The framework is open source and Ryu is a component-based python controller supporting various protocols for managing network devices, such as OpenFlow versions ranging from 1.0 to 1.5, Netconf and OF-config. Ryu aims to be an operating system for software-defined networks. It has various advantages like integration with OpenStack, support for various Nicira extensions and a well-defined API. All of Ryu's code is freely available under the Apache 2.0 license and the controller is fully written in Python. Being implemented purely in Python might make the software somewhat slower than other controller software like Floodlight, ONOS, POX and Beacon which are implemented in Java and C.

A Ryu application is a single threaded python module. Different modules can run simultaneously and implement various functionalities. Triggered events are messages between these entities. The message is asynchronous,

[4] RESTful: Representational State Transfer, web services who provide interoperability between systems.

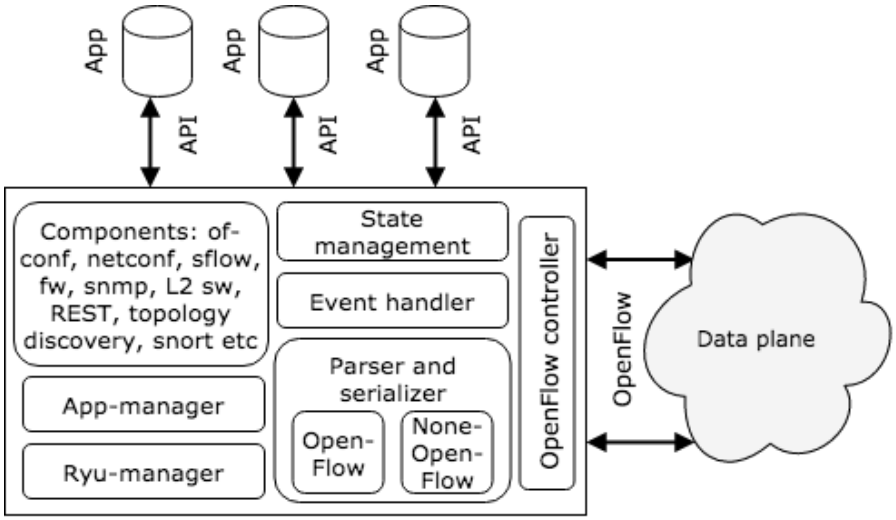

Figure 2.3: Ryu architecture

meaning they are not coordinated. Each Ryu application has a first in first out (FIFO) queue where received events gets stacked, preserving the event order. The underlying architecture is shown in figure 2.3. Each application has a dedicated thread to handle these events. If the event handler somehow gets blocked, processing is paused. The most common modules are listed below:

- **bin/ryu-manager** is the main executable.
- **ryu.base.app_manager** handles the central management of Ryu applications and is responsible for loading Ryu applications, providing context to Ryu applications and route message between Ryu applications.
- **ryu.controller.controller** is the main component of the OpenFlow controller and handles connections from switches and both generate and route events to and between appropriate Ryu applications.
- **ryu.controller.ofp_event** integrates the OpenFlow event definitions.
- **ryu.controller.ofp_handler** provides basic OpenFlow handling including negotiation.
- **ryu.ofproto** imports OpenFlow definitions and implementation
- **ryu.topology** is a switch and link discovery and management module
- **ryu.lib.packet** is a Ryu packet library provides decoder/encoder implementations of various protocols from the TCP/IP stack.

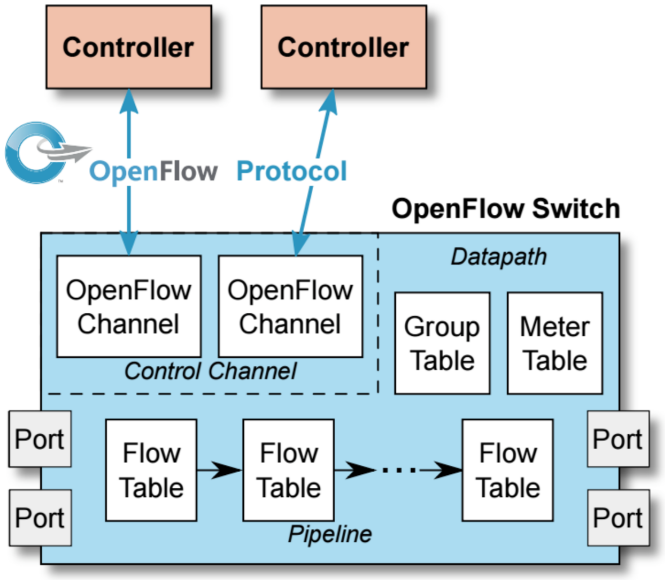

Figure 2.4: OpenFlow switch components [11]

2.3 OpenFlow

The OpenFlow [11] protocol is a standardized suite of protocols for interacting with the forwarding behaviours of switches from multiple vendors. The protocol suite is maintained by the Open Networking Foundation (ONF) [12]. In SDN environments OpenFlow defines the communication protocol enabling the controller to directly interact with the data plane of network devices. This provides a way to dynamically and programmatically control the behaviour of switches throughout a network. The protocol is comprised of four components dictating the overall flow of operation, define a valid message structure, semantics to controlling negotiation of transport channels, certifications and capability discovery, flow handling and statistics.

2.3.1 Switches

The anatomy of an OpenFlow enabled switch is depicted in figure 2.4 and is comprised of ports and flow tables. Network packets enter and exits the switches through the ports. The flow tables consist of various entries defining what actions to apply to the different types of packets. Types of actions range from queue, forward to drop or alter the packet as it passes through the switch. OpenFlow switches are broken into two components: a switch agent and the data plane. The agent communicates with the controller via the OpenFlow protocol and translates commands and messages into low-level instructions. The data plane performs the packet processing and packet forwarding.

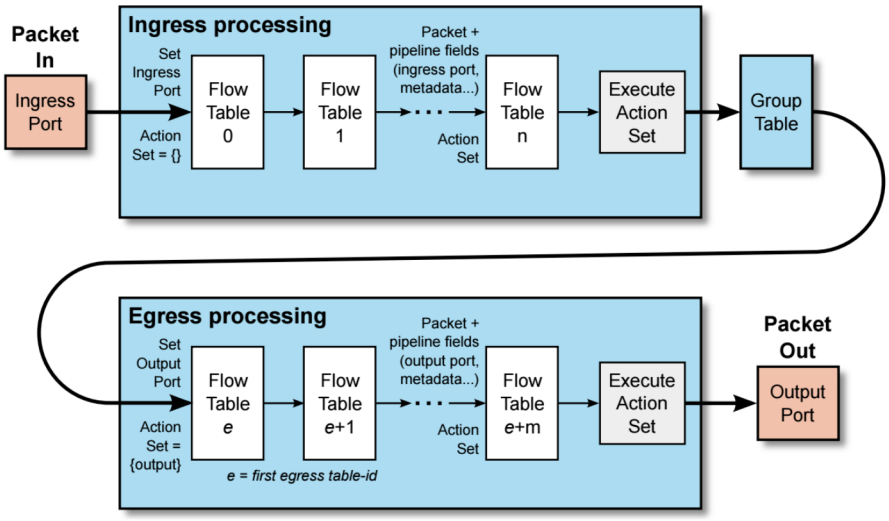

Figure 2.5: OpenFlow pipeline processing [11].

2.3.2 Flow tables

A traditional forwarding information table (FIB) uses information like the destination IP address from the layer 3 IP header to decide the next hop for a packet. A flow table in OpenFlow is a broad term. Its entries may use, or combine, information from multiple OSI layers and ports to form their respective entries – they are in some way similar to, and may resemble, access control lists (ACLs). Each flow table is comprised of entries consisting of classifiers, actions and instructions.

Flow table entries consist of a set of instructions which are applied to all matching packets. Each flow entry is uniquely identified by the combination of the match and priority fields, as seen in figure 3.6. As with firewalls and ACLs they usually hold an implicit catch-all/deny-all rule found at the bottom of the list. In OpenFlow, a table-miss flow entry with the priority of 0 and all-wildcard fields is usually installed. The rule sends a packet up to the controller if a match occurs. Entries in OpenFlow can match on fields and properties from switch ports to various protocol headers and fields in frames, packets and datagrams. In version 1.3 there are over 40 header fields from more than 7 protocols to match against. A list of all (OXM) fields to be used for matching can be found in appendix A.1.

Incoming packets flow through a pipeline as seen in figure 2.5 and any actions to be performed on the packet upon an entry is contained in the instructions field. The instructions table is comprised of the following entries: match fields, priority, counters, instructions, timeouts, cookies and flags.

- The match fields defines fields in the packets to match against and typically consists of ingress port and packet headers. Other fields

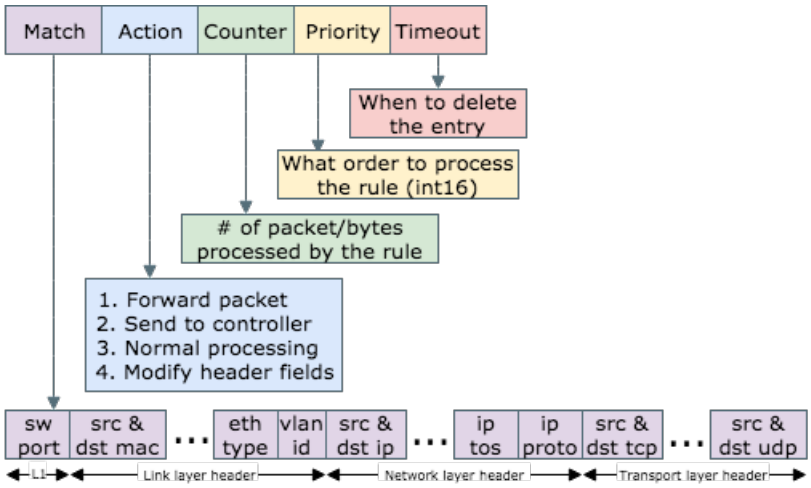

Figure 2.6: Anatomy of a flow table entry in OpenFlow [13]

such as metadata is also used for pipelining between flow tables.

- The priority field specifies the precedence of the flow entry.
- Counters hold metrics which is updated when packets get matched.
- Instructions are used for processing the pipeline or modify the set of actions.
- Timeouts controls if and when active and idle flows on the switches are to expire.
- The cookie field is used by the controller and not by the switches when packets are processed. The values are used to filter, modify or delete flow entries.
- Flags are used to alter how flow entries are managed. E.g. OFPFF_SEND_FLOW_REM triggers flow removed messages for that flow entry.

2.3.3 Match & action

As defined in the OpenFlow standard, packets can match on switchports and various fields found in headers from Layer-2 frames, Layer-3 packets and Layer-4 datagrams. When an OpenFlow switch receives a packet the switch starts performing table lookups based on its pipeline processing.

Every entry in the flow table holds a set of instructions telling what to do upon a packet match. Each instruction set can hold a maximum of one of each instruction type. Instructions perform various types of actions e.g. applying an action list, clear out the actions, write an action or metadata, or apply a rate limiter.

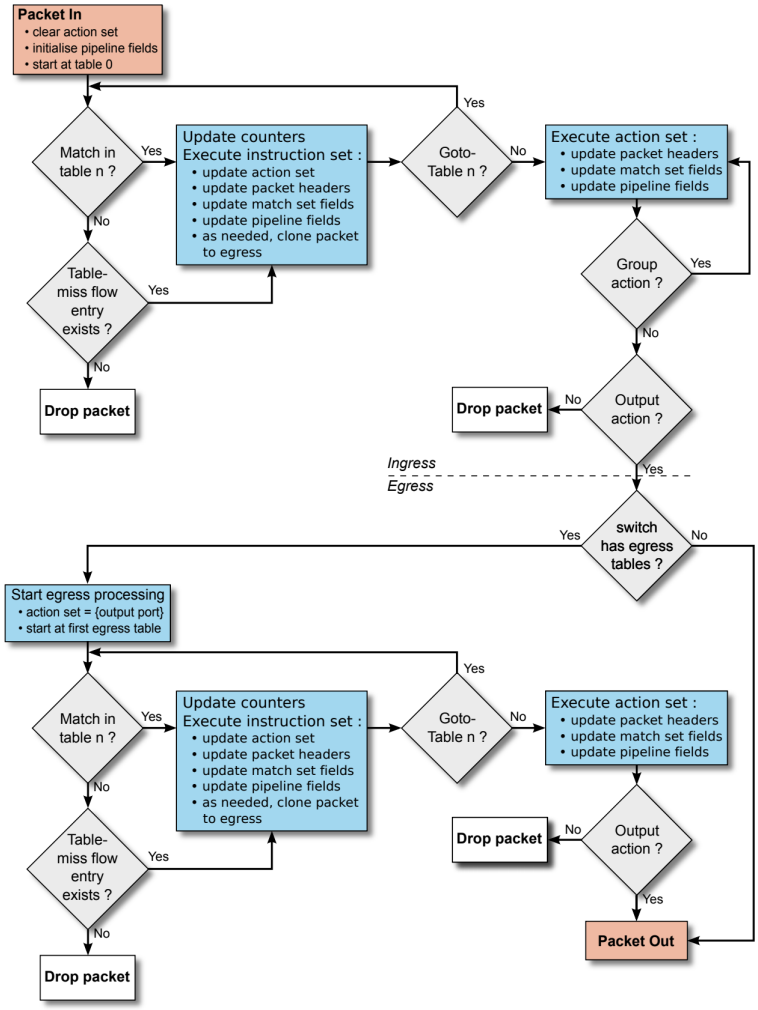

Figure 2.7: Packet flowchart through an OpenFlow switch [14]

Every packet has a set of actions associated with it. These are policies determining what should happen to the packet. Together with instructions, these options decide how packets are processed after they have matched against an entry found in the flow table. The action set is empty by default. In version 1.3 of OpenFlow actions can be performed against logical and physical port IDs and numerous headers and protocols like Ethernet, VLAN, ARP, MPLS, IPv4, IPv6 to TCP, UDP, SCTP, ICMPv4 and ICMPv6. Typical actions are drop, forward, decrement TTLs, (re)-write operations like push/pop VLAN headers or MPLS tags and set operations for packet altering and modifications to mention a few.

2.3.4 Packet in/out

OFPT_PACKET_IN messages are sent from an OpenFlow switch (datapath) to a controller. As seen in figure 2.1 the message consists of an OpenFlow header followed by a buffer id, total length, reason, table id, cookie and OFP_MATCH object. The buffer id is an opaque value used by the datapath to point to a locally buffered packet. The 8-bit reason field indicating why the packet is being sent (OFPR_* object), table_id of the flow table where the match happened, a cookie of the matched flow entry. The main reasons for why a captured packet gets sent from a datapath to a controller are either due to an explicit flow entry action, the packet matched a flow-miss rule in the flow table or the packet has an invalid TTL value.

```
1  /* Packet received on port (datapath -> controller) */
2  struct ofp_packet_in {
3    struct ofp_header header;
4    uint32_t buffer_id;      /* ID assigned by datapath */
5    uint16_t total_len;      /* Frame length */
6    uint8_t reason;          /* Reason packet is being sent */
7    uint8_t table_id;        /* Flow table ID */
8    uint64_t cookie;         /* Cookie of the flow table entry */
9    struct ofp_match match;  /* Packet metadata. Variable size. */
10 };
```

Listing 2.1: PacketIn header [11]

Any OpenFlow compatible SDN controller has the ability to inject packets into the data plane at any given time by using the OFPT_PACKET_OUT message. These outgoing packets can be injected onto any switch supporting the OpenFlow controller and who is connected to the controller. The packet out message can either carry a pointer to a place in the switch's local buffer for any stored packets to be released, or the message can carry a raw packet to be injected. The injected packets can be treated as normal packets if normal table processing in action set is indicated.

The 32-bit buffer_id field holds a pointer value. A value of 0xFFFFFFFF indicates that the packet contains a raw packet in the data byte array.

Otherwise, the value points to a packet stored in the switch's local buffer. The 32-bit in_port value determines the arrival port of the packet and is used when the packet must undergo standard table processing. The 32-bit action list holds a list of actions to be applied to the packet. The packet is dropped of the list is empty.

```
/* Send packet (controller -> datapath) */
struct ofp_packet_out {
  struct ofp_header header;
  uint32_t buffer_id;      /* ID assigned by datapath */
  uint32_t in_port;        /* Packets input port */
  uint16_t actions_len;    /* Size of action array */
  uint8_t pad[6];
  struct ofp_action_header actions[0]; /* Action list */
  uint8_t data[0];         /* Packet data */
};
```

Listing 2.2: PacketOut header [11]

2.4 Mininet

Mininet [15] is an emulator for creating virtual networks comprised of hosts, switches, links etc. By using process-based virtualization and network namespaces Mininet can emulate both large and realistic network topologies, with OpenFlow capabilities, using minimal system resources on a single machine. The emulator ships with multiple built-in controllers like ref, OVSC and NOX, but also supports the use of external/remote controllers like Beacon, Floodlight, Ryu etc. Mininet offers a Python API for creating complex custom topologies, interacting with nodes and perform other forms of experimentation.

By prototyping large topologies in an effortless manner, Mininet provides inexpensive network testbeds. Interaction with the created networks can also be done via the included command-line interface (CLI). Natively, the emulator includes tcpdump.

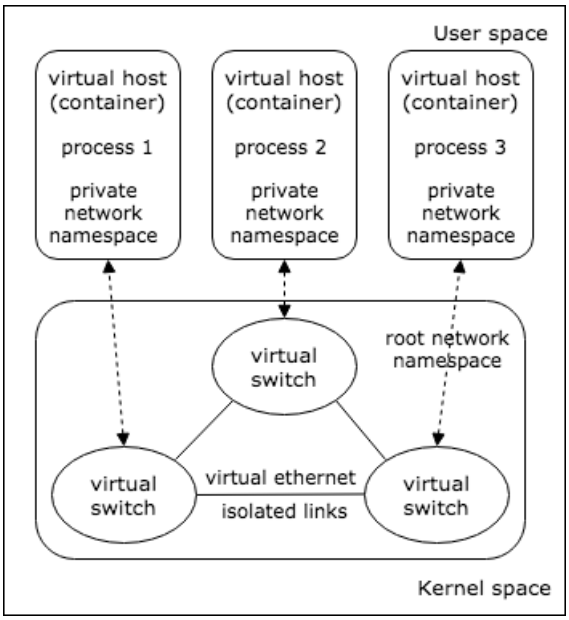

Figure 2.8: Mininet architecture [16]

2.5 Relevant work

Software-Defined Networking is a hot topic in academia. It provides a new paradigm to many networking concepts like traffic orchestration and management to troubleshooting. Many approaches for data plane troubleshooting have been researched. Network issues stemming from internal failures of network devices or software bugs are hard to troubleshoot as they are not seen by the control plane. Verifying the actual network behaviour and the desired state by confirming the presence of data plane rules can help determine the location of the rule(s) responsible for causing havoc. In this chapter we'll provide brief insights into various academic works. Some research about troubleshooting in SDN is reviewed as follows.

2.5.1 CherryPick

CherryPick [17] is a scalable in-band technique for tracing packet trajectories in SDN. CherryPick uses the assumption of a datacenter topology and exploits this structure to minimize the number of data plane rules and packet header space required to trace a packet's path. Each network link gets assigned a unique identifier using edge coloring[5] techniques to map the Top-of-Rack (ToR), Aggregate and Core switches. The technique is applied to a fat-tree topology. Quite similarly to [18] CherryPick is using 802.1ad[6] field to store values. Mapping the path in a fat-tree topology using colouring minimizes the number of switch flow rules and packet headers needed by selectively gathering the links representing an end-to-end path. The path gets embedded into the packet header while making hops on its way to the destination. Trading off slightly more packet header space requirements for all unique links, CherryPicks solution proves to significantly improve as they require three orders of magnitude fewer switch rules than competitors when evaluating a 48-ary fat-tree topology.

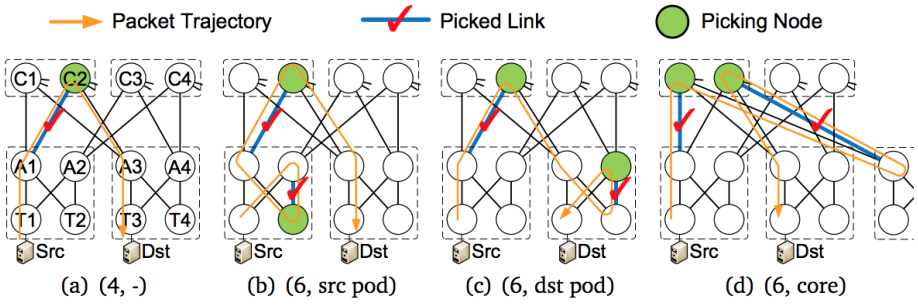

Figure 2.9: CherryPicking links [17] where (x, y) equals link and detour.

[5]Edge colouring: topic in graph theory where all edges in a network graph are coloured so that no adjacent edges share the same colour.

[6]802.1ad (QinQ): network standard allowing Ethernet frames to hold multiple VLAN tags.

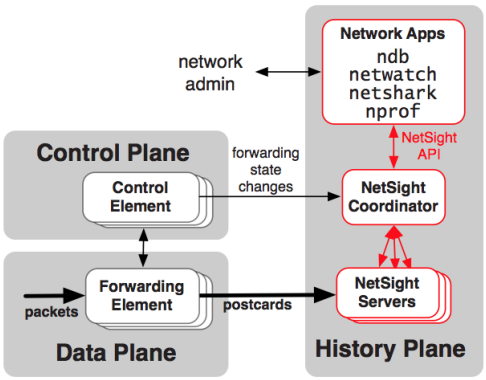

Figure 2.10: Netsight arcitecture [19].

2.5.2 NetSight

NetSight is a platform for improving network visibility by capturing what they call packet histories [19]. By transparently transposing the control channel between the switches and the SDN controller they can listen in on information from packets passing by and sends the information up to the application. Information of interest is switch state and header modification. The postcards are created at each hop of the packet's journey and contain packet headers, the matching flow table entry and the output port on the corresponding switch. For each hop, this information is sent back to the controller, picked up and stored for later analysis. The platform consists of four applications: (*ndb*) for interactive debugging, (*netwatch*) for live network invariant monitoring, (*netshark*) for logging and (*nprof*) for network profiling.

2.5.3 Monocle

Monocle [20] provides dynamic and fine-grained data plane monitoring by verifying that the network state derived from the data plane matches with the desired state the controller holds i.e. ascertaining the correspondence between the high-level network policies and the data plane configuration. This problem, the state mismatch, is referred to as data plane correspondence. Monocle verifies the network state by actively monitoring the data plane. By placing itself as a proxy between the controller and the switches, similarly as Netsight [19], Monocle listens in on the communication taking place on the control channel and performs southbound interaction with the various network devices in the data plane. The data plane is monitored in two ways; statically by systematically probing of switches and dynamically by listening for packets on the control channel indicating changes to the flow tables. After retrieving the flow table from a switch, Monocle generates a distinct probing packet upon a rule modification to exercise the newly installed rule. Before probes are injected onto the data plane, catch rules are installed onto the switches. Monocle can now examine the rules

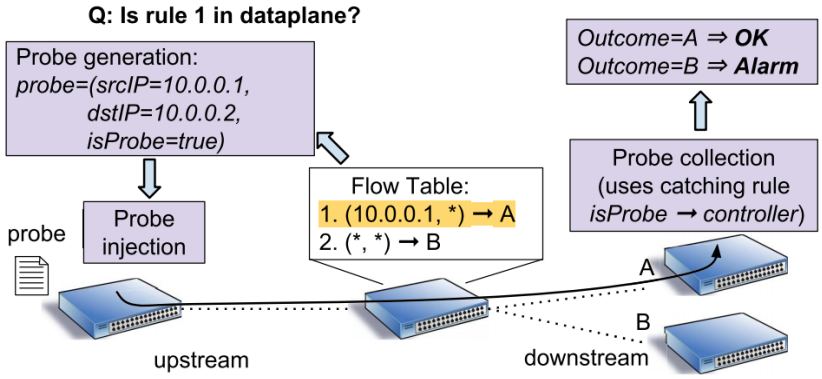

Figure 2.11: Unicast-rule verification in Monocle. [20].

and see if the overall behaviour matches the intended behaviour and thus the state from the controllers perspective.

2.5.4 RuleScope

SDN forwarding has been exposed to various faults and vulnerabilities in previous papers [8, 21, 22]. RuleScope presents a method for accurately and efficiently inspect forwarding [23]. The tool detects forwarding faults in the data plane by looking at missing rules and priority faults using customized probing-packets. Missing faults occur when a rule is not active on a device and a priority fault occur when a rule violates the designated order. RuleScopes monitor application is comprised of two main functions; detection and troubleshooting. The detection algorithm aims to find faulty rules and forwarding faults in the data plane. The troubleshooting algorithm of the application pulls flow tables from the SDN switches to find priority faults by looking at rule dependencies and building a dependency graph. Probing with respect to the dependencies and a calculated expected outcome paired with Netsights2.10 postcards will assist in finding priority faults.

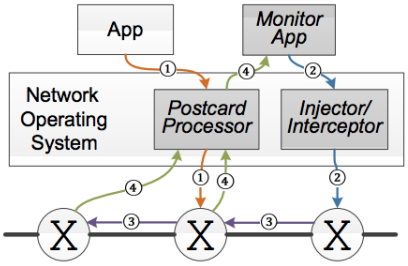

Figure 2.12: RuleScope workflow and architecture [23]

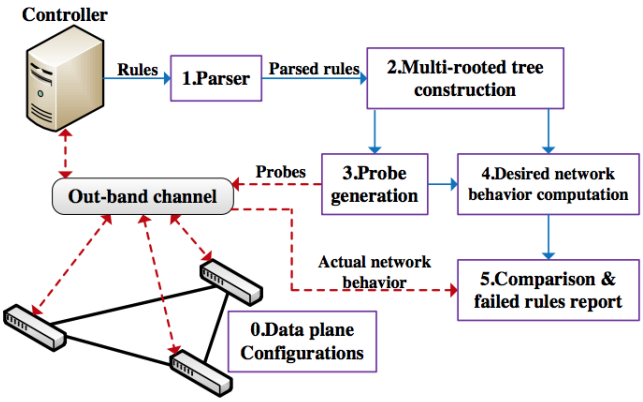

Figure 2.13: SERVE architecture [24]

2.5.5 SERVE

SERVE presents an SDN-enabled rule verification framework to identify network issues residing in the data plane. Specifically targeting invisible rules and packet black-holes. Invisible rules are rules with backups. Probing such rules results in success even though the backup-rule might have failed. Packet black-holes is often a result of a TCAM[7] deficit where packets with certain source addresses get dropped.

The flow tables in OpenFlow support pipeline processing. The new technique proposed in this framework builds a multi-rooted tree model of all rules found on a network device. The resulting model is a directed acyclic graph where each flow table represents a tier. Next, the framework uses the OFPIT_GOTO_TABLE instruction to direct packets to other sequentially numbered flow tables. A set of probes are then constructed for each network device using tree traversal. The desired network behaviour is computer before injecting probes into the network. The data plane is configured to export the probes after processing. A comparison can be made upon receipt. Using this model SERVE found an overall decrease in computing time necessary for state verification as seen in other papers such as [20] [23].

2.5.6 SDN traceroute

Tracing SDN forwarding without changing the network behaviour allows administrators to discover the forwarding behaviour of any ethernet packet and debug problems regarding both forwarding devices and applications [1]. The object of SDN traceroute [18] is to debug rule and forwarding problems in switch and controller logic by tracing network

[7] Ternary content-addressable memory (TCAM): high-speed memory capable of searching its entire content in a single clock cycle.

packet trajectories. The network paths, or trajectories, are measured by capturing the forwarding behaviour which occurs throughout the topology. The capturing is performed by trapping probing traffic pushed down to the data plane from the SDN controller. The be able to trap the probes sent from the controller, SDN traceroute first colours each switch using a graph colour algorithm and then inserts a small number of high-priority catch-rules onto the switches. The probing packets are specially crafted as they make use of the three 802.1p[8] priority bits in the 802.1q[9] tag in the frame header. This offers the possibility to trace layer-2 paths and thus extends the functionality of the original layer-3 traceroute. Due to the PCP field being 3-bits in size, this method is constrained to 2^k-1 colourable topology.

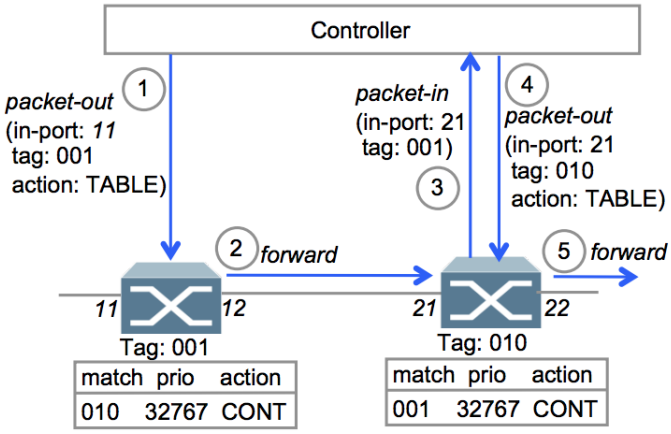

Figure 2.14: SDN Traceroute flow of operation [18]

2.5.7 VeriFlow

VeriFlow aims to provide verification capabilities of network-wide invariants in real-time. Violation of properties has a likelihood of occurring on most system. The tool tries to deal with faulty changes before they get applied by verifying the potential correctness of the network state after a proposed change. Being able to track the invariant state provides the means to block a change if the outcome of the change breaks the desired state. E.g. if inserting, modifying or remove rules breaks an invariant state like STP, which ensures no L2 loops are to occur, then stopping this change would be of interest.

VeriFlow operates by intercepting and verifying all rules before they are pushed out to the network devices. This is done by implementing a shim layer between the controller and the network. When a rule is obtained, the rule and its effect must be verified. Instead of checking the entire

[8]802.1p: 3-bit PCP field within the Ethernet frame header (class of service (CoS))
[9]802.1q: VLAN tagging (dot1q) of Ethernet frames.

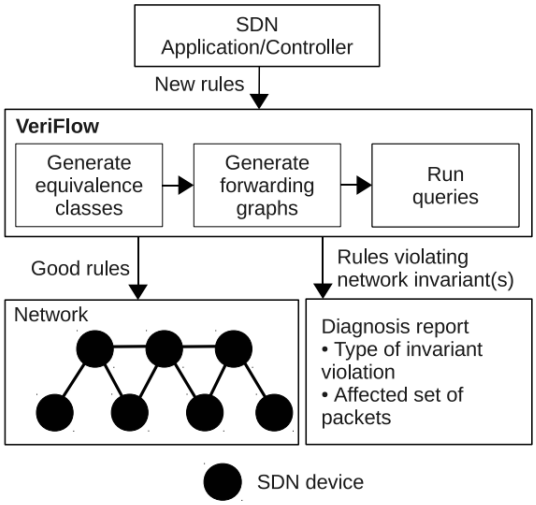

Figure 2.15: VeriFlow architecture [28]

network on every change like other research does [25–27], VeriFlow uses a different method. By first classifying what the rule targets alongside any overlapping existing rules, effectively slicing the network into classes where each class is a set of packets affected by the same forwarding actions throughout the network. Secondly, VeriFlow builds a forwarding graph for each class. Third, trapped rules are then traversed through the graph determining the status of invariants after each run. This way if mapping out affected areas makes VeriFlow able to verify changes to the overall states with good performance.

Chapter 3

Approach

This chapter outlines and explains the goals and design phase, the implementation phase and choices that were made in more detail. The design phase is characterized by a rather stepwise approach as the intended flow is logically broken down into smaller steps. Doing so highlights both the internal workflow as well as each component of the process and makes it easier to clarify the actions taken to answer the problem statement.

3.1 Objectives

The main objective is to make sure that the installed flow table entries, or rules, are working as intended. The design in section 3.2 below is proposed in order to achieve this, and is split into logical segments matching the overall and intended flow of operation.

Heller et al. [29] propose five checks for detecting mistranslation between state layers in SDN. This approach will confirm that the logical view is matching the device state and that we can answer some of their questions:

- Is rule *r* on switch *s* installed?
- Does rule *r* on switch *s* work as intended?
- Is the overall intended state matching the current network state?

By actively listening in on the control channel, there will be a lot of packets going back and forth between the controller and the data plane devices. The proposed approach should not interfere with unnecessary traffic i.e. should act as transparent as possible.

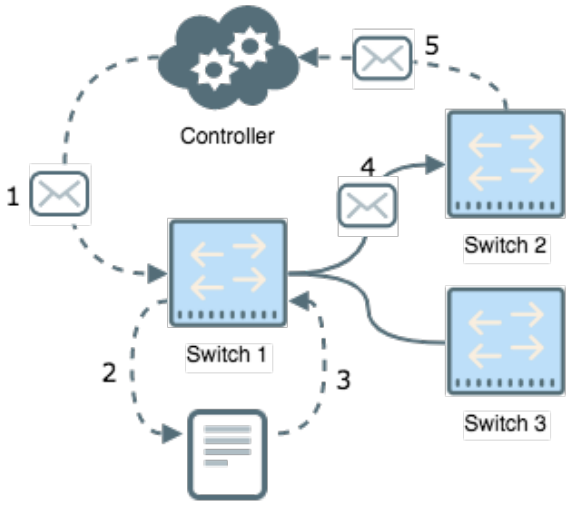

Figure 3.1: Verification flow in a simple topology

3.2 Design

All controller-to-switch and switch-to-controller information in an SDN enabled network is flowing on the control channel. Listening in on the packets being transmitted on this channel enables administrators to dynamically learn the topology, build information about the overall state or validate modifications to the data plane. The proposed solution in this project should be placed on the control channel in order to modify switch flow tables, output data plane probes and interact with events.

Probing is a technique, or process, of exploring or examining something. In this project data plane probing should be performed to determine the actual presence of rules in the flow table. Probing will further refer to the process of sending specially crafted packets, or probes, out from the SDN controller. Special catch-rules must be inserted onto the flow tables to pick up these probes and send them back to the controller. Figure 3.1 depicts the overall stepwise flow and what the initial supporting environment should look like.

1. A probe P matching flow entry F_n is sent out from the controller
2. Switch S_1 receives the probe P and performs a flow table lookup
3. The probe P matches with flow table entry F_n and is sent out port p_n
4. Switch S_2 receives probe P and performs a flow table lookup
5. The probe P matches with a catch rule and gets sent to controller
6. probe_in event is triggered and the incoming probe can be evaluated

The controller APIs[1] will be used to learn the topology, scrape flow tables

[1]Application Programming Interface (API): a general set of routines and protocols dictating communication between various components.

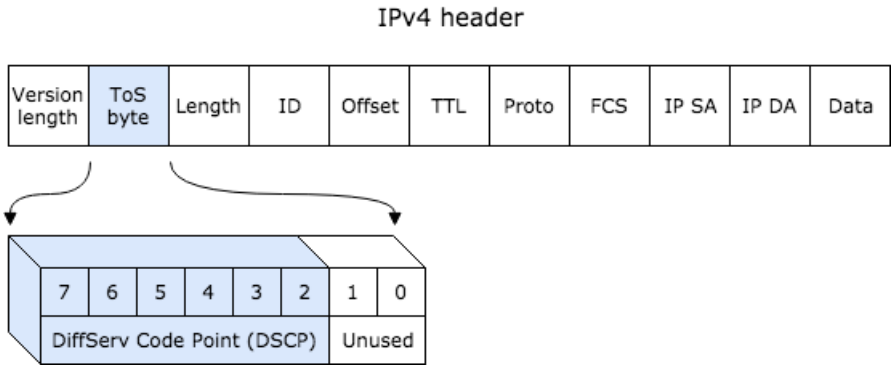

Figure 3.2: IPv4 DSCP field

off the switches and learn switch-to-switch links in order to map out the neighbourhood. Probe generation will be performed after the initial reconnaissance. By generating probes matching the flow table entries, they can exercise the rules in the data plane. The probes will be injected directly onto the targeted switch and should not depend on any device to travel via.

Catch-rules must be in place for the surrounding switches to pick up the injected probes. Both the catch rules and the probes must not conflict with production traffic and should have the highest priority in the flow table. In order to keep the number of catching rules as low as possible, a distinct value in the IPv4 header is chosen to match against. This allows for a single catch rule per neighbouring switch.

The distinct header value used for this task, figure 3.2, is placed in the 6-bit Differentiated Services Code Point (DSCP) [30] field in the 8-bit Differentiated Services field (DS) field residing in the 32-bit Internet Protocol version 4 (IPv4) header. The DSCP field was introduced with the intent of having extra bits for service discrimination and is used for network traffic classification and for providing quality of service (QoS) in modern IP networks.

The flow depicted in the aforementioned scenario 3.1 is defined for unicast[2] rules with basic forwarding as output action. There are various other types of actions and composition of rules. One type of rule in which ought to be tested in this project is drop rules. Drop rules are harder to test because they do not forward any packets and thus do not provide any output. In OpenFlow drop rules are defined either with an instruction of OFPIT_CLEAR_ACTIONS or an empty action list, distinguishing them from forwarding rules.

Data plane rules can be installed by those who are given access to the controller APIs. In order to avoid flaws in rule logic businesses might build security policies or unit testing surrounding the rule install process. Businesses should further strive to inherit either some form of oversight

[2]Unicast: communication between a single sender and a single receiver.

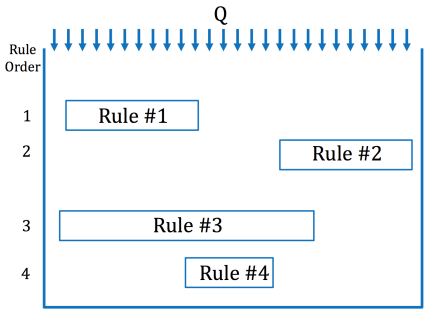

Figure 3.3: Shadow rule [31]

or dynamic testing and verification of rules in order to validate their correctness. One example of a logical flaw would be to append a rule which in effect is routing packets back where they came from. Such a scenario would result in a direct forwarding loop and should rather alert administrators than be installed.

A shadow rule is a rule which is totally overshadowed by another, "broader" rule with higher priority. Rule number 4 as shown in the raining 2D-box figure 3.3. This task assumes that the flow tables do not hold any shadow rules. Packets destined for such a rule will never hit the intended rule as they are caught by a higher priority rule instead. There are various ways to detect [32], clean up or steer away from such rules. They can be cleaned up by setting the idle_timeout option when inserting a rule, detected with offline parsing methods [31] or caught by combining parsing techniques with active monitoring of the control channel.

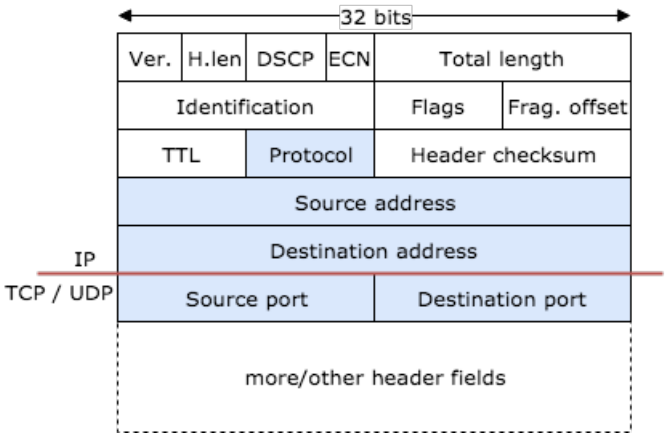

Figure 3.4: L3 and L4 values comprising the 5-tuple set

Performing packet classification at physical link speed is the basic operation of many networking devices. Network devices usually compare header fields for every incoming packet against a set of rules once they arrive. The packet classification optimization problem is not a new problem [33, 34], and is one which is continuously under improvement e.g. to keep up with increasing link speeds or the adoption of IPv6. This project

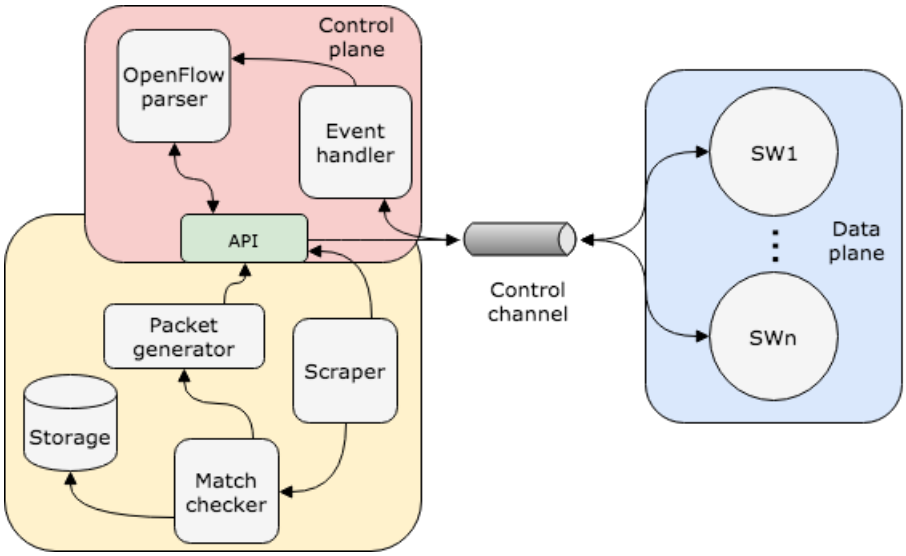

Figure 3.5: Architectural flowchart diagram

should strive to use as real data sets as possible so they mimic real-life traffic patterns.

An N-tuple is a collection of attributes or values. Together these attributes define certain requirements for matching on various network rules or access control lists (ACLs). The packet generation process in this task aims to satisfy a 5-tuple set comprised of five different values making up a Transmission Control Protocol/Internet Protocol (TCP/IP) connection. The values included in the tuple is source and destination IP address, source and destination port number and the protocol in use, see figure 3.4.

Figure 3.5 depicts the supporting flow for testing rules in the data plane. The flow starts at the scraper pulling flow tables and information of interest off of switches residing in the data plane. The information is handed to the matcher whose purpose is to find fields used to make a distinct probe for each entry in the flow table which – matching only with that entry. The packet along with the matching entry is stored for later analysis. The outgoing packet must be built, serialized and encapsulated in an OpenFlow message before being sent to the switch over the control channel. The API builds OpenFlow format messages, provides access to various imported modules and the northbound REST API.

3.3 Implementation

The implementation of the underlying infrastructure will be hosted in a virtualized environment using VirtualBox[3]. The main reason for this decision is to have the possibility to scale out any topology if needed, to contain and control the virtualised network or reprovisioning the setup in a rather effortless manner.

3.3.1 Topology

The topology in figure 3.1 can be thought of as a snippet of a larger network. This snippet is a rather simple one but will work as a proof-of-concept. There are three switches needed to make up the network.

Mininet is used to implement the main supporting topology seen in figure 3.1 which is used throughout the project. Expanding on this proof-of-concept topology to mimic a somewhat real-looking topology for more realistic testing or for benchmarking purposes is a matter of adding and mixing switches and links in the Mininet definition file, listing 3.1.

```
1 from mininet.topo import Topo
2 class simple( Topo ):
3     def __init__( self ):
4         Topo.__init__( self )
5         switch1 = self.addSwitch( 'switch1' )   # add switch
6         switch2 = self.addSwitch( 'switch2' )   # add switch
7         switch3 = self.addSwitch( 'switch3' )   # add switch
8         self.addLink( switch1 , switch2 )       # add link
9         self.addLink( switch1 , switch3 )       # add link
```

Listing 3.1: Creating a minimal topology in Mininet

When executing Mininet, listing 3.2, the topology file and name is sent as an argument. Other arguments are the switch type and protocol version, "–mac" to indicate easy-to-read MAC addresses, the IP address of the remote controller and "-x" for spawning XTERMs for each node.

```
1 mininet@mininet-vm:~$ sudo mn --custom topology.py --topo=simple
      --switch=ovsk,protocols=OpenFlow13 --mac
      --controller=remote,ip=<controller ip> -x
```

Listing 3.2: Executing Mininet

[3]VirtualBox: free and open source hypervisor developed by Oracle Corp.

3.3.2 Controller

On the controller, the Ryu-manager is both the main executable and the foundational component. Every Ryu-application inherits from the app-class, are single threaded and implements various functionality. The underlying architecture is depicted in figure 3.6. When executing Ryu different flags and parameters can be specified: # *ryu-manager –flag <path to configuration file>* and # *ryu-manager [generic app]*. These options can be combined and multiple flags and apps can be started at the same time. When the controller runs it listens for incoming connections. Output listing 3.3 shows the controller by default listening on 0.0.0.0 (any) and port 6633/tcp.

```
> lsof -i:6633
COMMAND    PID  USER   FD   TYPE           DEVICE SIZE/OFF NODE NAME
python3.6 91733 user  7u  IPv4 0x76d877116010c51    0t0  TCP *:6633
    (LISTEN)
```

Listing 3.3: lsof

Any OpenFlow capable switch (hardware enabled, OVS or Open vSwitch) can now initiate a connection to the controller. Event management and message handling are the core components of the Ryu architecture seen in figure 2.3. Listing 3.4 shows the application instantiating as a subclass of ryu.base.app_manager.RyuApp.

```
# Ryuapp subclass
class ctrlapp(app_manager.RyuApp):
    # Set the OpenFlow version to be used
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    # Constructor
    def __init__(self, *args, **kwargs):
        super(ctrlapp, self).__init__(*args, **kwargs)
        // vars
```

Listing 3.4: Ryu application class

Enabling –verbose mode on the controller we get detailed information when switches, or datapaths, connects. See listing 3.5. After the socket is created we see the switch sends an OFPT_HELLO message to the controller. Both parties send this message, which holds the highest version number support by each side. The control channel supports encrypted communication with TLS but this is not used in this project. The message triggers an event and initiates the session setup. The session gets established after the EventOFPSwitchFeatures event is triggered as a response to the outgoing OFPT_FEATURES_REQUEST from the controller to the switch.

```
connected socket:<eventlet.greenio.base.GreenSocket object at
    0x10818ab38> address:('192.168.56.102', 38020)
hello ev <ryu.controller.ofp_event.EventOFPHello object at
    0x1081a0198>
move onto config mode
EVENT ofp_event->ofctl_service EventOFPSwitchFeatures
switch features ev
    version=0x4,msg_type=0x6,msg_len=0x20,xid=0x18073227,
    OFPSwitchFeatures(auxiliary_id=0,capabilities=71,
    datapath_id=1,n_buffers=256,n_tables=254)
add dpid 1 datapath <ryu.controller.controller.Datapath object at
    0x10818aa58> new_info <ryu.app.ofctl.service._SwitchInfo object
    at 0x1064fd8d0> old_info None
```

Listing 3.5: Connecting switches to the controller

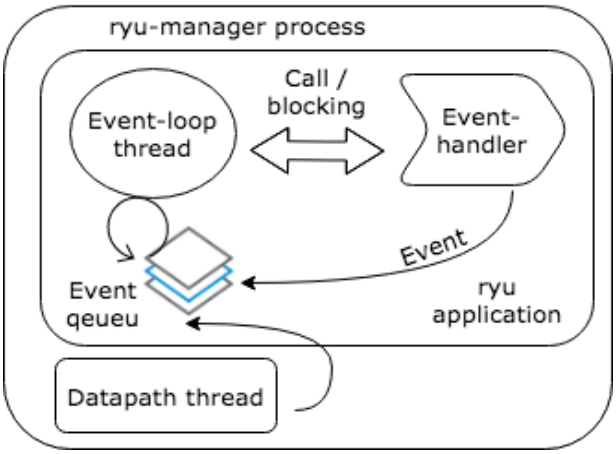

Figure 3.6: Ryu application workflow [35]

3.3.3 Gathering

The ofctl_rest module provides a REST API for gathering information, metrics and statistics from datapath devices connected to the controller. The endpoints provided by the API outputs information formatted as JSON. Calling /stats/flow/<datapath_id> returns the flow table for a given datapath. Listing 3.6 displays information of interest such as the priority value, action list and match field. Building atop the modules and APIs provided by Ryu are helper functions used in order to gather the necessary information like datapaths, flow tables, links and ports. Support functions built around the APIs are listed in C.1

```
{'priority': 14823, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match':
    {'in_port': 3, 'dl_type': 2048, 'nw_dst':
    '172.26.84.0/255.255.255.0', 'nw_proto': 6, 'tp_dst': 53}}
```

Listing 3.6: Flow table entry

The Ryu controller provides various APIs and modules by default. Switch and link discovery are imported with ryu.topology.api. Link discovery returns a link class object per datapath-to-datapath link or a list holding all objects from the topology. Link discovery uses Link Layer Discovery Protocol (LLDP)[4] and is enabled using the –observe-links parameter when running Ryu. From the LLDP information the controller learns what resides on each side of a switch link such as system name, capabilities, addressing, port information and other optional TLVs which might be advertised. Port information and addressing are passed along with the link object. The neighbourhood for each switch is mapped building on top of the information gathered by the link objects. The function in listing 3.7 returns a list of all neighbouring datapaths.

```
def getDatapathNeighbors(self, dpid):
    neigh = list()
    for link in self.getLinksByDatapathID(dpid):
        for k, v in link.to_dict().items():
            if k is 'dst':
                neigh.append(v['dpid'])
    return neigh
```

Listing 3.7: Getting all neighbours for a given datapath

3.3.4 Flow matching

The flow matching and packet generation is performed after the flow tables are gathered. The overall purpose of the matching process is to generate a distinct probe which is only matching against the targeted rule, or entry. As shown in appendix A.1 there are numerous fields to match on in OpenFlow version 1.3. In this task, we are focusing on the fields comprising the 5-tuple.

Rules can overlap on one or more fields used for matching or classification. If two rules have a soft overlap, a probe-packet should match with only the rule in question and not on any portion of any interfering rules. As depicted in figure 3.7, a probe matching on Rule3 must not hold values in the space overlapped by Rule1 and Rule2. With multiple match fields the classification process is a difficult problem [36]. This project does not focus on optimizing this match problem as that is a separate research area in itself [33, 34, 37]. Therefore, the matching algorithm used will be of a

[4]LLDP: vendor-neutral link-layer protocol for devices to advertise information

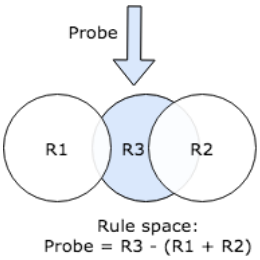

Figure 3.7: Venn diagram depicting overlapping rules

linear nature; sorting the flow table descending by priority and sequentially comparing each rule with a priority equal or higher than the rule to match on. It is a rather simple way of iterating through all entries, but also a poor one as it does not scale very well [36]. The time to classify a match grows linearly with the number of rules in the table. Such a difficulty has a complexity of O(n).

Some slight optimization is introduced into the matching process. By first comparing single-value fields like source and destination ports in the layer 4 headers we can quickly discard non-matching tuples. The same goes for the IP protocol field OXM_OF_IP_PROTO. If the values are not equal, we can move to the next rule in the flow table. To speed up the matching process a bit further, the OXM_OF_IN_PORT field is used in the matching process as well, as two identical rules with different OXM_OF_IN_PORT will not conflict with each other – if both values are set, and is not any.

As seen in table 3.1 the OXM_OF_IPV4_SRC and OXM_OF_IPV4_DST fields hold one of three values. Network ranges are represented with a CIDR notation[5], single addresses is treated as a /32 subnet and any-values is treated as None[6] objects. The netaddr[7] library provides set operations like union, intersection and slice operations to IP portions and subnets.

Match fields	Combinations
IPv4 source address	single, range or any
IPv4 destination address	single, range or any
IP protocol	int or any
TCP/UDP source port	single or any
TCP/UDP destination port	single or any
Total	72

Table 3.1: 5-tuple value combination

[5]CIDR: a compact way to represent an IP address along with its associated subnet mask.
[6]None: denoting an object who lacks value.
[7]netaddr: Python library for representing and manipulating network addresses.

3.3.5 Install/remove

OFPT_FLOW_MOD is used for modification of flow tables residing in the data plane. We are using the message to set up and tear down catch rules. The properties of the OFPT_FLOW_MOD message is listed in appendix A.1. In the listing 3.8 example we are inserting a new rule, or flow, indicated by the ofp.OFPFC_ADD parameter in line 11. The flow modification is mainly comprised of a match object (line 6), an action list (line 7) and an instruction list (line 8). The match object holds the OXM fields A.1 to match incoming packets on. The action specified in this scenario sets the output action for any matching packet to send the packet up to the controller. The list of instructions in line 8 tells the switch to apply the list of actions that are passed along.

```
1  def addCatchRule(self, datapath):
2      ofp        = datapath.ofproto
3      ofp_parser = datapath.ofproto_parser
4      priority   = 65535
5      buffer_id  = ofp.OFP_NO_BUFFER
6      match      = ofp_parser.OFPMatch(eth_type = 2048, ip_dscp = 1)
7      actions    = [ofp_parser.OFPActionOutput(ofp.OFPP_CONTROLLER)]
8      inst       = [ofp_parser.OFPInstructionActions(
9                      ofp.OFPIT_APPLY_ACTIONS, actions)]
10     req        = ofp_parser.OFPFlowMod(datapath, 0, 0, 0,
11                     ofp.OFPFC_ADD, 0, 0, priority, buffer_id,
12                     ofp.OFPP_ANY, ofp.OFPG_ANY, 0, match, inst)
13     datapath.send_msg(req)
```

Listing 3.8: Installing a high priority catch rule

When removing catch rules the fields are the same except for the OFPFlowMod object. The removeCatchRule(self, datapath) function has the value of the flow modification command A.2 set to 3 indicating an OFPFC_DELETE along with a flow mod flag A.3 value set to 1 indicating OFPFF_SEND_FLOW_REM. The command specifies that any matching flow table entry ought to be deleted and the flag tells the switch to send a descriptive flow removed message back to the controller.

3.3.6 Inject

An SDN controller can inject packets onto the data plane of any given switch by using the OFP_PACKET_OUT message structure described in figure 2.2. Ryu provides a packet library for building and serializing[8] various protocol packets which is imported in line 2 in listing 3.9. Building the packet seen in the listing we imply an ethernet header with the ethertype value ETH_TYPE_IP is already set and added to the packet.

[8]Serialization: the process of translating an object to a transmittable format.

Values for the IPv4 and TCP headers are set in line 4-5, the protocols are added to the packet in line 8-9 and in line 10 the packet is serialized.

```
1  from ryu.ofproto import ether
2  from ryu.lib.packet import packet, ipv4, tcp
3
4  ipheader = ipv4.ipv4(proto=6, src='172.16.20.1', dst='80.93.102.11')
5  tcpheader = tcp.tcp(src_port=45203, dst_pst=80)
6
7  pkt = packet.Packet()
8  pkt.add_protocol(ipheader)
9  pkt.add_protocol(tcpheader)
10 pkt.serialize()
```

Listing 3.9: Building a packet

The function in listing 3.10 takes the datapath, or switch, object as input along with the packet to be sent and the port the switch should treat the packet as coming in on. Setting the action output of OFPP_TABLE tells the datapath to submit the packet to the first flow table for processing through the existing flow table entries. Setting the buffer_id to OFP_NO_BUFFER indicates that the outgoing OpenFlow packet data is included and encapsulated in the data array.

```
1  def sendMsg(self, datapath, pkt=None, in_port=None):
2      ofp        = datapath.ofproto
3      ofp_parser = datapath.ofproto_parser
4      buffer_id  = ofp.OFP_NO_BUFFER
5      pkt        = pkt if pkt is not None else generate.testPacket()
6      actions    = [ofp_parser.OFPActionOutput(ofp.OFPP_TABLE)]
7      in_port    = in_port if in_port is not None else ofp.OFPP_ANY
8      datapath.send_msg(ofp_parser.OFPPacketOut(datapath,
9                        buffer_id, in_port, actions, pkt))
```

Listing 3.10: PacketOut from controller

3.3.7 Catch

When a packet matches with the catch rule, the OFPActionOutput specified sends the packet to the controller (OFPP_CONTROLLER). Upon receipt, a OFP_PACKET_IN_REASON message holding a value of 1 (OFPR_ACTION) indicating the switch output explicitly sent the packet to the controller. The *ev.msg.data* object holds the incoming packet and its header objects.

Upon receipt, the packet can be parsed (listing 3.11). A distinct DSCP value for probing purposes was declared and set earlier in the task. Parsing the packet headers looking for this value will determine if a received packet

is in fact a probing packet. The msg object holds information like the packet_in information, a protocol parser and the incoming packet. See figure 2.1 for the data structure.

```
msg = ev.msg                    # packet_in object
dp = msg.datapath               # switch object
ofp = dp.ofproto                # OF proto object
parser = dp.ofproto_parser      # OF parser object
pkt = packet.Packet(msg.data) # on-wire packet
```

Listing 3.11: Parsing the incoming probe

Listing 3.12 shows the event handler listening for EventOFPPacketIn events. The event holds the incoming message/packet and the reason for why the message was received. Upon receiving a probe we can loop the list finding the matching entry.

```
import ryu.controller.ofp_event
from ryu.controller.handler import set_ev_cls

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packetIn(self, event):
    if event.msg.reason == ofp.OFPR_ACTION:
        # ft entry triggered the packetin
            if pkt.get_protocols(ipv4.ipv4)[0].tos is 4:
                # test probe received, evaluate
```

Listing 3.12: Event handler on the controller

Outgoing packets are stored in a first in, first out (FIFO, figure 3.8) queue along with the entry they are matching, the ID of the switch in which the entry resides, the in_port (if set), a boolean value determining if the entry is a drop rule and the neighbouring switch on the other side of the outgoing port. The outgoing packets are sent in sequence, and thus received in sequence. If for some reason the packets should change order, the queue (or list) can be iterated looking for the incoming packet and the given entry can be popped from the list. Probing packets will not be lost due to drop rules in this proposed solution, which is covered more in detail in the experiments section.

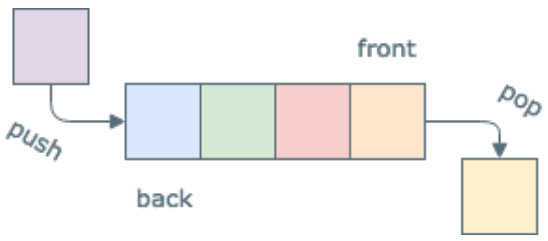

Figure 3.8: Storing packets in a FIFO qeueu

Storing this information eases the evaluation process once the packet is received. If an entry has an action of CLEAR_ACTIONS or None, the rule is a drop rule. Storing the neighbour ID will let us find a direct forwarding loop. As the OXM_OF_IN_PORT can hold an ANY value and loop the packet it is not sufficient to look for identical in_ports and outgoing ports.

Chapter 4

Experiments & results

Various experiments are discussed, outlined and performed in this chapter. The different scenarios are explained along with their intended goal. The proposed solution sits on the control channel listening in on all OpenFlow messages flowing in and out from the controller and the data plane devices – utilising the channel for communication purposes.

4.1 Rule generation

This project strives to generate as real-looking rulesets as possible to ensure the quality of the generated rules when performing the IPv4 five-tuple exact-match classification on IP packets. Having access to datasets containing OpenFlow rules pulled from production environments is hard due to the limited amount of real-life deployments [1]. Classbench [36] has been used in various publications [5, 20, 23] to generate different sets of rules and its successor Classbench-ng [37] has continued the development of this tool.

In this project, we use our own generator. Instead of using non-OpenFlow seeds [36] to generate rules, translating and parsing the outcome to match with OpenFlow syntax [23]. The synthetic OpenFlow based IPv4 rulesets randomly generated are based on RFC 1918[1] addresses and draws subnets and values based on probabilities derived from figure 15 in [37]. The generator looks at Classbench for inspiration when distributing the values making up every rule. Figure 4.1 shows the distribution of prefixes pulled from 30 individual example-runs each of 3000 rules on one switch.

The chance of drawing strict layer 3 rules is 70% and a mix of layer 3 and layer 4 is 30%. With strict layer 3 rules, the layer 4 portions are wildcarded. In the rulesets used [37] almost every instance of layer 4 rules are the combination of wildcard + exact match (WC-EM). An example of this would be a rule accepting connections from any source port destined for

[1] RFC 1918: range of non-routable IP addresses for internal use.

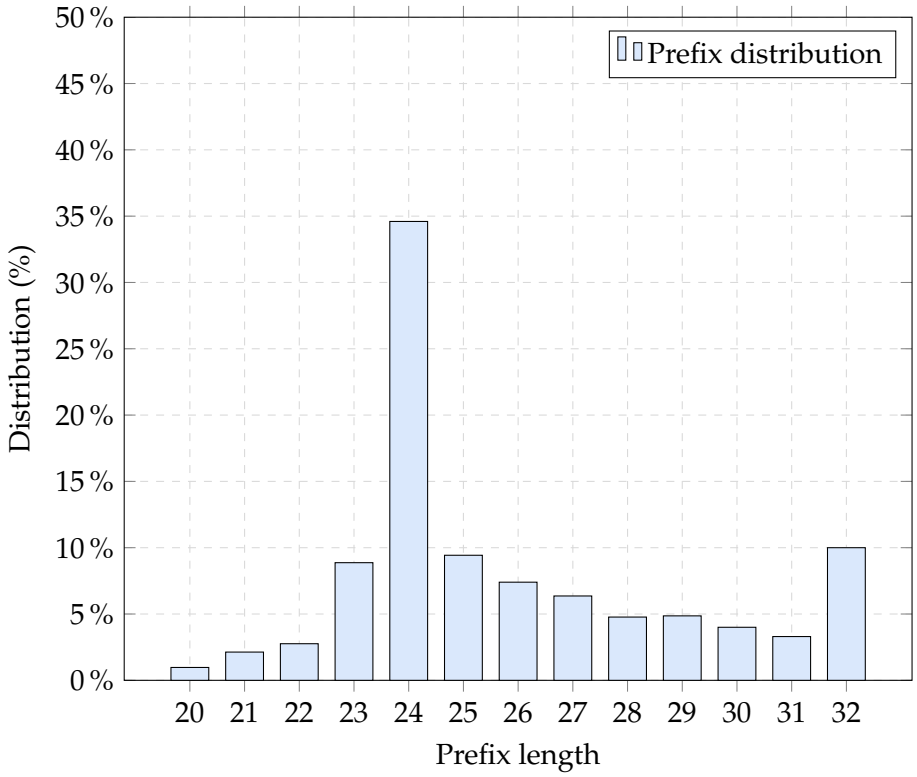

Figure 4.1: Match field value distribution.

port 80. The layer 4 source and destination ports in the generator are either drawn from a pool of the most used well-known port numbers or will draw an ephemeral[2] port number mimicking an operating system/client-side generated port.

The stacked chart in figure 4.2 shows the distribution of OpenFlow OXM fields making up the entries. The distribution of the values varies per run as they are randomly generated. The per-field distribution of header fields mimics that of Classbench-ng where rules are created using a combination of the of1 and of2 rule sets pulled from a live datacenter production environment [37].

4.2 Matching

A common type of packet classification examines the packet headers fields making up the standard IP 5-tuple (figure 3.4). An example would be 192.168.1.17/50272/63.49.123.3/80/6 for a packet coming from port 50272 of IP 192.168.1.17, destined for port 80 of IP 63.49.123.3 using IP protocol 6, which is TCP. For UDP this number would be 17. These five fields make for a lot of unique flow table entries, but the fields can also be wildcarded.

[2]Ephemeral port: short-lived port number dynamically assigned by the OS

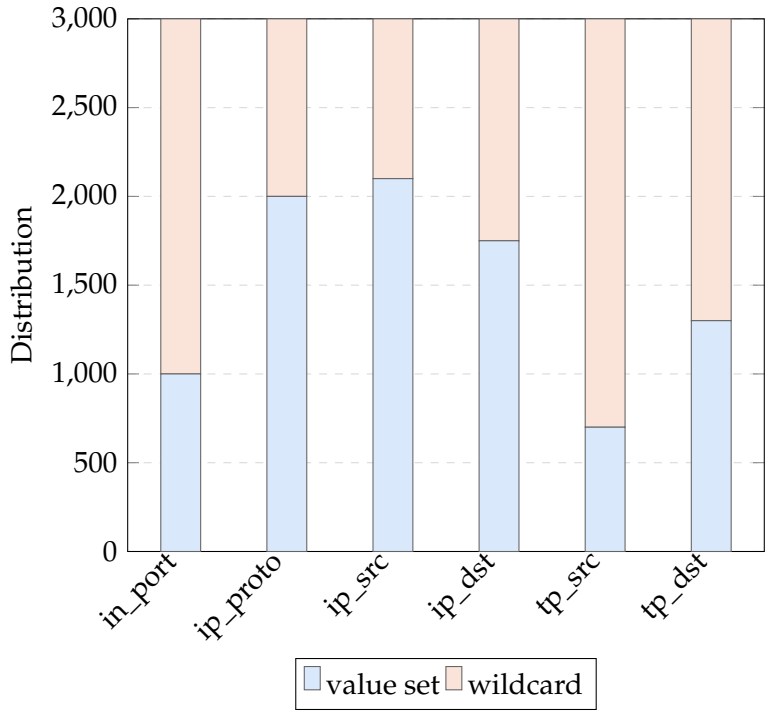

Figure 4.2: Match field value/header distribution.

A wildcard means that they can span, or overlap, with other fields in other rules currently sitting in the flow table. This is depicted in figure 4.3 where two rules partially overlapping the bottom rule. The complexity of rule matching problem increased with OpenFlow as the new protocol introduced more fields to match on. With OpenFlow v1.0 the number of fields where seven, and with version 1.3 there more than 40 fields. Having ranges in the IPv4 fields adds some complexity to the matching process. The process takes previous entries into account to steer away from corner cases in the flow table.

Figure 4.3 shows three rules. Rule1 with an IPv4 network address of

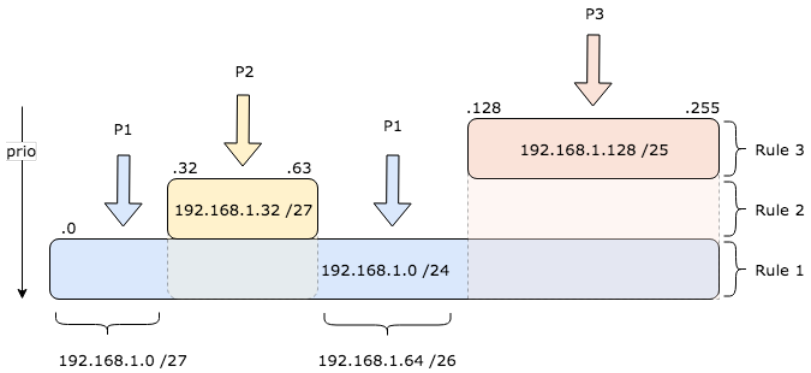

Figure 4.3: Overlapping rule

192.168.1.0 and a range of 255.255.255.0, or /24. Rule2 with an IPv4 network address of 192.168.1.32 and a range of 255.255.255.224, or /27. Rule3 with an IPv4 network address of 192.168.1.128 and a range of 255.255.255.128, or /25. Looking at the ranges we see that Rule1 is partially overlapped by Rule2 and Rule3. In order to generate a distinct probe P1 for Rule1 the subnets of the overlapping rules R1 and R2 must be subtracted. The IP portions left after the subtraction will hold values for generating distinct probes. Listing 4.1 shows a flowable dump from Switch1 in Mininet with the same three rules as in figure 4.3 installed.

```
1 OFPST_FLOW reply (OF1.3) (xid=0x2):
2 cookie=0x309, duration=132.07s, table=0, n_packets=0, n_bytes=0,
    priority=3,ip nw_src=192.168.1.128/25 actions=output:2
3 cookie=0x29a, duration=123.404s, table=0, n_packets=0, n_bytes=0,
    priority=2,ip nw_src=192.168.1.32/27 actions=output:2
4 cookie=0x22b, duration=114.619s, table=0, n_packets=0, n_bytes=0,
    priority=1,ip nw_src=192.168.1.0/24 actions=output:3
```

Listing 4.1: flow dump

Listing 4.2 shows output from the iterations performed internally by the matching algorithm when finding an IPv4 source address range left over after subtracting the portions from Rule2 and Rule3. The addresses are converted to netaddr.IPSet objects and Rule1 equals the subtraction of Rule2 and Rule3. The result is the non-overlapping IP subnets remaining, if any. The output matches with figure 4.3. Further, the result of 192.168.1.0/24 subtracted by 192.168.1.0/24 is zero, and non-overlapping subnets have no effect on each other.

```
1 ==========   init    ==========
2 match addr:    IPSet(['192.168.1.0/24'])
3 ==========   start   ==========
4 entry addr:    IPSet(['192.168.1.128/25'])
5 (new) match:    IPSet(['192.168.1.0/25'])
6 ========== iteration ==========
7 entry addr:    IPSet(['192.168.1.32/27'])
8 (new) match:    IPSet(['192.168.1.0/27', '192.168.1.64/26'])
9 ==========   done    ==========
```

Listing 4.2: output packet gen

Packets are sent out in sequence. Upon the first glance, they are also received in sequence because of the event queue (figure 3.6). The initial design stored objects using a traditional first in, first out (FIFO) qeueu as seen in figure 3.8. Making it rather quick to pop the first element. This is not a reliable method. If for any reason a packet gets delayed and another packet makes it to the queue first, the order would get disturbed and the wrong item will be popped from the list. Moving the design of the object storage to a list of dictionaries, figure 4.4. The dictionaries store

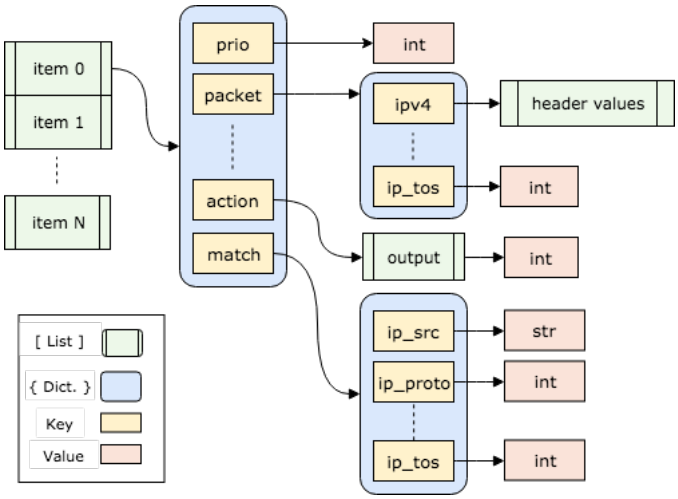

Figure 4.4: Short term storage structure.

the outgoing packets along with the flow table entry the packet belongs to, priority and cookie value and a boolean value marking the rule as a drop rule. Upon a packet_in event for a DSCP marked probe, the list is iterated, comparisons are made and the element gets popped if a match occurs.

The initial intended comparison was based upon a streamlined way of directly comparing header values. This proved not doable even when objects hold equal values (listing B.1. This forced the matching process to iterate through the header values checking multiple fields and conditions like comparing IP ranges, ip_proto values and layer 4 ports. Given that the whole tuple must match, the process should keep a record of earlier near-full matches to steer away from in other almost-match cases.

4.3 Benchmarking

The hardware used for hosting the topology and running the controller software is listed in appendix B.4. In short; Mininet runs on a single 2.20 GHz core and 1GB of memory and Ryu runs on four 2.20 GHz cores with two threads each and 16GB of memory.

The numbers in appendix 4.1 show some slight variance in the results. Looking at the average alone does not necessarily point out how precise a result is. The 95% confidence interval will assist us in pointing out the accuracy of a test. Figure 4.5 shows the total time to generate matching probes for all entries/rules in a flow table. The probe generation has been run on flow tables holding from 100 to 3.000 entries. The tests were performed 30 times (CLT) for each x-value, or step. The graph in the figure 4.5 shows the average time whereas the whiskers at each point show the confidence interval per measurement.

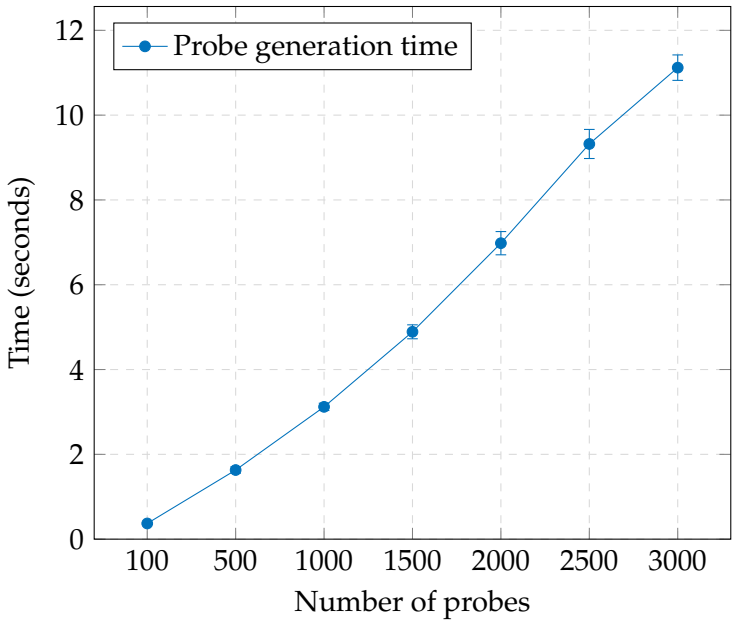

Figure 4.5: Time to generate distinct probes.

The outcome was rather expected as the matching algorithm is linear. There are other ways of improving the match time [36]. Depending on the type and dynamic nature of the network, probing the entire flow table of every switch in the topology is most likely a rare effort [20]. This project does not focus on improving the match time. Table 4.2 shows the cost of making HTTP GET calls to the controller's REST API. This call is made once during the packet generation and is not subtracted from the results seen plotted in figure 4.5.

# of rules	100	500	1000	1500	2000	2500	3000
Average (sec)	0.06	0.29	0.57	0.87	1.15	1.45	1.7

Table 4.2: Time to REST

Num. of packets	Average (sec)	CI (95%)	Min	Max
100	0.3649127	± 0.018412	0.29905	0.55192
500	1.626598	± 0.073901	1.36472	2.06207
1000	3.138552	± 0.086444	2.72592	3.78173
1500	4.890123	± 0.164425	4.39502	6.01276
2000	6.980074	± 0.274075	6.12678	8.72343
2500	9.315304	± 0.342894	7.90635	11.36782
3000	11.51869	± 0.300030	10.12929	13.44892

Table 4.1: Packet generation

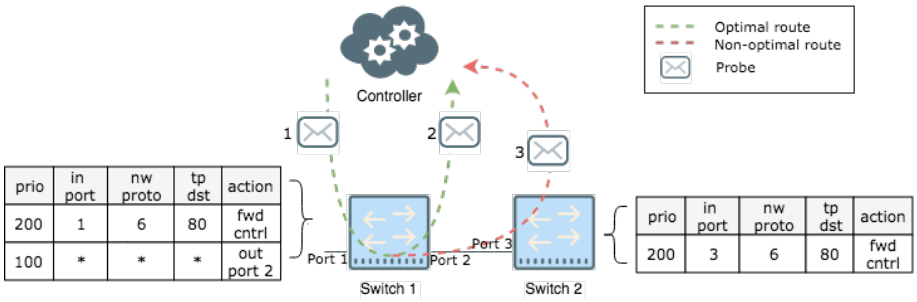

Figure 4.6: Verifying OFPP_TABLE

4.4 OFPP_TABLE

Research such as [8] and [20] dedicates a neighbouring switch in the topology as an entry point, or injection point, to send probing packets via. In this project, we argue for using the OFPP_TABLE reserved port as the specified action when sending OFPT_PACKET_OUT to inject probes directly to the targeted switch. Setting this action value submits the packet to the first flow table, table 0, of the receiving switch. The reserved port value equals to 0xFFFFFFF9.

Figure 4.6 shows the intended flow for verification. If the outgoing packet (1) takes the green coloured route (2), it means that the action value was correctly interpreted. If the packet (1) takes the red coloured route (3), it means that the action value was incorrectly interpreted.

The flowable rules used when testing this is shown in figure 4.6. The first rule on Switch1 forwards any 80/tcp packets coming in from port 1 to the controller. Any other packets coming in on any of the switchports hits the second rule, with the priority of 100, and gets sent off to Switch2. Switch2 holds just one rule catching 80/tcp packets and sends them to the controller. When the probe hits Switch1 the outcome will either send the probe up to the controller or pass it along to Switch2. An OFPT_PACKET_IN with a reason:action will be sent to the controller from Switch1 if OFPP_TABLE works correctly. If the reserved port does not work as intended, the packet will be received from the neighbouring Switch2. The outgoing probe is listed in listing 4.3.

```
1 OFPPacketOut(actions=[OFPActionOutput(len=16, max_len=65509,
    port=4294967289, type=0)], actions_len=0, buffer_id=4294967295,
    data=ethernet(dst='de:ad:be:ef', ethertype=2048,
    src='de:ad:be:ef'), ipv4(csum=0, dst='2.2.2.2', flags=0,
    header_length=5, identification=0, offset=0, option=None,
    proto=0, src='1.1.1.1', tos=0, total_length=0,
    ttl=255,version=4), tcp(ack=0,bits=0, csum=0, dst_port='80',
    offset=0, option=None,seq=0, src_port='45921',urgent=0,
    window_size=0), in_port=4294967295)
```

Listing 4.3: Outgoing packet

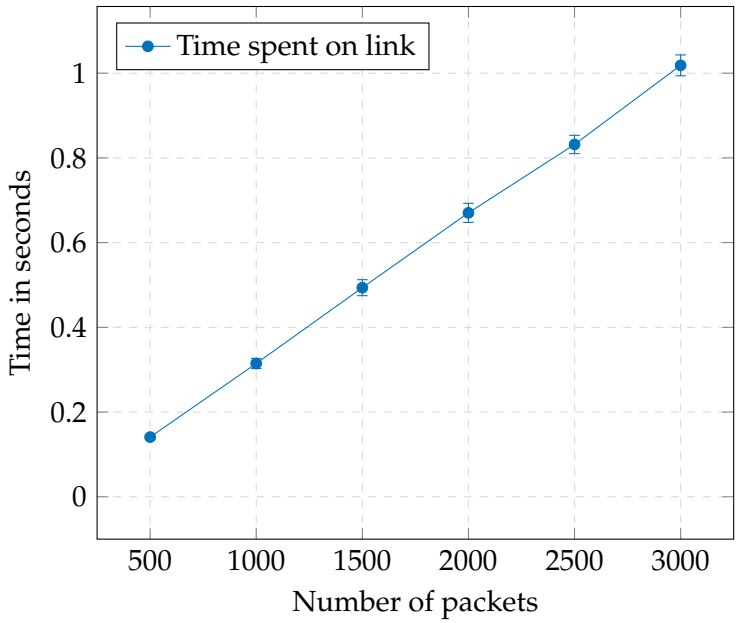

Figure 4.7: Time spent on link.

Num. of packets	Average (sec)	CI (95%)	Min	Max
500	0.1409040	± 0.0021330	0.13129	0.15249
1000	0.3146657	± 0.0044824	0.29958	0.34143
1500	0.4936147	± 0.0070510	0.46892	0.53054
2000	0.6702727	± 0.0084099	0.63484	0.72687
2500	0.8317127	± 0.0080684	0.80361	0.89194
3000	1.0184710	± 0.0091940	0.96698	1.06912

Table 4.3: Time spent on link

Appendix B.1 shows the outgoing OpenFlow packet in Wireshark and appendix B.2 shows the incoming packet. The packet dumps show the encapsulated OpenFlow packets sitting atop the TCP segment. From the packet dump, we see the outgoing packet with reserved the port set and the sequence and acknowledgement numbers shows the resulting packet_in after the outgoing packet hit the catch rule and gets sent back up to the controller.

Figure 4.7 shows the time spent sending and receiving probes i.e. the amount of time packets spend on the wire, in seconds, when testing OFPP_TABLE. Table 4.3 shows the mean and confidence interval from tests performed. The whiskers on the plot show the confidence interval. The observed results are rather stable with a quite low amount of fluctuation.

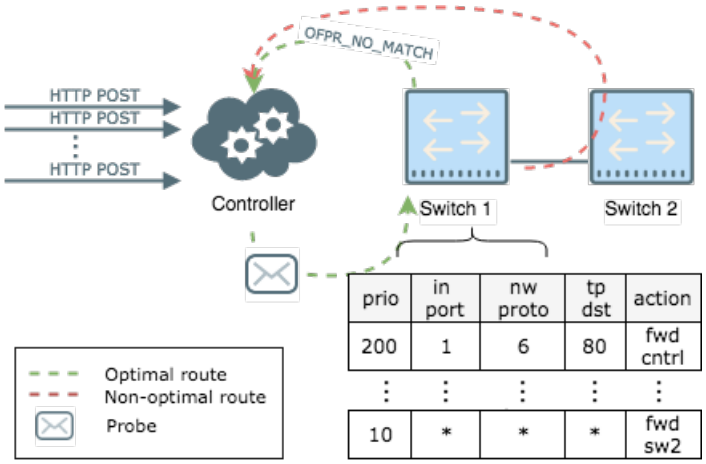

Figure 4.8: False positive testing

4.5 False positive

A false positive, or false alarm, occurs when a test receives a positive result but should actually have received a negative result. To rule out any false positives, or type I errors, from our tests we inject an error into the flow table N number of times during the tests.

The underlying flow used when looking for any false negatives are almost the same as seen earlier. What differs is the fault injection. To simulate a rule install error we remove a rule at different places in the flow table multiple times during runtime. During this test, the probe will either hit a different, lower priority rule and get forwarded out towards a neighbour, or hit the bottom of the flow table and a rule miss in the form of OFPR_NO_MATCH will occur. Either way, the outcome is not the expected outcome from the initial flowable point of view and a faulty rule is found.

The linear dual-switch topology as seen in figure 4.8 is used in this test. Switch1 will be randomly populated with N number of rules per test. Following the same logic as when testing for false negatives, the rules to be removed resides at the top of the flow table, at the bottom of the flowable and somewhere in between e.g. in the middle of the flow table. Rules are removed using the /stats/flowentry/delete REST endpoint. The output of the tests is found in appendix B.6.

4.6 Drop rule

Drop rules can be distinguished from other types of rules by looking at the set of actions of each individual flow table entry. In OpenFlow a drop rule is recognized by an empty action list or that the action is explicit set to OFPIT_CLEAR_ACTIONS.

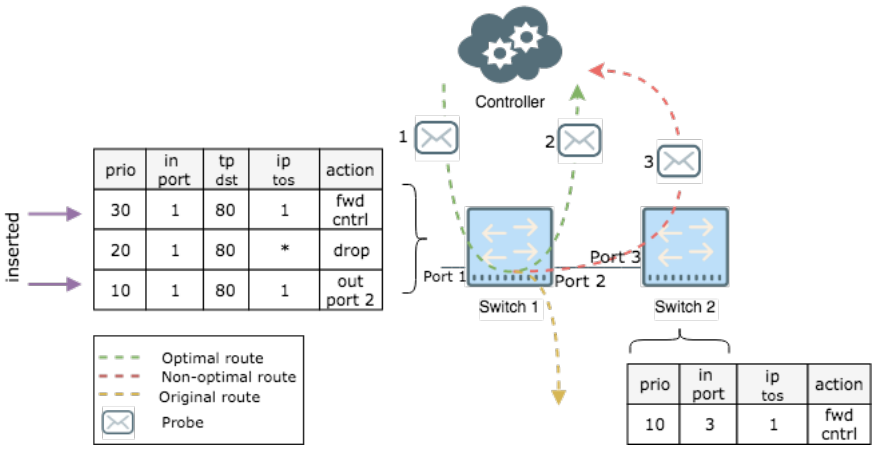

Figure 4.9: Testing drop rule

Figure 4.9 shows the intended flow issued to test and verify any drop rule residing in the flow table. Drop rules are inspected after a matching probe packet is generated. Before the probe is sent out the controller installs two rules onto the targeted switch. See the two inserted rules with priority 30 and 10 rule in Switch1s flow table in figure 4.9. Both rules are copies of the drop rule which is being tested, but with modifications to the priority and action list. The first rule increments the priority value with 1 and changes the output action to OFPP_CONTROLLER. The second rule decrements the priority value and changes the action to send the packet to a neighbour. Catch rules are installed on the neighbour as usual.

In figure 4.9 if the probe follows the green route (2) the rule installed and works fine. If the packet should follow the red route (3) something went wrong upon installing the first catch rule. The orange route in itself is classified as an untestable scenario. The output of the tests can be found in appendix B.7.

4.7 Forwarding loop

SDN with its programmable logic and potentially exposed APIs is prone to configuration errors which may result in logical faults. One interesting scenario is a "hard" forwarding loop where a rule is directing traffic back to where it came from. This might be the result of a human error or indirectly by an application installing rules via the northbound API. Packets are generally prevented from looping eternally by decrementing the TTL[3]. In traditional networks, these situations are usually handled by dynamic routing protocols. OpenFlow enabled switches sends a packet_in to the controller if the TTL holds an invalid value, but to catch loops earlier than waiting for a packet_in with a reason of OFPR_INVALID_TTL we want to try and actively look for loops ourselves.

[3]TTL: Time-to-live, mechanism preventing packets to circulate indefinitely.

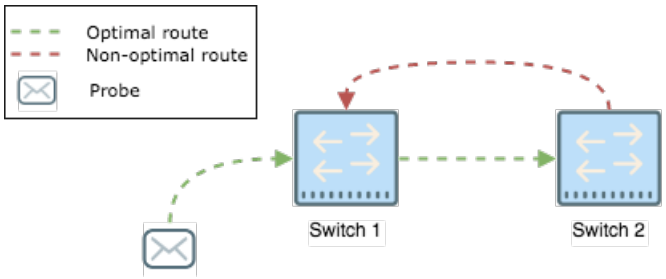

Figure 4.10: Forwarding loop

In the scenario drawn in figure 4.10 we are testing rules at Switch1 as usual, but want to check if any rule on Switch2 could cause a loop in the production traffic hitting the initial rule on Switch1. Probing as seen earlier will not catch scenarios like these due to the high priority catch-rule sitting at the receiving switch picking up the DSCP marked probes. In this scenario on Switch2. Looking for a loop-causing rule is a cumbersome task. If enabled, the approach presented does this check upon the receipt of every probe. By looping through the flow table on the neighbour who receives the probe, Switch2 in this case, and compares the "output:port" in the action list for each entry and look for a rule sending the packet back to the switch where it originated, Switch1. The port on which the packet entered the switch is not necessarily passed along with the PACKET_IN message. This means that when iterating the flow table and looking at the output port for each entry we must find the neighbour on the other side of the outgoing port and compare to the original switch in which the probe was initially injected. The output of the tests can be found in appendix B.8.

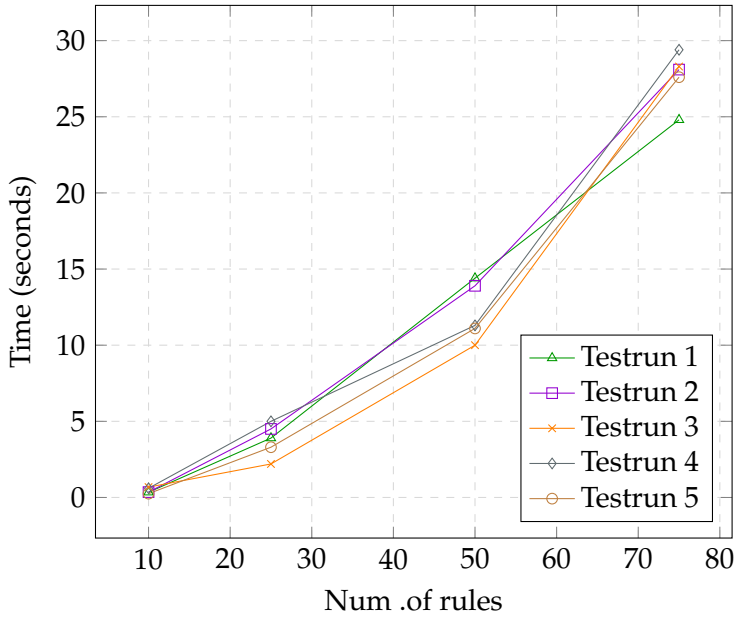

Figure 4.11: Time to check for loops

The presented technique for checking a flow table for potential loop-causing entries, per probe, is a costly manner. Figure 4.11 shows the cost when probing rather small flow tables holding few rules. Early on with 10 and 20 rules, the growth seems to be linear, but after 50 entries the results seem to take an exponential form.

4.8 Dedicated point of entry

Other research [8, 20] uses a dedicated point of entry to send probes via, as depicted in figure 4.12. We would like to assess the cost of making this extra hop and look at the time probes spend on the wire. The overlay network in VirtualBox is shared between the controller and the network switches. Figure 4.13 shows the time probes spend on-link. Time spent on-link means the time it takes for the controller to send out N number of probes over the control channel and retrieving them. The number of tests conducted is 30 for each x-value, or step, followed by the calculation of the 95% confidence interval. The graph shows the average and the whiskers show the confidence interval.

The results found for the total link time using a dedicated point of entry in 4.4 and the previous results from using TABLE 4.3 is plotted in figure 4.13. The overhead of sending probes via an entry point varies from 3.5% to almost 8%.

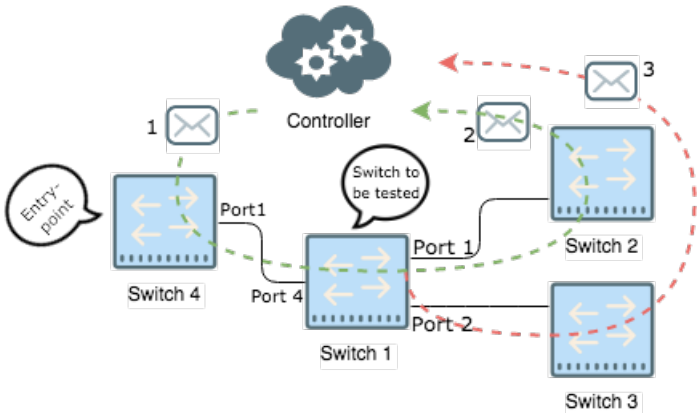

Figure 4.12: Dedicated point of entry

Num. of packets	Average (sec)	CI (95%)	Min	Max
500	0.14626630	± 0.0049426	0.13266	0.17733
1000	0.3389230	± 0.0073793	0.30811	0.39989
1500	0.5134950	± 0.0081312	0.46581	0.55813
2000	0.7198697	± 0.0093642	0.66839	0.77263
2500	0.9061517	± 0.0096656	0.84379	0.94978
3000	1.0680770	± 0.0174890	0.98558	1.19763

Table 4.4: Time spent on link (in seconds)

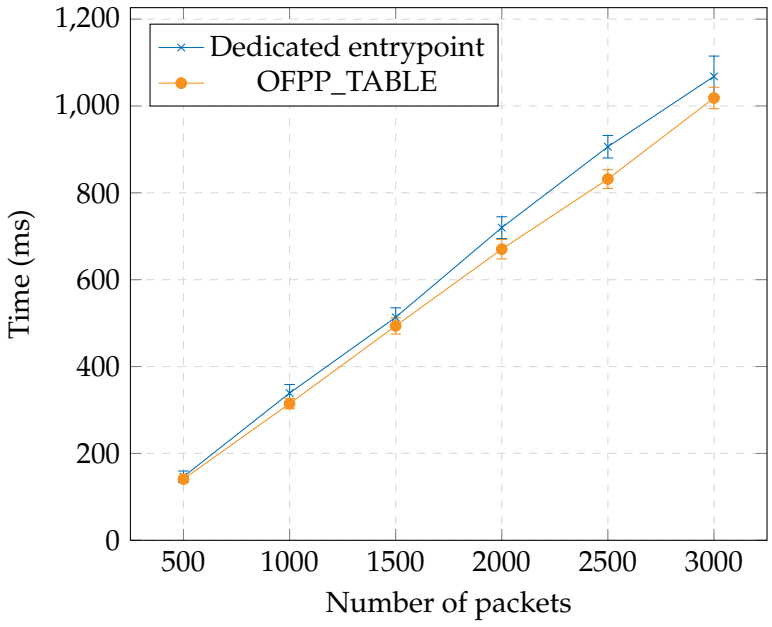

Figure 4.13: Time spent on link.

4.9 Scaling

A real-world topology may consist of ten, or maybe hundreds of network devices. Being able to scale out and probe an entire topology with interconnected devices should be feasible. In previous experiments, we are looking into different scenarios, but in a separate manner. By tying these together we can sequentially probe multiple switches. Figure 4.14 shows a topology modelled after the popular Stanford backbone network [4] which will work as the testbed for the scale test.

By sending probes directly to the targeted switch enables us to iterate through a topology without having to concern about which neighbour is used as the point of injection. The overall flow is depicted in the pseudo code in listing 4.4 and resembles that of figure 3.1. After retrieving the list and datapath objects for every switch residing in the topology, we can start iterating each and one of them. At each switch, we remove any catch-rule left over to ensure probes aren't getting sent directly back to the controller. Next, we scrape off the flow table, sort it descending by priority and starts iterating from the top. When generating a packet we only care for rules with higher or equal priority to the entry in which we are creating a probe for. After the probe is generated, the outcome is derived and stored along with the entry and the packet. The outcome, in this case, is the switch we expect the packet to be caught at. By looking at the action output we know where the packet is expected to end up i.e. at which neighbour. On the controller, the event handler actively listens for packet_in events. Upon arrival, we match the incoming packet to entries found in figure 4.4 and pop the match. The 5-tuple matching is performed both when generating probes and upon arrival of a probe. After probing every switch we iterate the topology making sure that no catch-rules are left behind.

```
for each switch in the topology:
    remove catch rule from self
    for each neighbour:
        install catch rule
    for each entry in the flowtable:
        generate a matching probe
        derive expected outcome
        send the packet out

// listen for event
if testpacket received:
    parse incoming packet
    if expected outcome:
        process next entry
    else:
        faulty rule detected

// cleanup
for each switch:
```

```
20      remove catch rule
```

Listing 4.4: Scaling out in pseudo code

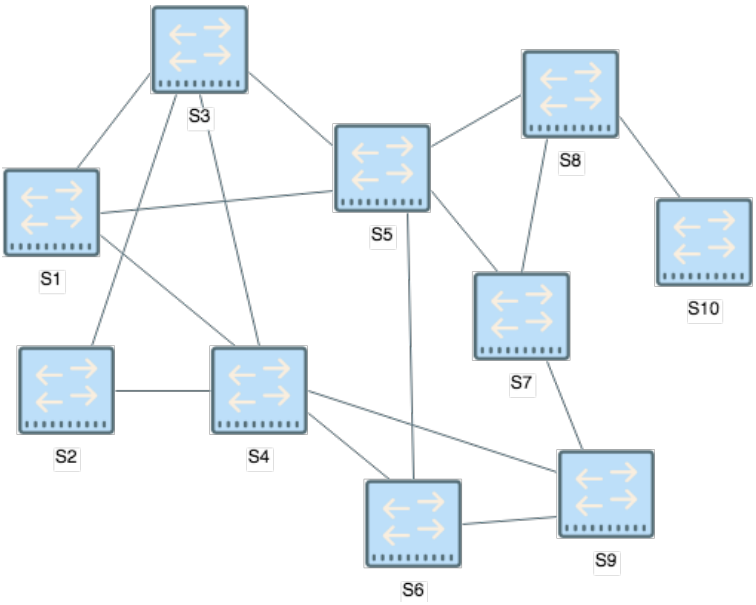

Figure 4.14: Scaled topology

Chapter 5

Discussion

5.1 Matching

When generating probes we make the assumption that a faulty rule, in the sense that it was not installed, will fail every matching packet. I.e. if the rule in question holds a range of IP addresses, packets matching each address in the range is assumed to fail. By this assumption, we only need to issue one probe per rule regardless of the prefix. After the IPSet operation, the subnet portions left are either a range holding two or more addresses, a /32 subnet holding one address or 0 indicating a total overlap. The addresses (source, destination) used for the probe packet is randomly picked from the IP portion left over after the set operation.

The box plots summarize the underlying numerical values for the probe generation and how they are spread. The x-axis represents the number of rules and y-axis represents the time. The numbers making up the boxes are made out of 30 executions per x-value. The spread of values found between the whiskers, relative to each other, seems rather stable. From the boxes, we do see a couple of outliers when generating 1000, 1500 and 2000 probes. This is possibly due to:

1. the greater amount of rules created, the more rules are likely to interfere with each other requiring additional calculations to generate the probe.
2. we draw RFC 1918 addresses with a spread as shown in figure 4.1. The likelihood of (soft) overlaps occurring is present.

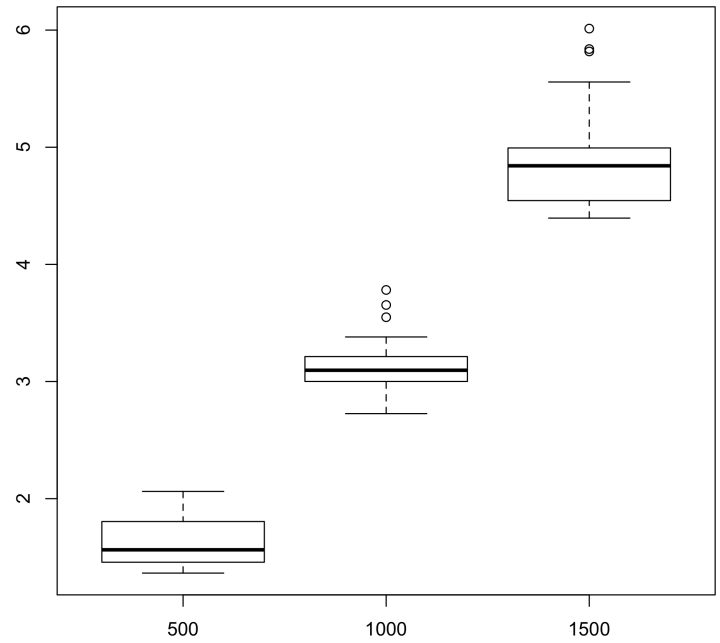

Figure 5.1: Rule generation boxplot (500 to 1500 rules)

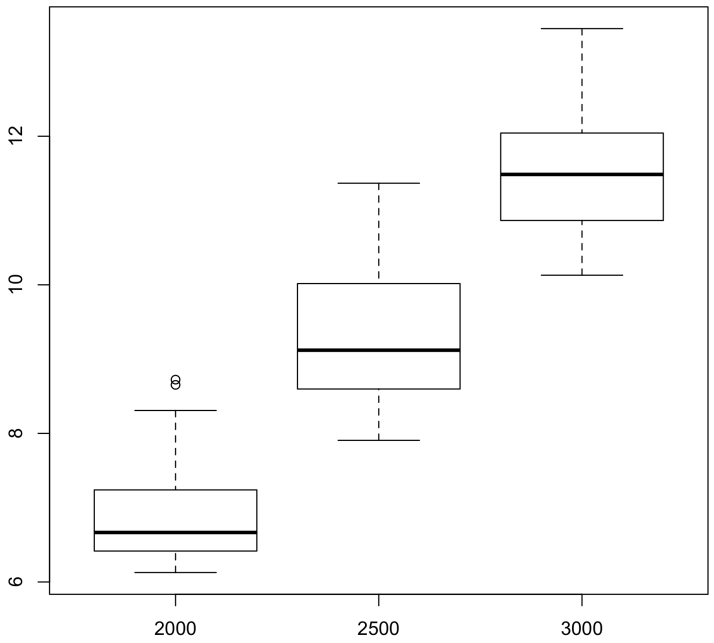

Figure 5.2: Rule generation boxplot (2000 to 3000 rules)

5.2 Project

Other research such as Pathlet Tracer [38] uses the upper 6-bits of the Type-of-Service (ToS) field in the IPv4 header to store and carry datapath IDs, Monocle [20] presents techniques for testing various rules and SDN Traceroute [18] uses OFPP_TABLE as a way of performing traceroute in SDN enabled networks. We first discovered the latter making use of the TABLE action after a couple months into the thesis. SDN Traceroute intercepts packets along their path, sending them up to the controller and uses OFPP_TABLE for injecting the altered packets back down into the data plane. This project combines these works in order to effectively inject probes and verify rules installed in the SDN data plane. By designating a single none-class value from the DSCP field, instead of using multiple header bits, we mark packets as probes. This value can be matched upon for identification purposes.

5.3 Experiments

As an addition to the experiments, one aspect of interest might be to look at the total bandwidth used by the probes. The number of probes seen within a timeframe increase with the flow table size. The time spent on-link for 3000 probes is ≈ 1 seconds. Even though the probes hold a size of ≈ 160 bytes and only comprise of header values, measuring this sudden and concentrated burst might be of interest.

For any experiment relying on a second rule on the same switch with a different output i.e. send the probe to a neighbouring switch instead of up to the controller, a different approach could be to make use of the cookie value found in the flow table entry. This value is sent with the packet_in message to the controller. By using distinct cookie values per rule one can differentiate the incoming probes and thus determine which rule the probe hit.

The rulesets used are based on private IP addresses. Expanding out from the RFC 1918 address space will increase the overall range and soft overlaps are less likely to occur. The result of this is likely to decrease the time to generate matching probe because the rules are less likely to overlap.

Chapter 6

Conclusion

6.1 Conclusion

Network issues in the data plane often manifest themselves as failed rules. This project aims to assist in the troubleshooting domain by testing rules installed in the data plane. Plugging into the controller and utilizing the control channel lets us interact with devices residing in the data plane.

Different supporting topologies of virtual networks have been used to conduct the different experiments, from small proof-of-concept topologies to scaling out on an interconnected network mimicking the Stanford backbone. When exercising flow table rules we argue for the importance of working with as realistic rulesets as possible. This is achieved by modelling how Classbench-ng probabilistically distributes IP prefixes and OXM match fields.

Looking at the research questions raised in the problem statement, we can make the final conclusion:

1) How can we ensure network policies installed works as intended?

We have shown that by utilizing probing techniques to send special-crafted probing packets testing each rule residing in a flow table, we can ensure that the network policies installed are working as intended. Learning the flow table entry beforehand, we know where the packet is expected to end up. Comparing the actual output with the expected output we determine if the rule is installed and is working as intended.

2) How can we insert probes into the data plane in an effective manner?

By comparing other research we argue for inserting probes directly onto the targeted switch currently undergoing testing. Using a dedicated point

of injection we found an added overhead in time spent on-link varying between 3.5% to $\approx$ 8%.

3) How can we probe policies using a least-amount of catch-rules?

Designating a specific value from the DSCP field in the IPv4 header we successfully mark packets as probes used for testing purposes. Probing is performed on a per-hop basis, and we end up with a total number of catch-rules equaling the number of neighbours for each switch currently in the topology.

4) How can we verify different types of rules?

By deriving different test-scenarios for the different types of rules, as seen in chapter 4, we successfully probe unicast, drop and loop-causing rules. Further, we show that by using a linear search algorithm we can create distinct rule-matching probes for 500 rules in 1.6 seconds and 3000 rules in 11.5 seconds. Spending more than 10-15 seconds looking through flow tables holding 50 rules and 25-30 seconds for 80 rules, our solution for looking for loop-causing rules prove to be rather slow when iterating flow tables.

6.2 Future work

Optimizing number of probes
The topologies used in the experiments can be thought of as a snippet of a larger network. In larger networks or topologies, a packet might perform several hops before it reaches its destination. When iterating the network devices the probability of generating a new probe packet for the same rule on several hops is presenet, accumulating overhead in the form of time spent making probes. Mapping out the span of different rules i.e. on how many hops do we see the same rule, might be leveraged to generate probes in batches beforehand instead of the streamlined per-hop fashion used in this thesis.

Concurrent probing
Pulling and analysing datasets from production environments might hold patterns in rule-layout and packet flow. Building dependency graphs around a potential pattern might assist in mapping out any paths these rules might take throughout the topology. This could be leveraged to concurrently probe the network. An offline way of calculating paths might be a costly manner as the entire topology must be considered for each rule.

Traceroute
Other research [38] impound multiple bits from the ToS octet or [18] all bits residing in the PCP field in the Ethernet header to perform traceroute in SDN enabled networks. These bits might be used by layer 2 class of service (CoS) and layer 3 quality of service (QoS) and may interfere with

production traffic. Keeping the number of bits used for marking to a minimum i.e. one, combined with TTL processing may enable a close to non-intrusive way of tracing packet trajectories from the controller.

Optimize probe marking
In this thesis, we set aside a specific DSCP value marking packets as probes. The DSCP value used is found in the best effort default class 0 (routine). Leveraging the multiple table architecture in OpenFlow, similar rules can be grouped together into different tables. Different probe marking techniques might be derived per table, or group. Looking for unused header bits per group could result in a dynamic approach of marking packets as probes differently per group.

Bibliography

[1] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[2] Theophilus Benson, Aditya Akella, and David Maltz. "Unraveling the Complexity of Network Management". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 335–348. URL: http://dl.acm.org/citation.cfm?id=1558977.1559000 (visited on 01/19/2018).

[3] Networkworld. *The dark side of server virtualization*. 2010. URL: https://www.networkworld.com/article/2213301/virtualization/the-dark-side-of-server-virtualization.html.

[4] H. Zeng et al. "Automatic Test Packet Generation". In: *IEEE/ACM Transactions on Networking* 22.2 (Apr. 2014), pp. 554–566. ISSN: 1063-6692. DOI: 10.1109/TNET.2013.2253121.

[5] P. C. d R. Fonseca and E. S. Mota. "A Survey on Fault Management in Software-Defined Networks". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2284–2321. DOI: 10.1109/COMST.2017.2719862.

[6] IHS Markit. *Businesses Losing $700 Billion a Year to IT Downtime*. 2016. URL: http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs.

[7] Q. Zhi and W. Xu. "MED: The Monitor-Emulator-Debugger for Software-Defined Networks". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications. Apr. 2016, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524562.

[8] Maciej Kuzniar, Peter Peresini, and Dejan Kostić. "Providing Reliable FIB Update Acknowledgments in SDN". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: ACM, 2014, pp. 415–422. ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2675006. URL: http://doi.acm.org/10.1145/2674005.2675006.

[9] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. "What You Need to Know About SDN Flow Tables". In: *PAM*. 2015.

[10] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. "ProboScope: Data Plane Probe Packet Generation". In: (2014).

[11] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: http://doi.acm.org/10.1145/1355734.1355746.

[12] Open Networking Foundation. *ONF*. 2018. URL: https://www.opennetworking.org/.

[13] Myron Lane. *CPS 590: Software Defined Networking*. 2016. URL: http://slideplayer.com/slide/5722212/.

[14] Open Networking Foundation. *ONF*. 2018. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[15] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: http://doi.acm.org/10.1145/1868447.1868466.

[16] Beatrix Clarke. *Introduction to Mininet, Open vSwitch, and POX*. 2016. URL: http://slideplayer.com/slide/9534240/.

[17] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. "CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: ACM, 2015, 23:1–23:7. ISBN: 978-1-4503-3451-8. DOI: 10.1145/2774993.2775066. URL: http://doi.acm.org/10.1145/2774993.2775066.

[18] Kanak Agarwal et al. "SDN Traceroute: Tracing SDN Forwarding Without Changing Network Behavior". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 145–150. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620756. URL: http://doi.acm.org/10.1145/2620728.2620756.

[19] Nikhil Handigol et al. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 71–85. ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol.

[20] Peter Peresini, Maciej Kuzniar, and Dejan Kostic. "Monocle: Dynamic, Fine-grained Data Plane Monitoring". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '15. Heidelberg, Germany: ACM, 2015, 32:1–32:13. ISBN: 978-1-4503-3412-9. DOI: 10.1145/2716281.2836117. URL: http://doi.acm.org/10.1145/2716281.2836117.

[21] Maciej Kuzniar, Peter Peresíni, and Dejan Kostic. "What You Need to Know About SDN Flow Tables". In: *PAM*. 2015.

[22] Hongqiang Harry Liu et al. "Traffic Engineering with Forward Fault Correction". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 527–538. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626314. URL: http://doi.acm.org/10.1145/2619239.2626314.

[23] X. Wen et al. "RuleScope: Inspecting Forwarding Faults for Software-Defined Networking". In: *IEEE/ACM Transactions on Networking* 25.4 (Aug. 2017), pp. 2347–2360. ISSN: 1063-6692. DOI: 10.1109/TNET.2017.2686443.

[24] Y. Zhao et al. "SDN-enabled Rule Verification on Data Plane". In: *IEEE Communications Letters* (2017), pp. 1–1. ISSN: 1089-7798. DOI: 10.1109/LCOMM.2017.2701369.

[25] Ehab Al-Shaer and Saeed Al-Haj. "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures". In: *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*. SafeConfig '10. Chicago, Illinois, USA: ACM, 2010, pp. 37–44. ISBN: 978-1-4503-0093-3. DOI: 10.1145/1866898.1866905. URL: http://doi.acm.org/10.1145/1866898.1866905.

[26] Haohui Mai et al. "Debugging the Data Plane with Anteater". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 290–301. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018470. URL: http://doi.acm.org/10.1145/2018436.2018470.

[27] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 113–126. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[28] Ahmed Khurshid et al. "Veriflow: Verifying Network-wide Invariants in Real Time". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Sept. 2012), pp. 467–472. ISSN: 0146-4833. DOI: 10.1145/2377677.2377766. URL: http://doi.acm.org/10.1145/2377677.2377766.

[29] Brandon Heller et al. “Leveraging SDN Layering to Systematically Troubleshoot Networks”. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 37–42. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491197. URL: http://doi.acm.org/10.1145/2491185.2491197.

[30] IANA. *Differentiated Services Field Codepoints (DSCP)*. URL: http://www.iana.org/assignments/dscp-registry/dscp-registry.xhtml.

[31] R. Aryan et al. “A General Formalism for Defining and Detecting OpenFlow Rule Anomalies”. In: *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*. Oct. 2017, pp. 426–434. DOI: 10.1109/LCN.2017.94.

[32] Ehab S. Al-shaer and Hazem H. Hamed. *Design and Implementation of Firewall Policy Advisor Tools*. Tech. rep. 2002.

[33] Y. Xu et al. “High-Throughput and Memory-Efficient Multimatch Packet Classification Based on Distributed and Pipelined Hash Tables”. In: *IEEE/ACM Transactions on Networking* 22.3 (June 2014), pp. 982–995. ISSN: 1063-6692. DOI: 10.1109/TNET.2013.2270441.

[34] Y. C. Cheng and P. C. Wang. “Scalable Multi-Match Packet Classification Using TCAM and SRAM”. In: *IEEE Transactions on Computers* 65.7 (July 2016), pp. 2257–2269. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2470242.

[35] Ryu SDN Framework Community. *Ryu OpenFlow controller*. 2018. URL: https://osrg.github.io/ryu/.

[36] D. E. Taylor and J. S. Turner. “ClassBench: a packet classification benchmark”. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 3. 2005, 2068–2079 vol. 3. DOI: 10.1109/INFCOM.2005.1498483.

[37] J. Matoušek et al. “ClassBench-ng: Recasting ClassBench after a Decade of Network Evolution”. In: *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. May 2017, pp. 204–216. DOI: 10.1109/ANCS.2017.33.

[38] Hui Zhang et al. “Enabling Layer 2 Pathlet Tracing Through Context Encoding in Software-defined Networking”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 169–174. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620742. URL: http://doi.acm.org/10.1145/2620728.2620742.

Appendices

Appendix A

OpenFlow

A.1 OFPT_FLOW_MOD

```
/* Flow setup and teardown (controller -> datapath). */
struct ofp_flow_mod {
    struct ofp_header header;
    uint64_t cookie;      /* Opaque controller-issued identifier. */
    uint64_t cookie_mask; /* Mask used to restrict the cookie bits
                           that must match when the command is
                           OFPFC_MODIFY* or OFPFC_DELETE*. A value
                           of 0 indicates no restriction. */
    /* Flow actions. */
    uint8_t table_id;     /* ID of the flow table */
    uint8_t command;      /* One of OFPFC_*. */
    uint16_t idle_timeout; /* Idle time before discarding (sec). */
    uint16_t hard_timeout; /* Max time before discarding (sec). */
    uint16_t priority;    /* Priority level of flow entry. */
    uint32_t buffer_id;   /* Buffered packet to apply to, or
                           OFP_NO_BUFFER.
                           Not meaningful for OFPFC_DELETE*. */
    uint32_t out_port;    /* For OFPFC_DELETE* commands, require
                           matching entries to include this as an
                           output port. A value of OFPP_ANY
                           indicates no restriction. */
    uint32_t out_group;   /* For OFPFC_DELETE* commands, require
                           matching entries to include this as an
                           output group. A value of OFPG_ANY
                           indicates no restriction. */
    uint16_t flags;       /* One of OFPFF_*. */
    uint8_t pad[2];
    // Fields to match. Variable size.
    struct ofp_match match;
    // Instruction set
    //struct ofp_instruction instructions[0];
};
OFP_ASSERT(sizeof(struct ofp_flow_mod) == 56);
```

Listing A.1: OFPT_FLOW_MOD

```
enum ofp_flow_mod_command {
    // New flow.
    OFPFC_ADD         = 0
    // Modify all matching flows.
    OFPFC_MODIFY      = 1
    /* Modify entry strictly matching wildcards
       and priority. */
    OFPFC_MODIFY_STRICT = 2
    // Delete all matching flows.
    OFPFC_DELETE      = 3
    /* Delete entry strictly matching wildcards
       and priority. */
    OFPFC_DELETE_STRICT = 4
```

Listing A.2: OFPT_FLOW_MOD_COMMAND

```
enum ofp_flow_mod_flags {
    // Send flow removed message when flow expires or is deleted.
    OFPFF_SEND_FLOW_REM = 1 << 0
    // Check for overlapping entries first.
    OFPFF_CHECK_OVERLAP = 1 << 1
    // Reset flow packet and byte counts.
    OFPFF_RESET_COUNTS = 1 << 2
    // Don't keep track of packet count.
    OFPFF_NO_PKT_COUNTS = 1 << 3
    // Don't keep track of byte count.
    OFPFF_NO_BYT_COUNTS = 1 << 4
};
```

Listing A.3: OFPT_FLOW_MOD_FLAGS

A.2 OXM fields

Field	Bits	Mask	Pre-requisite
OXM_OF_IN_PORT	32	No	None
OXM_OF_IN_PHY_PORT	32	No	IN_PORT present
OXM_OF_METADATA	64	Yes	None
OXM_OF_ETH_DST	48	Yes	None
OXM_OF_ETH_SRC	48	Yes	None
OXM_OF_ETH_TYPE	16	No	None
OXM_OF_VLAN_VID	12+1	Yes	None
OXM_OF_VLAN_PCP	3	No	VLAN_VID != NONE
OXM_OF_IP_DSCP	6	No	ETH_TYPE = 0x0800 or 0x86dd
OXM_OF_IP_ECN	2	No	ETH_TYPE = 0x0800 or 0x86dd
OXM_OF_IP_PROTO	8	No	ETH_TYPE = 0x0800 or 0x86dd
OXM_OF_IPV4_SRC	32	Yes	ETH_TYPE = 0x0800
OXM_OF_IPV4_DST	32	Yes	ETH_TYPE = 0x0800
OXM_OF_TCP_SRC	16	No	IP_PROTO = 6
OXM_OF_TCP_DST	16	No	IP_PROTO = 6
OXM_OF_UDP_SRC	16	No	IP_PROTO = 17
OXM_OF_UDP_DST	16	No	IP_PROTO = 17
OXM_OF_SCTP_SRC	16	No	IP_PROTO = 132
OXM_OF_SCTP_DST	16	No	IP_PROTO = 132
OXM_OF_ICMPV4_TYPE	8	No	IP_PROTO = 1
OXM_OF_ICMPV4_CODE	8	No	IP_PROTO = 1
OXM_OF_ARP_OP	16	No	ETH_TYPE = 0x0806
OXM_OF_ARP_SPA	32	Yes	ETH_TYPE = 0x0806
OXM_OF_ARP_TPA	32	Yes	ETH_TYPE = 0x0806
OXM_OF_ARP_SHA	48	Yes	ETH_TYPE = 0x0806
OXM_OF_ARP_THA	48	Yes	ETH_TYPE = 0x0806
OXM_OF_IPV6_SRC	128	Yes	ETH_TYPE = 0x86dd
OXM_OF_IPV6_DST	128	Yes	ETH_TYPE = 0x86dd
OXM_OF_IPV6_FLABEL	20	Yes	ETH_TYPE = 0x86dd
OXM_OF_ICMPV6_TYPE	8	No	IP_PROTO = 58
OXM_OF_ICMPV6_CODE	8	No	IP_PROTO = 58
OXM_OF_IPV6_ND_TARGET	128	No	ICMPV6_TYPE = 135 or 136
OXM_OF_IPV6_ND_SLL	48	No	ICMPV6_TYPE = 135
OXM_OF_IPV6_ND_TLL	48	No	ICMPV6_TYPE = 136
OXM_OF_MPLS_LABEL	20	No	ETH_TYPE = 0x8847 or 0x8848
OXM_OF_MPLS_TC	3	No	ETH_TYPE = 0x8847 or 0x8848
OXM_OF_MPLS_BOS	1	No	ETH_TYPE = 0x8847 or 0x8848
OXM_OF_PBB_ISID	24	Yes	ETH_TYPE = 0x88E7
OXM_OF_TUNNEL_ID	64	Yes	None
OXM_OF_IPV6_EXTHDR	9	Yes	ETH_TYPE = 0x86dd

Table A.1: OXM Flow match field types.

Appendix B

Tests/output

B.1 Benchmarks

Packets	Run #1	Run #2	Run #3	Run #4	Run #5	Run #6	Run #7	Run #8	Run #9	Run #10
100	0.29905	0.33214	0.33368	0.35869	0.32310	0.37185	0.35334	0.34025	0.35900	0.42103
500	1.80485	1.56350	1.59119	2.06207	1.57818	1.50688	1.45002	1.67102	1.40870	1.74730
1000	3.21327	2.97594	3.07494	3.00060	3.37434	3.07309	3.03187	3.02237	3.16325	3.15635
1500	4.45192	4.87486	4.52947	4.94220	5.81770	4.68137	4.99397	4.44733	4.57385	4.73837
2000	7.87965	7.23923	6.62649	6.58623	6.36527	8.65313	6.55051	6.22572	6.53359	7.14350
2500	9.53647	8.64303	9.44523	11.36782	10.41022	7.90635	8.86894	10.93603	9.10549	10.01683
3000	10.74782	10.61442	11.56476	11.57418	10.70689	10.60625	12.04318	12.19779	10.28235	10.12929
Packets	Run #11	Run #12	Run #13	Run #14	Run #15	Run #16	Run #17	Run #18	Run #19	Run #20
100	0.34384	0.40047	0.32684	0.37171	0.31345	0.36756	0.31597	0.38000	0.42764	0.39751
500	1.48383	1.36472	1.55305	1.41550	1.84116	2.03636	1.81678	1.38412	1.64167	1.45753
1000	3.27957	3.07938	3.15854	3.78173	3.01240	3.09880	2.84558	2.88553	3.38050	3.09369
1500	5.38174	4.87077	4.66322	4.74591	5.12835	4.40803	5.55701	4.97531	4.57936	4.54454
2000	6.33658	6.25433	6.39868	8.22729	6.59248	7.28714	6.67918	7.14474	8.30854	7.12858
2500	8.52328	8.46808	9.16442	8.97908	8.57020	9.48606	8.79424	8.56396	8.59756	9.27796
3000	10.91433	11.28330	11.66031	12.56403	11.32633	12.49071	11.20763	11.40698	12.84985	11.62382
Packets	Run #21	Run #22	Run #23	Run #24	Run #25	Run #26	Run #27	Run #28	Run #29	Run #30
100	0.41478	0.37619	0.30274	0.32750	0.55192	0.40443	0.34483	0.35783	0.36246	0.36758
500	1.87897	1.43942	1.42809	1.54838	1.96006	1.84745	1.65232	1.56357	1.56310	1.53815
1000	2.99957	3.14241	3.09930	2.90129	3.15838	2.93195	3.65344	3.29369	2.72592	3.54888
1500	4.39502	5.03556	6.01276	4.40880	4.97835	4.90644	4.84880	5.83818	4.53984	4.83465
2000	7.57381	7.02912	6.79461	6.42738	6.41632	8.72343	6.34051	6.12678	6.65349	7.15591
2500	9.78623	8.98721	8.90072	7.94897	10.20193	8.10752	10.86253	10.76301	9.13439	10.10536
3000	11.67730	12.39787	11.31151	11.71668	10.86588	11.76772	11.37442	13.44892	10.58301	12.62309

Table B.1: Packet generation

Packets	Run #1	Run #2	Run #3	Run #4	Run #5	Run #6	Run #7	Run #8	Run #9	Run #10
100	0.01452	0.01371	0.01360	0.01401	0.01402	0.01515	0.01327	0.01454	0.01350	0.01288
500	0.15249	0.14596	0.14717	0.14709	0.14088	0.14965	0.13672	0.14085	0.14040	0.13403
1000	0.31622	0.30068	0.30439	0.32307	0.30822	0.33238	0.32476	0.32174	0.31736	0.33522
1500	0.52872	0.48107	0.47324	0.48865	0.48231	0.49295	0.50605	0.47873	0.46892	0.52546
2000	0.66840	0.66221	0.68927	0.69263	0.69521	0.67682	0.63484	0.68824	0.70217	0.67200
2500	0.84292	0.83256	0.85569	0.84463	0.82011	0.82978	0.80361	0.81112	0.81415	0.81960
3000	1.00456	1.00994	1.00687	1.03988	1.01875	1.02201	1.04063	1.03016	0.98968	0.99894
Packets	Run #11	Run #12	Run #13	Run #14	Run #15	Run #16	Run #17	Run #18	Run #19	Run #20
100	0.01378	0.01269	0.01386	0.01438	0.01409	0.01389	0.01455	0.01523	0.01530	0.01225
500	0.13879	0.13385	0.13612	0.14252	0.13974	0.14403	0.14192	0.13362	0.14352	0.13347
1000	0.31241	0.30802	0.33654	0.30080	0.29958	0.31856	0.30805	0.33122	0.30858	0.31677
1500	0.51138	0.47612	0.49538	0.48128	0.53054	0.47429	0.49304	0.52353	0.47543	0.49174
2000	0.65525	0.67712	0.67196	0.67584	0.64561	0.66598	0.72687	0.64548	0.66906	0.66446
2500	0.89194	0.80844	0.86513	0.81603	0.84064	0.82881	0.85299	0.83721	0.88066	0.81642
3000	1.05299	0.98010	1.02035	0.97898	1.00736	1.00585	1.06912	1.03365	1.04820	1.00453
Packets	Run #21	Run #22	Run #23	Run #24	Run #25	Run #26	Run #27	Run #28	Run #29	Run #30
100	0.01386	0.01361	0.01350	0.01342	0.01572	0.01378	0.01360	0.01624	0.01263	0.01384
500	0.13979	0.13129	0.13246	0.13384	0.14681	0.14125	0.14792	0.13890	0.14759	0.14445
1000	0.31842	0.30102	0.30248	0.31151	0.32212	0.30447	0.34143	0.30258	0.30702	0.30435
1500	0.48932	0.47379	0.49560	0.48076	0.50878	0.47826	0.48814	0.50598	0.52871	0.48027
2000	0.66415	0.67231	0.66882	0.65354	0.63489	0.65701	0.71923	0.65129	0.67216	0.63536
2500	0.83167	0.82516	0.82471	0.85162	0.82478	0.83012	0.80422	0.80972	0.81732	0.81962
3000	1.04379	0.99110	1.02141	0.96698	1.00521	1.03581	1.05931	1.02956	1.02840	1.01002

Table B.2: Time spent on link

Packets	Run #1	Run #2	Run #3	Run #4	Run #5	Run #6	Run #7	Run #8	Run #9	Run #10
100	0.01373	0.01757	0.01942	0.01353	0.01332	0.01362	0.01605	0.01348	0.01474	0.01314
500	0.14169	0.14215	0.13609	0.14002	0.13769	0.13266	0.13682	0.14225	0.13407	0.14474
1000	0.35680	0.32182	0.30975	0.35972	0.31613	0.33065	0.30811	0.31250	0.32244	0.32419
1500	0.53535	0.46581	0.47394	0.50001	0.51872	0.47465	0.48421	0.51300	0.51191	0.51995
2000	0.71793	0.67378	0.71874	0.71383	0.72593	0.69762	0.72772	0.71461	0.69133	0.73548
2500	0.92782	0.85757	0.93537	0.87331	0.91073	0.89357	0.86476	0.87783	0.89452	0.91973
3000	1.07684	1.04339	1.11497	1.04964	1.04323	1.04925	1.09210	1.06192	1.01893	1.05997
Packets	Run #11	Run #12	Run #13	Run #14	Run #15	Run #16	Run #17	Run #18	Run #19	Run #20
100	0.01558	0.01469	0.01409	0.01344	0.01401	0.01353	0.01572	0.01610	0.01461	0.01397
500	0.13604	0.14315	0.13683	0.17255	0.16074	0.14314	0.13707	0.13592	0.15943	0.17207
1000	0.35267	0.34482	0.32116	0.35485	0.32189	0.33675	0.33415	0.33553	0.33646	0.33822
1500	0.49362	0.50372	0.50983	0.50693	0.52675	0.49991	0.50788	0.53021	0.51998	0.51441
2000	0.74334	0.68448	0.67942	0.69173	0.71526	0.74972	0.72603	0.71852	0.70894	0.73605
2500	0.88761	0.90485	0.89681	0.89098	0.92375	0.89796	0.90945	0.92121	0.92395	0.84379
3000	0.98558	1.06123	1.09082	1.19763	0.99061	1.04175	1.05488	1.05335	1.04561	0.99369
Packets	Run #21	Run #22	Run #23	Run #24	Run #25	Run #26	Run #27	Run #28	Run #29	Run #30
100	0.01362	0.01273	0.01351	0.01347	0.01494	0.01349	0.01361	0.01441	0.01417	0.01442
500	0.13775	0.14570	0.14519	0.13412	0.14732	0.16309	0.17242	0.13803	0.14192	0.17733
1000	0.33255	0.33550	0.34247	0.35283	0.36176	0.39989	0.33692	0.34386	0.36472	0.35858
1500	0.49528	0.52479	0.51435	0.53072	0.53214	0.52205	0.55516	0.55813	0.53595	0.52549
2000	0.77263	0.66839	0.71497	0.74253	0.73019	0.72247	0.74430	0.75112	0.72540	0.75363
2500	0.92674	0.89378	0.93328	0.93423	0.91273	0.94978	0.91549	0.89329	0.92420	0.94546
3000	1.00650	1.05305	1.10402	1.06351	1.09999	1.10904	1.11981	1.11162	1.12883	1.12056

Table B.3: Linktime with dedicated point of entry

B.2 Comparison

```
>>> rule = ipv4.ipv4(csum=11413, dst='172.26.209.196', flags=0,
    header_length=5, identification=0, offset=0, option=None,
    proto=0, src='172.22.101.91', tos=4, total_length=20, ttl=255,
    version=4)
>>>
>>> pkt = ipv4.ipv4(csum=11413, dst='172.26.209.196', flags=0,
    header_length=5, identification=0, offset=0, option=None,
    proto=0, src='172.22.101.91', tos=4, total_length=20, ttl=255,
    version=4)
>>>
>>> rule == pkt
False
>>> rule is pkt
False
>>> type(rule)
<class 'ryu.lib.packet.ipv4.ipv4'>
>>> type(pkt)
<class 'ryu.lib.packet.ipv4.ipv4'>
```

Listing B.1: output packet gen

B.3 Hardware

	Mininet	**Ryu**
CPU Model	Intel i7-4770HQ	Intel i7-4770HQ
CPU Architecture	x86_64	x86_64
CPU Speed	2.20 GHz	2.20 GHz
CPU Cores	1	4
Thread per core	1	2
CPU L2 Cache	256K	256K
Memory size	1GB	16GB
Memory type	DDR3	DDR3
Memory speed	1600 MHz	1600 MHz

Table B.4: Hardware specifications

B.4 OFPP_TABLE

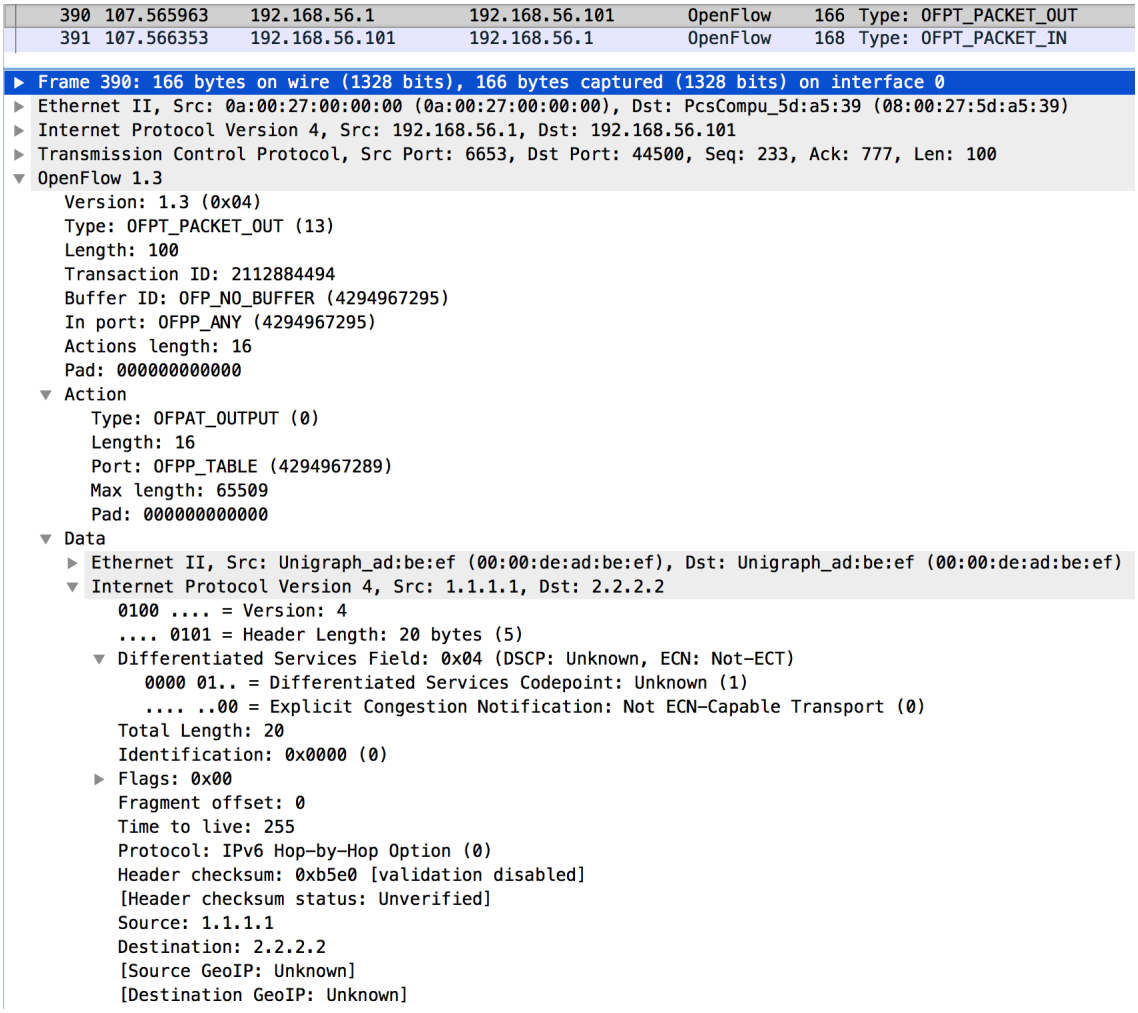

Figure B.1: OFPT_PACKET_OUT capture showing the OFPP_TABLE set.

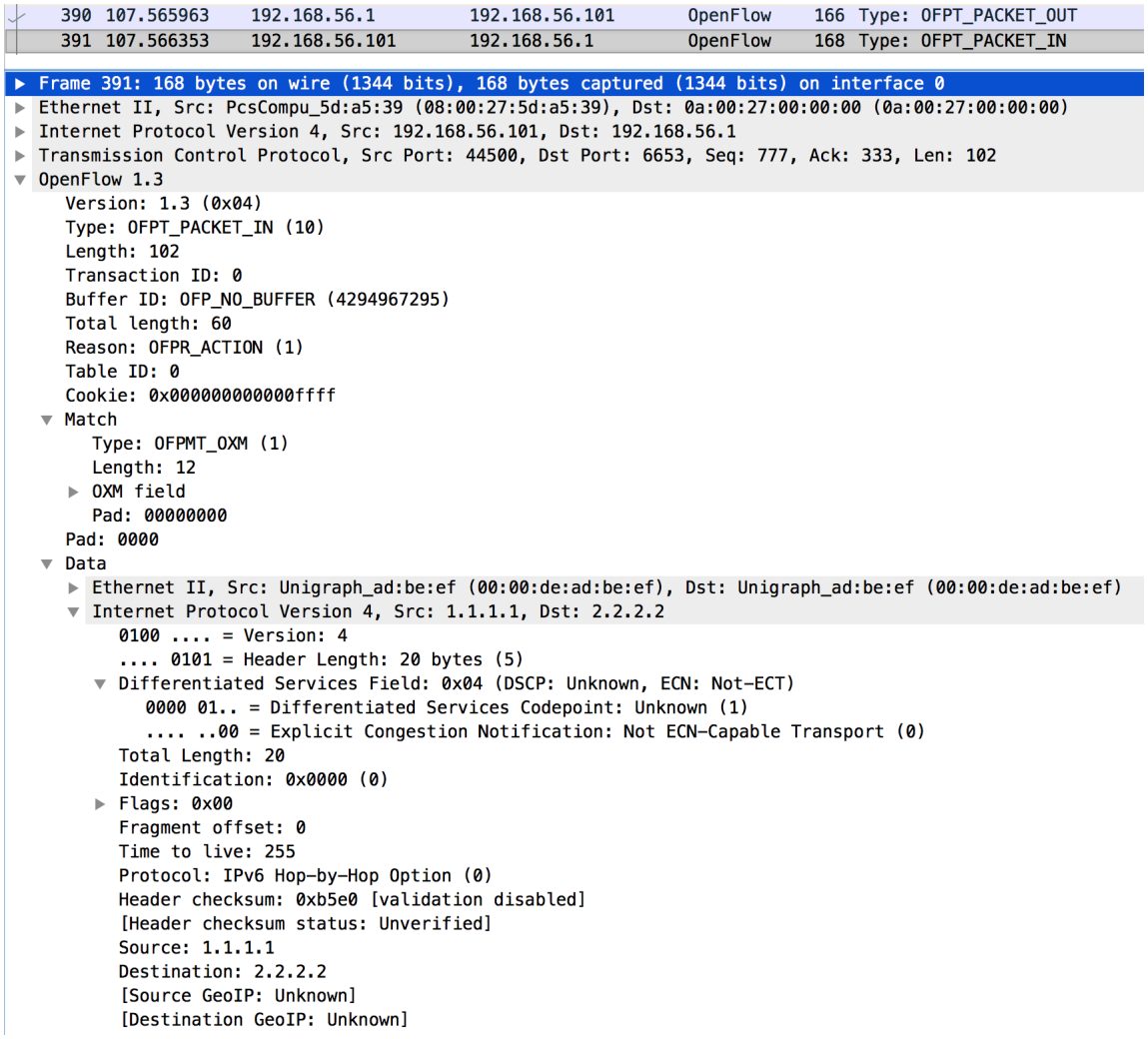

Figure B.2: OFPT_PACKET_IN capture showing the incoming packet.

B.5 Packetgen/match

```
{' priority ': 57724, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src':
    '10.126.171.0/255.255.255.0', 'nw_dst': '172.26.167.99', 'nw_proto': 6, 'tp_src': 123, 'tp_dst': 39329}}
{' priority ': 54963, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src':
    '192.168.57.0/255.255.255.0', 'nw_dst': '172.30.215.60'}}
{' priority ': 54563, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src': '10.104.47.6',
    'nw_dst': '192.168.58.87'}}
{' priority ': 45738, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src': '10.38.7.147',
    'nw_dst': '192.168.185.0/255.255.255.0'}}
{' priority ': 36848, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src': '192.168.82.30',
    'nw_dst': '192.168.54.44'}}
{' priority ': 32495, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src':
    '192.168.123.28/255.255.255.254', 'nw_dst': '172.25.221.157'}}
{' priority ': 30326, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src': '172.16.61.222',
    'nw_dst': '10.87.4.196/255.255.255.252'}}
{' priority ': 20716, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src':
    '192.168.20.0/255.255.254.0', 'nw_dst': '192.168.62.0/255.255.254.0'}}
{' priority ': 19260, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src': '172.25.143.233',
    'nw_dst': '172.28.154.62'}}
{' priority ': 19201, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src':
    '172.28.229.128/255.255.255.128', 'nw_dst': '192.168.89.199', 'nw_proto': 6, 'tp_src': 32985, 'tp_dst': 20}}
{' priority ': 16056, 'cookie': 0, 'actions': ['OUTPUT:2'], 'match': {'in_port': 3, 'dl_type': 2048, 'nw_src':
    '172.18.65.0/255.255.255.0', 'nw_dst': '10.9.39.251'}}
{' priority ': 12136, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src':
    '10.131.152.0/255.255.255.0', 'nw_dst': '172.18.130.225'}}
{' priority ': 10607, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src':
    '10.93.219.0/255.255.255.0', 'nw_dst': '172.31.171.19'}}
{' priority ': 8879, 'cookie': 0, 'actions': ['OUTPUT:3'], 'match': {'in_port': 2, 'dl_type': 2048, 'nw_src': '10.126.205.5',
    'nw_dst': '172.29.132.224/255.255.255.248', 'nw_proto': 6, 'tp_src': 51377, 'tp_dst': 21}}

PACKET: ethernet(dst='f6:6f:db:7f:0b:6a',ethertype=2048,src='64:32:22:39:b4:d7'),
        ipv4(csum=0,dst='172.26.167.99',flags=0,header_length=5,identification=0,offset=0,option=None,proto=6,
        src='10.126.171.66',tos=4,total_length=0, ttl=255,version=4),
        tcp(ack=0,bits=0,csum=0,dst_port=39329,offset=0,option=None,seq=0,src_port=123,urgent=0,window_size=0)

PACKET: ethernet(dst='d4:23:0a:38:22:fe',ethertype=2048,src='65:d7:92:fe:03:48'),
        ipv4(csum=0,dst='172.30.215.60',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='192.168.57.51',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='51:07:2f:19:8b:e5',ethertype=2048,src='d5:ed:3a:ca:7d:97'),
        ipv4(csum=0,dst='192.168.58.87',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='10.104.47.6',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='bf:81:ef:86:4a:54',ethertype=2048,src='15:b4:3d:d5:d3:a5'),
        ipv4(csum=0,dst='192.168.185.113',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='10.38.7.147',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='be:57:a6:dd:aa:87',ethertype=2048,src='dc:20:0e:55:29:b6'),
        ipv4(csum=0,dst='192.168.54.44',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='192.168.82.30',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='47:b2:a4:93:8b:9d',ethertype=2048,src='09:ca:aa:90:17:63'),
        ipv4(csum=0,dst='172.25.221.157',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='192.168.123.28',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='54:23:42:f8:7e:bd',ethertype=2048,src='e0:ce:e2:96:aa:30'),
        ipv4(csum=0,dst='10.87.4.198',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='172.16.61.222',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='24:04:0a:50:0e:58',ethertype=2048,src='9d:a7:0b:e0:2e:95'),
        ipv4(csum=0,dst='192.168.62.218',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='192.168.20.25',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='3a:e8:be:b0:8c:4e',ethertype=2048,src='49:fa:a1:68:3b:32'),
        ipv4(csum=0,dst='172.28.154.62',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='172.25.143.233',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='0e:89:4c:5d:b8:11',ethertype=2048,src='0e:fd:e8:4b:44:3e'),
        ipv4(csum=0,dst='192.168.89.199',flags=0,header_length=5,identification=0,offset=0,option=None,proto=6,
        src='172.28.229.216',tos=4,total_length=0, ttl=255,version=4),
        tcp(ack=0,bits=0,csum=0,dst_port=20,offset=0,option=None,seq=0,src_port=32985,urgent=0,window_size=0)

PACKET: ethernet(dst='63:48:fb:2f:a6:79',ethertype=2048,src='4f:14:ad:78:2e:f9'),
        ipv4(csum=0,dst='10.9.39.251',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='172.18.65.171',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='76:9a:0c:c9:44:f6',ethertype=2048,src='56:8e:4d:88:26:ce'),
        ipv4(csum=0,dst='172.18.130.225',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='10.131.152.78',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='f9:1e:4a:2b:3b:7c',ethertype=2048,src='0b:2a:1c:71:9b:fa'),
        ipv4(csum=0,dst='172.31.171.19',flags=0,header_length=5,identification=0,offset=0,option=None,proto=0,
        src='10.93.219.58',tos=4,total_length=0, ttl=255,version=4)

PACKET: ethernet(dst='bb:0b:03:17:f5:3a',ethertype=2048,src='1e:cb:a1:5f:1d:86'),
        ipv4(csum=0,dst='172.29.132.226',flags=0,header_length=5,identification=0,offset=0,option=None,proto=6,
        src='10.126.205.5',tos=4,total_length=0, ttl=255,version=4),
        tcp(ack=0,bits=0,csum=0,dst_port=21,offset=0,option=None,seq=0,src_port=51377,urgent=0,window_size=0)
```

Listing B.2: Packet generating/matching

B.6 FP-test

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows -O OpenFlow13 s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
    cookie=0x0, duration=1.369s, table=0, n_packets=0, n_bytes=0,
        priority=10594,ip,in_port=2, nw_src=172.24.8.176/28,
        nw_dst=10.164.121.218 actions=output:3
    cookie=0x0, duration=1.995s, table=0, n_packets=8, n_bytes=480,
        priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
        actions=CONTROLLER:65535
    cookie=0x0, duration=1.369s, table=0, n_packets=0, n_bytes=0,
        priority=21010,ip,in_port=3, nw_src=172.25.107.97,
        nw_dst=10.28.10.232 actions=output:2
    cookie=0x0, duration=1.369s, table=0, n_packets=0, n_bytes=0,
        priority=43069,ip,in_port=2, nw_src=10.213.165.62,
        nw_dst=192.168.39.163 actions=output:3
    cookie=0x0, duration=1.369s, table=0, n_packets=0, n_bytes=0,
        priority=29603,ip,in_port=2, nw_src=172.22.101.64/30,
        nw_dst=192.168.221.0/24 actions=output:3
    cookie=0x0, duration=1.369s, table=0, n_packets=0, n_bytes=0,
        priority=16854,udp,in_port=2, nw_src=192.168.18.167,
        nw_dst=10.107.149.155, tp_src=143,tp_dst=35414
        actions=output:3
```

Listing B.3: Flowtable before test

```
$ curl -X POST -d '{
    "dpid": 1,
    "priority": 10594,
    "match": {
        "in_port": 2,
        "dl_type": 2048,
        "nw_src": "172.24.8.176/255.255.255.240",
        "nw_dst": "10.164.121.218"
    }
    }' http://192.168.56.1:8080/stats/flowentry/delete_strict
```

Listing B.4: Deleting flowtable entry during test (HTTP POST)

```
Rule MISS received on 1 , but was expected on 3
ENTRY:
    {'priority': 10594, 'cookie': 0, 'actions': ['OUTPUT:3'],
        'match': {'in_port': 2, 'dl_type': 2048, 'nw_src':
        '172.24.8.176/255.255.255.240', 'nw_dst': '10.164.121.218'},
        'packet': {'ip': ipv4(csum=33438, dst='10.164.121.218',
        flags=0,header_length=5, identification=0,
        offset=0,option=None, proto=0, src='172.24.8.177',
        tos=4,total_length=20, ttl=255,version=4)}, 'isDrop': False,
        'outport': '3', 'neighbour': '3', 'dpid': 1, 'in_port': 2}
```

Listing B.5: Output after test pointing out the missing rule

B.7 Drop-test

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows -O OpenFlow13 s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
    cookie=0x4d2, duration=1.68s, table=0, n_packets=0, n_bytes=0,
        priority=1234,ip,nw_src=8.8.8.8 actions=drop
    cookie=0x0, duration=85.149s, table=0, n_packets=3715,
        n_bytes=222900, priority=65535,
        dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
        actions=CONTROLLER:65535
    cookie=0x0, duration=1987.523s, table=0, n_packets=1,
        n_bytes=60, priority=21010,
        ip,in_port=3,nw_src=172.25.107.97, nw_dst=10.28.10.232
        actions=output:2
    cookie=0x0, duration=1987.523s, table=0, n_packets=1,
        n_bytes=60, priority=43069, ip,in_port=2,
        nw_src=10.213.165.62, nw_dst=192.168.39.163 actions=output:3
    cookie=0x0, duration=1987.523s, table=0, n_packets=1,
        n_bytes=60, priority=29603, ip,in_port=2,
        nw_src=172.22.101.64/30, nw_dst=192.168.221.0/24
        actions=output:3
    cookie=0x0, duration=1987.523s, table=0, n_packets=1,
        n_bytes=60, priority=16854,
        udp,in_port=2,nw_src=192.168.18.167,
        nw_dst=10.107.149.155,tp_src=143, tp_dst=35414
        actions=output:3
```

Listing B.6: Flowtable on Switch1 before test

```
ADD CATCH RULE ON SWITCH: 1
version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=0 , cookie=1235 ,
    cookie_mask=0 , flags=0 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(
    actions=[OFPActionOutput(len=16 , max_len=65509 ,
    port=4294967293 , type=0)] , type=4)] , match=OFPMatch(
    oxm_fields={'eth_type': 2048 , 'ip_dscp': 1}) ,
    out_group=4294967295 , out_port=4294967295 , priority=1235 ,
    table_id=0)


ADD CATCH RULE ON SWITCH: 1
version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=0 , cookie=1233 ,
    cookie_mask=0 , flags=0 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(
    actions=[OFPActionOutput(len=16 , max_len=65509 , port=3 ,
    type=0)] , type=4)] , match=OFPMatch( oxm_fields={'eth_type':
    2048 , 'ip_dscp': 1}) , out_group=4294967295 ,
    out_port=4294967295 , priority=1233 , table_id=0)


PACKET OUT FROM SW 1 : version=None , msg_type=None , msg_len=None
    , xid=None , OFPPacketOut(actions=[OFPActionOutput(len=16 ,
```

```
    max_len=65509 , port=4294967289 , type=0)] , actions_len=0 ,
    buffer_id=4294967295 , data=ethernet(dst='b4:7c:7c:54:9c:7b' ,
    ethertype=2048 , src='3f:32:fa:1b:e2:b2') , ipv4(csum=42962 ,
    dst='2.2.2.2' , flags=0 , header_length=5 , identification=0 ,
    offset=0 , option=None , proto=0 , src='8.8.8.8' , tos=4 ,
    total_length=20 , ttl=255 , version=4) , in_port=4294967295)
10
11 REMOVE CATCH RULE ON SWITCH: 1
12 version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=3 , cookie=1235 ,
    cookie_mask=0 , flags=1 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(actions=[OFPActionOutput(len=16
    , max_len=65509 , port=4294967293 , type=0)] , type=4)] ,
    match=OFPMatch(oxm_fields={'eth_type': 2048 , 'ip_dscp': 1}) ,
    out_group=4294967295 , out_port=4294967295 , priority=1235 ,
    table_id=0)
13
14
15 REMOVE CATCH RULE ON SWITCH: 1
16 version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=3 , cookie=1233 ,
    cookie_mask=0 , flags=1 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(actions=[OFPActionOutput(len=16
    , max_len=65509 , port=3 , type=0)] , type=4)] ,
    match=OFPMatch(oxm_fields={'eth_type': 2048 , 'ip_dscp': 1}) ,
    out_group=4294967295 , out_port=4294967295 , priority=1233 ,
    table_id=0)
```

Listing B.7: Output during test

```
1 OFPPacketIn received: switch= 1 buffer_id= 4294967295 total_len= 60
    reason= 1 table_id= 0 cookie= 1235 match=
    OFPMatch(oxm_fields={'in_port': 4294967295}) pkt=
    ethernet(dst='b4:7c:7c:54:9c:7b' , ethertype=2048 ,
    src='3f:32:fa:1b:e2:b2') , ipv4(csum=42962 , dst='2.2.2.2' ,
    flags=0 , header_length=5 , identification=0 , offset=0 ,
    option=None , proto=0 , src='8.8.8.8' , tos=4 , total_length=20
    , ttl=255 , version=4)
2
3 DROP RULE SUCCESS
```

Listing B.8: Result after test

B.8 Loop-test

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows -O OpenFlow13 s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
    cookie=0x4d2 , duration=8.378s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=1234 , ip , nw_src=8.8.8.8
        actions=output:2
    cookie=0x0 , duration=39.038s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=39560 , tcp , in_port=3 ,
        nw_src=172.26.29.192/26 , nw_dst=172.19.198.4 , tp_src=62682
        , tp_dst=23 actions=output:2
    cookie=0x0 , duration=39.038s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=55823 , ip , in_port=2 ,
        nw_src=192.168.85.97 , nw_dst=192.168.226.26 actions=output:3
    cookie=0x0 , duration=39.038s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=37106 , ip , in_port=2 ,
        nw_src=192.168.213.164 , nw_dst=192.168.50.5 actions=output:3
    cookie=0x0 , duration=39.038s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=29915 , ip , in_port=2 ,
        nw_src=172.16.252.192/27 , nw_dst=172.16.14.128/26
        actions=output:3
    cookie=0x0 , duration=39.038s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=22158 , udp , in_port=2 ,
        nw_src=10.100.243.52 , nw_dst=10.125.36.225 , tp_src=68 ,
        tp_dst=63643 actions=output:3
```

Listing B.9: Flowtable on Switch1 before test

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows -O OpenFlow13 s2
OFPST_FLOW reply (OF1.3) (xid=0x2):
    cookie=0x4d2 , duration=4.828s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=1234 , ip , nw_src=8.8.8.8
        actions=output:1
    cookie=0x0 , duration=40.073s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=31231 , ip , in_port=ANY ,
        nw_src=192.168.175.0/24 , nw_dst=192.168.40.229
        actions=output:1
    cookie=0x0 , duration=40.073s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=11650 , ip , in_port=ANY ,
        nw_src=10.246.219.180 , nw_dst=172.23.51.167 actions=output:1
    cookie=0x0 , duration=40.073s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=44684 , ip , in_port=ANY ,
        nw_src=192.168.66.25 , nw_dst=172.22.50.75 actions=output:1
    cookie=0x0 , duration=40.073s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=48723 , ip , in_port=ANY ,
        nw_src=10.251.239.50 , nw_dst=10.201.4.0/24 actions=output:1
    cookie=0x0 , duration=40.073s , table=0 , n_packets=0 ,
        n_bytes=0 , priority=35125 , udp , in_port=ANY ,
        nw_src=10.63.8.0/22 , nw_dst=172.24.22.216 , tp_src=993 ,
        tp_dst=62308 actions=output:1
```

Listing B.10: Flowtable on Switch2 before test

```
ADD CATCH RULE ON SWITCH: 3
version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=0 , cookie=65535 ,
    cookie_mask=0 , flags=0 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(actions=[OFPActionOutput(len=16
    , max_len=65509 , port=4294967293 , type=0)] , type=4)] ,
    match=OFPMatch(oxm_fields={'eth_type': 2048 , 'ip_dscp': 1}) ,
    out_group=4294967295 , out_port=4294967295 , priority=65535 ,
    table_id=0)

ADD CATCH RULE ON SWITCH: 2
version=None , msg_type=None , msg_len=None , xid=None ,
    OFPFlowMod(buffer_id=4294967295 , command=0 , cookie=65535 ,
    cookie_mask=0 , flags=0 , hard_timeout=0 , idle_timeout=0 ,
    instructions=[OFPInstructionActions(actions=[OFPActionOutput(len=16
    , max_len=65509 , port=4294967293 , type=0)] , type=4)] ,
    match=OFPMatch(oxm_fields={'eth_type': 2048 , 'ip_dscp': 1}) ,
    out_group=4294967295 , out_port=4294967295 , priority=65535 ,
    table_id=0)

PACKET OUT FROM SW 1 : version=None , msg_type=None , msg_len=None
    , xid=None , OFPPacketOut(actions=[OFPActionOutput(len=16 ,
    max_len=65509 , port=4294967289 , type=0)] , actions_len=0 ,
    buffer_id=4294967295 , data=ethernet(dst='f6:43:87:31:5a:14' ,
    ethertype=2048 , src='0b:cc:4a:bf:a1:7e') , ipv4(csum=42962 ,
    dst='2.2.2.2' , flags=0 , header_length=5 , identification=0 ,
    offset=0 , option=None , proto=0 , src='8.8.8.8' , tos=4 ,
    total_length=20 , ttl=255 , version=4) , in_port=4294967295)
```

Listing B.11: Output during test

```
OFPPacketIn received: switch= 2 buffer_id= 4294967295 total_len= 60
    reason= 1 table_id= 0 cookie= 65535 match=
    OFPMatch(oxm_fields={'in_port': 1}) pkt=
    ethernet(dst='f6:43:87:31:5a:14' , ethertype=2048 ,
    src='0b:cc:4a:bf:a1:7e') , ipv4(csum=42962 , dst='2.2.2.2' ,
    flags=0 , header_length=5 , identification=0 , offset=0 ,
    option=None , proto=0 , src='8.8.8.8' , tos=4 , total_length=20
    , ttl=255 , version=4)

LOOP FOUND FOR SWITCH 2
PACKET: ethernet(dst='58:a0:1e:fd:a1:41' , ethertype=2048 ,
    src='b1:b4:46:c3:77:24') , ipv4(csum=42962 , dst='2.2.2.2' ,
    flags=0 , header_length=5 , identification=0 , offset=0 ,
    option=None , proto=0 , src='8.8.8.8' , tos=4 , total_length=20
    , ttl=255 , version=4)
ENTRY: {'priority': 1234 , 'cookie': 1234 , 'actions': ['OUTPUT:1']
    , 'match': {'dl_type': 2048 , 'nw_src': '8.8.8.8'}}
```

Listing B.12: Output after test pointing out the loop-causing rule

Appendix C

Code

C.1 Functions

Datapath	
Function	**Description**
getAllDatapaths	Returns a list of all datapath objects
getDatapathByID	Returns a datapath object by ID
Link	
Function	**Description**
getAllLinks	Get all datapath link objects
getLinksByDatapathID	Get datapath links by datapath ID
Neighbourhood	
Function	**Description**
getAllDatapathNeighbors	Get dict of all datapath neighbor (IDs)
getNeighborsByID	Get list of neighbors for a specific datapath ID
getNeighborByPort	Get a datapaths neighbour by port ID
Catch rules	
Function	**Description**
addCatchRule	Install catch rule by datapath object
addCatchRuleByID	Add catch rule by datapath ID
removeCatchRule	Remove catch rule
removeCatchRuleByID	Remove catch rule by datapath ID
Packet out	
Function	**Description**
sendPacket	Send packet by datapath object
sendPacketByID	Send test packet by datapath ID
Parsing	
Function	**Description**
checkDropRule	Check drop rule, rw ft entry and probe
checkUnicastRule	Probe unicast rule
compareMatchPacket	Compare packet with entries in dictlist.
checkForLoops	Iterate flowtable, match and look at outport.
Generator	
Function	**Description**
drawMacAddr	Generate random MAC addresses
drawIPAddr	Generate single IP address or range.
drawPort	Return a well-known pnum or pnum > 1023
makeTestPacket	Generate fake test packet
makeFlowMod	Generate OpenFlow FlowMod request
addressToIPSet	IP address to netaddr.IPSet
sortDictByKey	Sort list of dict by key
packetFromMatch	Make packet from match entry
Gather	
Function	**Description**
removeAllEntries	Emptying out all flow table entries
getAllDatapathID	Get all datapath IDs
getFlowtable	Get each switch's flow table
getFlowtableMatch	Get match field from flow table
getDatapathLinks	Get all connected ports on all switches
getAllDatapathPorts	Get all switches and ports in the topology
getMatchData	Extracts prio, cookie, match and action

Table C.1: Support-functions

C.2 Controller

```
1
2  # imports
3  from ryu.base import app_manager
4  from ryu.controller import ofp_event
5  from ryu.controller.handler import CONFIG_DISPATCHER,
       MAIN_DISPATCHER
6  from ryu.controller.handler import set_ev_cls
7  from ryu.ofproto import ofproto_v1_3, ofproto_v1_2, ether, inet
8  from ryu.lib.packet import packet, ethernet, ether_types, ipv4,
       arp, tcp, udp
9  from ryu.topology import event, switches
10 from ryu.topology.api import get_switch, get_all_switch
11 from ryu.topology.api import get_link, get_all_link
12 import ryu.app.ofctl.api as api
13 import ryu.utils as utils
14
15 import gather, requests, random, json, re, sys
16 import generator, time, gather, netaddr, array
17
18
19 #### CLASS
20 class ctrlapp(app_manager.RyuApp):
21     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
22
23
24     #### INIT
25     def __init__(self, *args, **kwargs):
26         super(ctrlapp, self).__init__(*args, **kwargs)
27         self.isTesting = False
28         self.testingRules =[]
29         self.isTablesPopulated = False
30         self.totalSent = 0
31         self.totalReceived = 0
32         self.starttime = 0
33         self.rulesInstalled = 0
34         self.rulesRemoved = 0
35         self.generateTime = 0
36         self.totalOverlap = 0
37         self.pullFlowTable = 0
38         self.allFlowTables = dict()
39         self.packetGenTime = 0
40         self.skipFirstMiss = 0
41
42
43     #### PACKET IN
44     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
45     def packetIn(self, ev):
46         """ PacketIn message """
47
48         # Parse the incoming packet
```

```
49          msg = ev.msg
50          datapath = msg.datapath
51          ofp = datapath.ofproto
52          parser = datapath.ofproto_parser
53          pkt = packet.Packet(msg.data)
54
55
56          # Empty packets, will/should never happen
57          if len(pkt.get_protocols(ethernet.ethernet)) < 1:
58              return
59          eth = pkt.get_protocols(ethernet.ethernet)[0]
60
61          # Ignore LLDP packets when using verbose mode
62          if eth.ethertype == ether_types.ETH_TYPE_LLDP:
63              return
64
65          # print out the packetin
66          #print ("\n OFPPacketIn received: switch=", datapath.id,
                "buffer_id=",
67          #    msg.buffer_id , "total_len=" , msg.total_len , "
                reason=" ,
68          #    msg.reason , "table_id=" , msg.table_id , "cookie=" ,
                msg.cookie ,
69          #    "match=" , msg.match , "pkt=" , pkt , "\n")
70
71
72          # Populate flow tables, once
73          if eth.ethertype == ether_types.ETH_TYPE_ARP:
74              if (pkt.get_protocols(arp.arp)[0].src_ip == "10.0.0.1"
                    and
75              pkt.get_protocols(arp.arp)[0].dst_ip == "10.0.0.20" and
76              pkt.get_protocols(arp.arp)[0].src_mac ==
                    "00:00:00:00:00:01"):
77                  if not self.isTablesPopulated:
78                      self.populateAllFlowtables()
79                      self.isTablesPopulated = True
80
81
82          # Check trigger conditions to start testing
83          if (eth.ethertype == ether_types.ETH_TYPE_ARP and not
                self.isTesting):
84              if (pkt.get_protocols(arp.arp)[0].src_ip == "10.0.0.1"
                    and
85              pkt.get_protocols(arp.arp)[0].dst_ip == "10.0.0.10" and
86              pkt.get_protocols(arp.arp)[0].src_mac ==
                    "00:00:00:00:00:01"):
87
88                  self.isTesting = True
89                  allSwitches = self.getAllDatapaths()
90
91                  # Time the execution
92                  self.starttime = time.time()
93
94                  # Loop through all switches
```

```
95                  for sw in allSwitches:
96                      print ("Testing switch: " , sw.dp.id)
97                      #if sw.dp.id is 1:
98                      # remove catch rule from self, if any
99                      self.removeCatchRuleByID(sw.dp.id)
100
101                     # Install catch rules on neighbours
102                     allNeighbours = self.getNeighborsByID(sw.dp.id)
103                     for neigh in allNeighbours: # id
104                         self.addCatchRuleByID(int(neigh.lstrip("0")))
105
106                     # Scrape and sort flowtable
107                     flowtable = gather.getMatchData(sw.dp.id)
108                     flowtable = sorted(flowtable,
                            key=generator.sortKey, reverse=True)
109
110
111                     # Loop through flow table entries
112                     for entry in flowtable:
113                         # Generate packet from match field and rules
                                above
114                         pkt = generator.packetFromMatch(entry,
                                flowtable)
115                         self.generateTime = time.time()
116
117                         # add packet to list
118                         entry["packet"] = {"ip" :
                                pkt.get_protocol(ipv4.ipv4())}
119
120                         if entry["packet"]["ip"].proto is 6:
121                             entry["packet"]["tcp"] =
                                    pkt.get_protocol(tcp.tcp())
122                         elif entry["packet"]["ip"].proto is 17:
123                             entry["packet"]["udp"] =
                                    pkt.get_protocol(udp.udp())
124
125                         # is total overlap?
126                         if pkt == -1:
127                             # log and move on
128                             entry["totalOverlap"] = True
129                             self.totalOverlap += 1
130                             break
131
132                         # is drop rule?
133                         if (len(entry["actions"]) is 0 or
                                re.search('CLEAR_ACTIONS',
                                entry["actions"][0]) is not None):
134                             # get match and send packet
135                             self.checkDropRule(entry, pkt, sw)
136
137                         # is unicast
138                         else:
139                             # get match and send packet
140                             self.checkUnicastRule(entry, pkt, sw)
```

```
141
142
143
144                 self.packetGenTime = self.generateTime -
                        self.starttime
145                 print ("PACKET GEN TIME: " ,
                        format(self.packetGenTime, '.5f'))
146
147                 # done testing?
148                 #self.isTesting = False
149
150                 # clean up
151                 #for sw in allSwitches:
152                 #    self.removeCatchRule(sw.dp)
153
154
155         # Check packetin L3 PDU for ToS
156         if msg.reason == ofp.OFPR_ACTION:
157             if len(pkt.get_protocols(ipv4.ipv4)) is not 0:
158                 if pkt.get_protocols(ipv4.ipv4)[0].tos is 4:
159                     self.totalReceived += 1
160
161                     # if probe is not -1, it matched with one in the
                            list
162                     entry = self.compareMatchPacket(pkt)
163                     if entry == -1:
164                         print ("HOUSTON WE GOT A PROBLEM")
165
166                     # drop rule packet?
167                     if entry["isDrop"] is True:
168                         if (datapath.id is int(entry["dpid"])):
169                             # rule caught on cntrl -> success
170                             print ("DROP RULE SUCCESS")
171                         elif (datapath.id is int(entry["neighbour"])):
172                             # rule caught on neighbour -> failed
173                             print ("DROP RULE FAIL. RECEIVED ON , " ,
                                    datapath.id , " BUT WAS EXPECTED ON "
                                    , entry["neighbour"] )
174                         else:
175                             # else? something def wrong
176                             print ("MAJOR FLAW..")
177
178
179                     # forwarding packet?
180                     #elif (datapath.id is int(entry["neighbour"])):
181                     #    print ("RECEIVED " , self.totalReceived)
182                     #else:
183                     #    print ("PACKET FAILED THE RULE CHECK / WRONG
                            NEIGH")
184
185                     # LOOP CHECK
186                     self.loopcheck(pkt, datapath.id, entry)
187
188                     # is last rule?
```

```
189                 if len(self.testingRules) is 0:
190                     print ("TOTAL SENT " , self.totalSent)
191                     print ("RECEIVED LAST " , self.totalReceived)
192                     print ("TOTAL OVERLAPS: " , self.totalOverlap)
193                     print ("TIME ON LINK: " , format(time.time()
                            - self.starttime - self.packetGenTime,
                            '.5f'))
194                     print ("TOTAL RUNTIME: " , format(time.time()
                            - self.starttime, '.5f'))
195
196
197         # Invalid TTL, might be caused by loop
198         elif msg.reason == ofp.OFPR_INVALID_TTL:
199             print ("INVALID TTL")
200             # is test packet?
201             if len(pkt.get_protocols(ipv4.ipv4)) is not 0:
202                 if pkt.get_protocols(ipv4.ipv4)[0].tos is 4:
203                     print ("INVALID TTL ON TEST PACKET")
204
205
206
207         # Table miss, might be shadow or non-working rule? or just a
                miss
208         elif msg.reason == ofp.OFPR_NO_MATCH:
209             # is test packet?
210             if len(pkt.get_protocols(ipv4.ipv4)) > 0:
211                 if pkt.get_protocols(ipv4.ipv4)[0].tos is 4 and
                        self.skipFirstMiss > 0:
212                     # yes its a test packet, investigate..
213                     self.totalReceived += 1
214                     probe = self.compareMatchPacket(pkt)
215                     if probe != -1:
216                         print ("Rule MISS received on " , datapath.id
                                , " , but was expected on " ,
                                probe["neighbour"])
217                         print ("ENTRY: \n" , probe)
218                 self.skipFirstMiss += 1
219
220
221     #### DATAPATHS
222     def getDatapathByID(self, dpid):
223         """ Returns datapath object by ID """
224         return api.get_datapath(self, dpid)
225
226     def getAllDatapaths(self):
227         """ Returns a list of all switch objects """
228         switches = list()
229         for i in get_all_switch(self):
230             switches.append(i)
231         return switches
232
233
234     #### LINKS
235     def getAllLinks(self):
```

```
236         """ Get all link objects """
237         links = list()
238         for i in get_all_link(self):
239             links.append(i)
240         return links
241 
242     def getLinksByDatapathID(self, dpid):
243         """ Get datapath links by object ID """
244         #dp = self.getDatapathByID(dpid)
245         #link = get_link(self, dp.id)
246         link = get_link(self, dpid)
247         return link
248 
249 
250     #### NEIGHBOURHOOD
251     def getNeighborsByID(self, dpid):
252         """ Get list of datapath neighbor (IDs) """
253         neigh = list()
254         for link in self.getLinksByDatapathID(dpid):
255             for k, v in link.to_dict().items():
256                 if k is 'dst':
257                     neigh.append(v['dpid'])
258         return neigh
259 
260     def getAllDatapathNeighbors(self):
261         """ Get dict of all datapath neighbor (IDs) """
262         allNeighbors = {}
263         for d in self.getAllDatapaths():
264             allNeighbors[d.dp.id] = self.getNeighborsByID(d.dp.id)
265         return allNeighbors
266 
267     def getNeighborByPort(self, dpid, port):
268         """ Get dpid from port number """
269         for link in self.getLinksByDatapathID(dpid):
270             if link.to_dict()["src"]["port_no"].lstrip("0") ==
                    str(port):
271                 return link.to_dict()["dst"]["dpid"].lstrip("0")
272 
273 
274 
275     #### CATCH RULES
276     def addCatchRule(self, datapath, prio=None, ckie=None,
        prt=None):
277         """ Install catch rule by datapath object """
278         ofp = datapath.ofproto
279         ofp_parser = datapath.ofproto_parser
280         priority = prio if prio is not None else 65535
281         cookie = ckie if ckie is not None else 65535
282         port = prt if prt is not None else ofp.OFPP_CONTROLLER
283         buffer_id = ofp.OFP_NO_BUFFER
284         match = ofp_parser.OFPMatch(eth_type = 2048, ip_dscp = 1)
285         actions = [ofp_parser.OFPActionOutput(port)]
286         inst =
                [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
```

```
            actions)]
287         req = ofp_parser.OFPFlowMod(datapath, cookie, 0, 0,
                ofp.OFPFC_ADD, 0, 0, priority,
288                     buffer_id, ofp.OFPP_ANY, ofp.OFPG_ANY, 0, match,
                        inst)
289         #print ("\nADD CATCH RULE ON SWITCH: " , datapath.id , "\n"
                , req , "\n")
290         datapath.send_msg(req)
291
292
293     def addCatchRuleByID(self, dpid):
294         """ Add catch rule by datapath ID """
295         self.addCatchRule(self.getDatapathByID(dpid))
296
297
298     def removeCatchRule(self, datapath, prio=None, ckie=None,
                prt=None):
299         """ Remove catch rule """
300         ofp = datapath.ofproto
301         ofp_parser = datapath.ofproto_parser
302         priority = prio if prio is not None else 65535
303         cookie = ckie if ckie is not None else 65535
304         port = prt if prt is not None else ofp.OFPP_CONTROLLER
305         buffer_id = ofp.OFP_NO_BUFFER
306         match = ofp_parser.OFPMatch(eth_type = 2048, ip_dscp = 1)
307         actions = [ofp_parser.OFPActionOutput(port)]
308         inst =
                [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                actions)]
309         req = ofp_parser.OFPFlowMod(datapath, cookie, 0, 0,
                ofp.OFPFC_DELETE, 0, 0,
310                                 priority, buffer_id,ofp.OFPP_ANY,
                                    ofp.OFPG_ANY,
311                                 ofp.OFPFF_SEND_FLOW_REM, match,
                                    inst)
312         #print ("\nREMOVE CATCH RULE ON SWITCH: " , datapath.id ,
                "\n" , req , "\n")
313         datapath.send_msg(req)
314
315
316     def removeCatchRuleByID(self, dpid):
317         """ Remove catch rule by datapath ID """
318         self.removeCatchRule(self.getDatapathByID(dpid))
319
320
321
322     #### SEND OUT PACKET
323     def sendPacket(self, datapath, pkt=None, in_port=None):
324         """ Send packet by datapath object """
325         ofp = datapath.ofproto
326         ofp_parser = datapath.ofproto_parser
327         buffer_id = ofp.OFP_NO_BUFFER
328         actions = [ofp_parser.OFPActionOutput(ofp.OFPP_TABLE)]
329         pkt = pkt if pkt is not None else generator.makeTestPacket()
```

```
330
331         # in port set or use ANY?
332         if in_port is None or re.search('ANY', str(in_port)) is not
                None:
333             in_port = ofp.OFPP_ANY
334
335         req = ofp_parser.OFPPacketOut(datapath, buffer_id, in_port,
                actions, pkt)
336         #print ("\nPACKET OUT FROM SW " , datapath.id , " : " , req)
337         datapath.send_msg(req)
338
339
340     def sendPacketByID(self, dpid, pkt=None, in_port=None):
341         """ Send test packet by datapath ID """
342         self.sendPacket(self.getDatapathByID(dpid), pkt, in_port)
343
344
345
346     #### RULE PARSING
347     def checkDropRule(self, entry, pkt, sw):
348         """ Check drop rule by rewriting the ft entry and probe the
                rule """
349
350         # edge cases for prio # not testable
351         if (int(entry["priority"]) is 65535 or
                int(entry["priority"]) is 0):
352             return -1
353
354         entry["isDrop"] = True
355
356         # pick a random neighbour to (maybe) receive the probe
357         dplink = self.getLinksByDatapathID(sw.dp.id)[0] # pick first
                port
358         dpport = dplink.src.to_dict()["port_no"].lstrip("0")
359         entry["neighbour"] = self.getNeighborByPort(sw.dp.id,
                dpport) # get neigh on link
360         prio = int(entry["priority"])
361         ckie = int(entry["cookie"])
362
363         # choose port
364         if "in_port" in entry.get("match", {}):
365             port = entry["match"]["in_port"]
366         else:
367             port = "ANY"
368
369         # append info
370         entry["port"] = port
371         entry["dpid"] = sw.dp.id
372
373         # add catch rules above (cntrl) and below (neigh) target rule
374         self.addCatchRule(sw.dp, prio+1, prio+1)
375         self.addCatchRule(sw.dp, prio-1, prio-1, int(dpport))
376
377         # send packet
```

```
378         self.sendPacket(sw.dp, pkt, port)
379         self.testingRules.append(entry)
380         self.totalSent += 1
381 
382         # delete rules
383         self.removeCatchRule(sw.dp, prio+1, prio+1)
384         self.removeCatchRule(sw.dp, prio-1, prio-1, int(dpport))
385 
386 
387     def checkUnicastRule(self, entry, pkt, sw):
388         """ Probe unicast rule """
389 
390         entry["isDrop"] = False
391 
392         # ignore out to cntrl
393         isOutput = re.search('OUTPUT', entry["actions"][0])
394         isCntrl = re.search('CONTROLLER', entry["actions"][0])
395 
396         if isOutput is not None and isCntrl is None:
397             entry["outport"] = entry["actions"][0].split(":",1)[1]
398             entry["neighbour"] = self.getNeighborByPort(sw.dp.id,
                    entry["outport"])
399             entry["dpid"] = sw.dp.id
400 
401             # choose port
402             if "in_port" in entry.get("match", {}):
403                 entry["in_port"] = entry["match"]["in_port"]
404             else:
405                 entry["in_port"] = "ANY"
406 
407             # send out the probe
408             self.sendPacket(sw.dp, pkt, entry["in_port"])
409             self.testingRules.append(entry)
410             self.totalSent += 1
411 
412 
413     def compareMatchPacket(self, pkt):
414         """
415         Compare incoming packet with entries in dictlist.
416         If no entry found, return -1
417         """
418 
419         entry = ""
420         # incoming probe, find target entry in list
421         for rule in self.testingRules:
422 
423             # does layer 3 match?
424             packetIP = pkt.get_protocol(ipv4.ipv4())
425             ruleIP = rule["packet"]["ip"]
426             if (ruleIP.src == packetIP.src and ruleIP.dst ==
                    packetIP.dst):
427 
428                 # has layer 4, and is matching?
```

```
429                 if (ruleIP.proto != 0 and ruleIP.proto ==
                        packetIP.proto):
430                     if ruleIP.proto is 6:
431                         packetTCP = pkt.get_protocol(tcp.tcp())
432                         ruleTCP = rule["packet"]["tcp"]
433                         if (ruleTCP.src_port == packetTCP.src_port and
434                         ruleTCP.dst_port == packetTCP.dst_port):
435                             index = self.testingRules.index(rule)
436                             return self.testingRules.pop(index)
437 
438                     elif ruleIP.proto is 17:
439                         udpPacket = pkt.get_protocol(udp.udp())
440                         ruleUDP = rule["packet"]["udp"]
441                         if (ruleUDP.src_port == udpPacket.src_port and
442                         ruleUDP.dst_port == udpPacket.dst_port):
443                             index = self.testingRules.index(rule)
444                             return self.testingRules.pop(index)
445 
446                     # rule is L4, but not matching
447                     print ("rule is L4, but not matching")
448                     return -1
449             index = self.testingRules.index(rule)
450             return self.testingRules.pop(index)
451         # no match found
452         print ("no match found")
453         return -1
454 
455 
456     def loopcheck(self, pkt, dpid, entry):
457         """
458         Method for looping through flowtable, match packet and look
                at outport
459         - pull the flowtable from dpid
460         - loop through and find match for pkt
461         - if any match, loop is found
462         - print the pkt and original entry
463         """
464 
465 
466         # scrape flowtable
467         if dpid not in self.allFlowTables:
468             flowtable = gather.getMatchData(dpid)
469             flowtable = sorted(flowtable, key=generator.sortKey,
                    reverse=True)
470             self.allFlowTables[dpid] = flowtable
471 
472         # create packet from entry and compare with the incoming
                packet
473         for field in self.allFlowTables[dpid]:
474             fieldPacket = generator.packetFromMatch(field,
                    self.allFlowTables[dpid])
475 
476             # proto matching?
477             if (pkt.get_protocol(ipv4.ipv4()).proto !=
```

```
478             fieldPacket.get_protocol(ipv4.ipv4()).proto):
479                 continue
480
481             # check layer 4
482             fieldSport = ""
483             fieldDport = ""
484
485             if fieldPacket.get_protocol(tcp.tcp()) is not None:
486                 fieldSport =
                        fieldPacket.get_protocol(tcp.tcp()).src_port
487                 fieldDport =
                        fieldPacket.get_protocol(tcp.tcp()).dst_port
488             elif fieldPacket.get_protocol(udp.udp()) is not None:
489                 fieldSport =
                        fieldPacket.get_protocol(udp.udp()).src_port
490                 fieldDport =
                        fieldPacket.get_protocol(udp.udp()).dst_port
491
492             if "tp_src" in field["match"]:
493                 if fieldSport != field["match"]["tp_src"]:
494                     continue
495             elif "tp_dst" in field["match"]:
496                 if fieldDport != field["match"]["tp_src"]:
497                     continue
498
499
500             # check layer 3
501             fieldSrcRange = ""
502             fieldDstRange = ""
503
504             if "nw_src" in field["match"]:
505                 fieldSrcRange =
                        generator.addressToIPSet(field["match"]["nw_src"])
506             if "nw_dst" in field["match"]:
507                 fieldDstRange =
                        generator.addressToIPSet(field["match"]["nw_dst"])
508
509             # check if packet resides in the fields IP range, if any
510             pktSrcRange =
                    netaddr.IPSet([pkt.get_protocol(ipv4.ipv4()).src])
511             pktDstRange =
                    netaddr.IPSet([pkt.get_protocol(ipv4.ipv4()).dst])
512
513             overlap = [False, False]
514             for i in fieldSrcRange:
515                 for k in pktSrcRange:
516                     if i == k:
517                         overlap[0] = True
518
519             for i in fieldDstRange:
520                 for k in pktDstRange:
521                     if i == k:
522                         overlap[1] = True
523
```

```
524             # if packet might match with the entry
525             if overlap[0] is True or overlap[1] is True:
526                 isOutput = re.search('OUTPUT', field["actions"][0])
527                 isCntrl = re.search('CONTROLLER', field["actions"][0])
528                 if isOutput is not None and isCntrl is None:
529                     # check neighbour on link
530                     outport = field["actions"][0].split(":",1)[1]
531                     neigh = self.getNeighborByPort(dpid, outport)
532                     if neigh is None:
533                         # not possible to determine
534                         return 0
535 
536                     if int(neigh) == entry["dpid"]:
537                         print ("LOOP FOUND FOR SWITCH " , dpid)
538                         print ("PACKET: " , pkt)
539                         #print ("FIELDPACKET: " , fieldPacket)
540                         print ("ENTRY: " , field)
541                         return 1
542         # no loop, move to next
543         return 0
544 
545 
546 
547     #### POPULATE WITH FAKE FLOWS
548     def populateAllFlowtables(self):
549         """ Populate all datapaths with fake flow table entries """
550         for sw in self.getAllDatapaths():
551             links = self.getLinksByDatapathID(sw.dp.id)
552             for r in range(random.randint(3, 3)):
553                 sw.dp.send_msg(generator.makeRandomFlowMod(sw.dp,
                        links))
```

C.3 Generator

```
1  # Imports
2  import random, ipaddress, netaddr, re
3  from ryu.ofproto import ether
4  from ryu.lib.packet import packet, ethernet, ether_types, ipv4,
       arp, tcp, udp
5  
6  
7  def drawRandomMac():
8      """ Generate random MAC addresses """
9      return "%02x:%02x:%02x:%02x:%02x:%02x" % ( random.randint(0,
           255),
10         random.randint(0, 255), random.randint(0,
               255),random.randint(0, 255),
11         random.randint(0, 255),random.randint(0, 255))
12 
13 
```

```
14 def drawRandomIPaddr(single=False):
15     """
16     Generate either single random IP address or range. High
           probability of
17     small CIDR range, low probability for large range. RFC 1918
           addresses.
18     """
19     draw = random.randint(0, 2)
20     addr = ""
21 
22     if draw is 0: # "class a"
23         addr = "10"
24         for i in range(3):
25             addr += "." + str(random.randint(1, 254))
26     elif draw is 1: # "class b"
27         addr = "172"
28         addr += "." + str(random.randint(16, 31))
29         for i in range(2):
30             addr += "." + str(random.randint(1, 254))
31     else: # "class c"
32         addr = "192.168"
33         for i in range(2):
34             addr += "." + str(random.randint(1, 254))
35 
36     if single: # return if single addr
37         return addr + "/32"
38 
39     draw = random.randint(1, 100)
40     # 10 % for /32
41     if draw <= 10:
42         addr += "/32"
43     # 35 % chance for a /24 subnet
44     elif draw <= 45:
45         addr += "/24"
46     # 8 % chance for /23
47     elif draw <= 53:
48         addr += "/23"
49     # 10 % for /25
50     elif draw <= 63:
51         addr += "/25"
52     # 15 % for /26 or /27
53     elif draw <= 78:
54         addr += "/"+ str(random.randint(26, 27))
55     # 3 % for /22
56     elif draw <= 81:
57         addr += "/22"
58     # 9 % for /28 and /29
59     elif draw <= 90:
60         addr += "/"+ str(random.randint(28, 29))
61     # 1 % for /20
62     elif draw <= 91:
63         addr += "/20"
64     # 2 % for /21
65     elif draw <= 93:
```

```
66          addr += "/21"
67      # 7 % for /30 and /31
68      else:
69          addr += "/"+ str(random.randint(30, 31))
70
71      return str(ipaddress.ip_interface(addr).network)
72
73
74  def drawRandomPort(known=True):
75      """ Returns a well known pnum if known is true, else return
            pnum > 1023 """
76      nums = [20, 21, 22, 23, 25, 53, 67, 68, 69, 80, 110, 123, 137,
            138, 139,
77              143, 161, 162, 179, 389, 443, 465, 587, 993, 995, 989,
                    990]
78
79      if known is True:
80          return random.choice(nums)
81      else:
82          # mix of iana & linux ephemeral ports
83          return random.randint(32768, 65535)
84
85
86  def makeTestPacket():
87      """ Generates a test probe with fake header values """
88      pkt = packet.Packet()
89      eth = ethernet.ethernet(dst='00:00:de:ad:be:ef',
            src='00:00:de:ad:be:ef',
90                              ethertype=ether.ETH_TYPE_IP)
91      ip = ipv4.ipv4(src="1.1.1.1", dst = "2.2.2.2", tos=4)
92      pkt.add_protocol(eth)
93      pkt.add_protocol(ip)
94      pkt.serialize()
95      return pkt
96
97
98  def ddnToCidr(ip):
99      """ Dotted decimal to CIDR """
100     hasRange = re.match(".*/(.*)", ip)
101     if hasRange is None:
102         return "/32"
103     else:
104         cidr = "/" +
                str(netaddr.IPAddress(hasRange.group(1)).netmask_bits())
105         return cidr
106
107
108 def makeRandomFlowMod(datapath, links):
109     """ Generate random fake OpenFlow FlowMod request """
110
111     # ofp and args
112     priority = random.randint(1, 65534)
113     ofp = datapath.ofproto
114     ofp_parser = datapath.ofproto_parser
```

```
115     match = ofp_parser.OFPMatch()
116     kwargs = {}
117
118     # wildcard in_port or actual port
119     portlist = list()
120     for link in links:
121         if "dpid" in link.to_dict()["src"]:
122             if int(link.to_dict()["src"]["dpid"].lstrip("0")) is
                    datapath.id:
123                 if "port_no" in link.to_dict()["src"]:
124                     portlist.append(int(link.to_dict()["src"]["port_no"].lstrip("0")))
125
126     if len(portlist) > 1:
127         kwargs["in_port"] = random.choice(portlist)
128     else:
129         kwargs['in_port'] = ofp.OFPP_ANY
130
131     # draw variations of the 5 tuple
132     kwargs['eth_type'] = 0x0800 # add ip proto
133
134     # layer 3 - every time
135     draw = random.randint(1, 4)
136     if draw is 1:
137         kwargs['ipv4_src'] = drawRandomIPaddr()
138         kwargs['ipv4_dst'] = drawRandomIPaddr("single")
139     elif draw is 2:
140         kwargs['ipv4_src'] = drawRandomIPaddr("single")
141         kwargs['ipv4_dst'] = drawRandomIPaddr()
142     elif draw is 3:
143         kwargs['ipv4_src'] = drawRandomIPaddr("single")
144         kwargs['ipv4_dst'] = drawRandomIPaddr("single")
145     else:
146         kwargs['ipv4_src'] = drawRandomIPaddr()
147         kwargs['ipv4_dst'] = drawRandomIPaddr()
148
149     # layer 4 - are we doing layer 4? 30 % of the time
150     if random.randint(1, 100) <= 30:
151         # 7 out of 10 is tcp
152         if random.randint(1, 10) <= 7:
153             kwargs['ip_proto'] = 6
154             if random.randint(1, 2) is 1:
155                 kwargs["tcp_src"] = drawRandomPort()
156                 kwargs["tcp_dst"] = drawRandomPort(False)
157             else:
158                 kwargs["tcp_src"] = drawRandomPort(False)
159                 kwargs["tcp_dst"] = drawRandomPort()
160         # 3 out of 10 is udp
161         else:
162             kwargs['ip_proto'] = 17
163             draw = random.randint(1, 3)
164             if random.randint(1, 2) is 1:
165                 kwargs["udp_src"] = drawRandomPort()
166                 kwargs["udp_dst"] = drawRandomPort(False)
167             else:
```

```
168                 kwargs["udp_src"] = drawRandomPort(False)
169                 kwargs["udp_dst"] = drawRandomPort()
170
171     # pick a port
172     draw = True
173     while draw:
174         pick = random.sample(portlist, 1)[0]
175         if pick != kwargs["in_port"]:
176             outport = pick
177             draw = False
178
179     # create OF objects
180     match = ofp_parser.OFPMatch(**kwargs)
181     actions = [ofp_parser.OFPActionOutput(outport)]
182     inst =
            [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
            actions)]
183     req = ofp_parser.OFPFlowMod(datapath, 0, 0, 0, ofp.OFPFC_ADD,
            0, 0,
184             priority, ofp.OFP_NO_BUFFER, ofp.OFPP_ANY, ofp.OFPG_ANY,
                0, match, inst)
185
186     # return
187     return req
188
189
190 def addressToIPSet(ip):
191     """ Takes ip addr in string format and returns IPSet object """
192     hasRange = re.match(".*/(.*)" , ip)
193     if hasRange is None:
194         return netaddr.IPSet([ip+"/32"]) # no range, treat as single
                addr
195     else:
196         cidr = "/" +
                str(netaddr.IPAddress(hasRange.group(1)).netmask_bits())
197         ip = re.sub('/(.*)' , cidr , ip)
198         return netaddr.IPSet([ip])
199
200
201 def sortKey(dct):
202     """ Sort list of dictionaries by key """
203     return dct['priority']
204
205
206 def packetFromMatch(match=None, ftable=None):
207     """ Make packet from match entry """
208
209     # check object
210     if match is None:
211         print("match is None")
212         return -1
213     else:
214         # build layer 2
215         pkt = packet.Packet()
```

```
216         pkt.add_protocol(ethernet.ethernet(src = drawRandomMac(),
217             dst = drawRandomMac(), ethertype=ether.ETH_TYPE_IP))
218
219         # remove lower prio ft entries and LLDP rules
220         ftable = [x for x in ftable if x["priority"] >
                match["priority"] if x["match"]["dl_type"] != 35020]
221
222         # extract 5-tuple values from match object
223         matchSrcaddr = addressToIPSet(match["match"]["nw_src"]) if
                "nw_src" in match["match"] else None
224         matchDstaddr = addressToIPSet(match["match"]["nw_dst"]) if
                "nw_dst" in match["match"] else None
225         matchProto = match["match"]["nw_proto"] if "nw_proto" in
                match["match"] else None
226         matchSport = match["match"]["tp_src"] if "tp_src" in
                match["match"] else None
227         matchDport = match["match"]["tp_dst"] if "tp_dst" in
                match["match"] else None
228
229         # declare
230         ipheader = ipv4.ipv4(tos=4)
231         tcpheader = tcp.tcp()
232         udpheader = udp.udp()
233         addresses = { "src" : [] , "dst" : [] }
234         usedports = { "tcp" : [] , "udp" : [] }
235         overlap = { "src" : False , "dst" : False }
236
237         # LOOP compare entries descending
238         for entry in ftable:
239
240             ## check for differences; if any, break out
241
242             # if in_port in both dicts && differs, move on
243             if "in_port" in entry and "in_port" in match:
244                 if entry["match"]["in_port"] is not match["in_port"]:
245                     break
246
247             # if nw_proto differs, move on
248             elif matchProto is not None and "nw_proto" in
                    entry["match"]:
249                 if matchProto is entry["match"]["nw_proto"]:
250                     break
251
252             # if sport differs, move on
253             elif "tp_src" in entry["match"] and
                    entry["match"]["tp_src"] is not matchSport:
254                 if entry["match"]["nw_proto"] is 6:
255                     usedports["tcp"].append(entry["match"]["tp_src"])
256                 else:
257                     usedports["udp"].append(entry["match"]["tp_src"])
258                 break
259
260             # if dport differs, move on
```

```
261             elif "tp_dst" in entry["match"] and
                    entry["match"]["tp_dst"] is not matchDport:
262                 if entry["match"]["nw_proto"] is 6:
263                     usedports["tcp"].append(entry["match"]["tp_dst"])
264                 else:
265                     usedports["udp"].append(entry["match"]["tp_dst"])
266                 break
267
268             # done checking for differences
269             # extract 5-tuple values from the entry to compare
                    against
270             entrySrcaddr = addressToIPSet(match["match"]["nw_src"])
                    if "nw_src" in entry["match"] else None
271             entryDstaddr = addressToIPSet(entry["match"]["nw_dst"])
                    if "nw_dst" in entry["match"] else None
272             entryProto = entry["match"]["nw_proto"] if "nw_proto" in
                    entry["match"] else None
273             entrySport = entry["match"]["tp_src"] if "tp_src" in
                    entry["match"] else None
274             entryDport = entry["match"]["tp_dst"] if "tp_dst" in
                    entry["match"] else None
275
276             # L3 - SOURCE ADDRESS MATCHING
277
278             # total overlap after subtraction? add set to list
279             if matchSrcaddr is not None and entrySrcaddr is not None:
280                 if len(matchSrcaddr - entrySrcaddr) <= 0:
281                     overlap["srcaddr"] = True
282                     addresses["src"].append(matchSrcaddr)
283                 # subtracting (possible) ipset portion
284                 matchSrcaddr -= entrySrcaddr
285
286             # L3 - DESTINATION ADDRESS MATCHING
287
288             # total overlap after subtraction? add set to list
289             if matchDstaddr is not None and entryDstaddr is not None:
290                 if len(matchDstaddr - entryDstaddr) <= 0:
291                     overlap["dstaddr"] = True
292                     addresses["dst"].append(matchDstaddr)
293                 # subtracting (possible) ipset portion
294                 matchDstaddr -= entryDstaddr
295
296         # DONE LOOPING
297
298         # check for total shadowing
299         if overlap["src"] is True and overlap["dst"] is True:
300             return -1 # true
301
302         # L3 - set fake addr if None is set
303         if matchSrcaddr is None:
304             #matchSrcaddr = drawRandomIPaddr(True) <--- sketchy
305             matchSrcaddr = "1.1.1.1"
306         else:
307             # field is shadowed, pick last "good" range
```

```
308             if len(matchSrcaddr) is 0:
309                 addr = list()
310                 for iprange in addresses["src"]:
311                     for ip in iprange:
312                         addr.append(ip.format())
313                 matchSrcaddr = random.choice(addr)
314             else:
315                 # collect ip addresses and mask, could be multiple
                        portions
316                 addr = list()
317                 for i in range(len(matchSrcaddr.iter_cidrs())):
318                     for ip in matchSrcaddr.iter_cidrs()[i]:
319                         addr.append(ip.format())
320                 matchSrcaddr = random.choice(addr)
321 
322         if matchDstaddr is None or len(matchDstaddr) is 0:
323             #matchDstaddr = drawRandomIPaddr(True)
324             matchDstaddr = "2.2.2.2"
325         else:
326             # field is shadowed, pick last "good" range
327             if len(matchDstaddr) is 0:
328                 addr = list()
329                 for iprange in addresses["dst"]:
330                     for ip in iprange:
331                         addr.append(ip.format())
332                 matchDstaddr = random.choice(addr)
333             else:
334                 # collect ip addresses and mask, could be multiple
                        portions
335                 addr = list()
336                 for i in range(len(matchDstaddr.iter_cidrs())):
337                     for ip in matchDstaddr.iter_cidrs()[i]:
338                         addr.append(ip.format())
339                 matchDstaddr = random.choice(addr)
340 
341         # set values and add to header
342         ipheader.src = matchSrcaddr
343         ipheader.dst = matchDstaddr
344         pkt.add_protocol(ipheader)
345 
346 
347         # L4 check proto and set header value
348         # tcp
349         if matchProto is 6:
350             ipheader.proto = 6
351             # src
352             if matchSport is not None:
353                 tcpheader.src_port = matchSport
354             else:
355                 # draw random value && is not in list of used port
                        nums
356                 port = drawRandomPort()
357                 while port in usedports["tcp"]:
358                     port = drawRandomPort() # draw new
```

```
359                 tcpheader.src_port = port
360             # dst
361             if matchDport is not None:
362                 tcpheader.dst_port = matchDport
363             else:
364                 # draw random value && is not in list of used port
                        nums
365                 port = drawRandomPort()
366                 while port in usedports["udp"]:
367                     port = drawRandomPort() # draw new
368                 tcpheader.dst_port = port
369             pkt.add_protocol(tcpheader) # adding header to packet
370 
371         # udp
372         elif matchProto is 17:
373             ipheader.proto = 17
374             if matchSport is not None:
375                 udpheader.src_port = matchSport
376             else:
377                 # draw random value && is not in list of used port
                        nums
378                 port = drawRandomPort()
379                 while port in usedports:
380                     port = drawRandomPort() # draw new
381                 tcpheader.src_port = port
382             if matchDport is not None:
383                 udpheader.dst_port = matchDport
384             else:
385                 # draw random value && is not in list of used port
                        nums
386                 port = drawRandomPort()
387                 while port in usedports:
388                     port = drawRandomPort() # draw new
389                 tcpheader.dst_port = port
390             pkt.add_protocol(udpheader) # adding header to packet
391 
392         # serialize and send
393         #print ("ENTRY: " , match , "\n")
394         #print ("PACKET: " , pkt , "\n")
395         pkt.serialize()
396         return pkt
```

C.4 Scraper

```
1 # imports
2 import json, requests, sys, re, random, generator
3 
4 # vars
5 proto = "http://"
6 ctrl_ip = "0.0.0.0"
```

```
7  port = ":8080"
8
9  # OFCTL_REST
10 getDatapaths = "/stats/switches"
11 getFlowtables = "/stats/flow/"
12 removeEntries = "/stats/flowentry/clear/"
13
14 # REST_TOPOLOGY
15 getLinks = "/v1.0/topology/links"
16 allPorts = "/v1.0/topology/switches"
17
18
19 # REMOVE ALL TABLE ENTRIES
20 def removeAllEntries():
21     """ Method for emptying out all flow table entries """
22     for dpid in getAllDatapathID():
23         url = proto + ctrl_ip + port + removeEntries
24         url = url + str(dpid)
25         try:
26             req = requests.delete(url)
27         except requests.exceptions.RequestException as e:
28             print ("Server error %s " % e)
29             return -1
30
31
32 # GET ALL DATAPATH IDs / SWITCH IDs
33 def getAllDatapathID():
34     """ Get all datapath IDs (list of integers) """
35     url = proto + ctrl_ip + port + getDatapaths
36     try:
37         req = requests.get(url)
38         if req.status_code is 200:
39             return json.loads(req.text)
40     except requests.exceptions.RequestException as e:
41         print ("Server error: %s" % e)
42
43
44 # GET FLOW TABLES
45 def getFlowtable(dpid=None):
46     """ Method for getting each switch's flow table (json objects)
        """
47     url = proto + ctrl_ip + port + getFlowtables
48     if dpid is not None:
49         url = url + str(dpid)
50         try:
51             req = requests.get(url)
52             if req.status_code is 200:
53                 return json.loads(req.text)
54         except requests.exceptions.RequestException as e:
55             print ("Server error %s " % e)
56     else:
57         ftables = {}
58         for dpid in getAllDatapathID():
59             url = proto + ctrl_ip + port + getFlowtables + str(dpid)
```

```
60              ftables[dpid] = getFlowtable(str(dpid))
61          return ftables
62
63
64  # GET FLOW TABLE MATCH (as json/dict)
65  def getFlowtableMatch(dpid=None):
66      """ Get match field from flow table """
67      ftable = getFlowtable(dpid)
68      if dpid is not None:
69          entries = list()
70          for entry in ftable[str(dpid)]:
71              entries.append(entry["match"])
72          return entries
73      else:
74          allEntries = {}
75          for k, v in ftable.items():
76              for a, b in v.items():
77                  entries = list()
78                  for entry in b:
79                      entries.append(entry["match"])
80                  allEntries[k] = entries
81          return allEntries
82
83
84  # GET ALL LINK PORTS
85  def getDatapathLinks(dpid=None):
86      """ Get all connected ports on all switches """
87      url = proto + ctrl_ip + port + getLinks
88      if dpid is not None:
89          if isinstance(dpid, int):
90              dpid = str(dpid)
91          if isinstance(dpid, str) and len(dpid) < 16:
92              while len(dpid) < 16:
93                  dpid = "0" + dpid
94          url = url + "/" + dpid
95      try:
96          req = requests.get(url)
97          if req.status_code is 200:
98              return json.loads(req.text)
99      except requests.exceptions.RequestException as e:
100         print ("Server error %s " % e)
101
102
103 # GET ALL PORTS
104 def getAllDatapathPorts(dpid=None):
105     """ Get all switches and ports """
106     url = proto + ctrl_ip + port + allPorts
107     if dpid is not None:
108         if isinstance(dpid, int):
109             dpid = str(dpid)
110         if isinstance(dpid, str) and len(dpid) < 16:
111             while len(dpid) < 16:
112                 dpid = "0" + dpid
113         url = url + "/" + dpid
```

```
114     try:
115         req = requests.get(url)
116         if req.status_code is 200:
117             return json.loads(req.text)
118     except requests.exceptions.RequestException as e:
119         print ("Server error %s " % e)
120 
121 
122 # GET MATCH DATA
123 def getMatchData(dpid=None):
124     """ Extracts: prio, cookie, match and action & returns list of
            dicts """
125     url = proto + ctrl_ip + port + getFlowtables
126     table = []
127     if dpid is not None:
128         url = url + str(dpid)
129         try:
130             req = requests.get(url)
131             if req.status_code is 200:
132                 req = json.loads(req.text)
133                 for k, v in req.items():
134                     for line in v:
135                         if line["match"]["dl_type"] != 35020:
136                             entry = {}
137                             entry["priority"] = line["priority"]
138                             entry["cookie"] = line["cookie"]
139                             entry["actions"] = line["actions"]
140                             entry["match"] = line["match"]
141                             table.append(entry)
142         except requests.exceptions.RequestException as e:
143             print ("Server error %s " % e)
144     else:
145         raise NotImplementedError
146     return table
```

C.5 Simple topology

```
1  # Mininet topology
2  from mininet.cli import CLI
3  from mininet.log import setLogLevel
4  from mininet.net import Mininet
5  from mininet.topo import Topo
6  from mininet.node import RemoteController, OVSSwitch
7  
8  class simple( Topo ):
9      def __init__( self ):
10         # Initialize topology
11         Topo.__init__( self )
12 
13         # Left switch and host
```

```
        leftHost = self.addHost( 'h1' )
        leftSwitch = self.addSwitch( 's1', protocols=["OpenFlow13"] )

        # Top right switch and host
        topSwitch = self.addSwitch( 's2', protocols=["OpenFlow13"] )
        topHost = self.addHost( 'h2' )

        # Bottom right switch and host
        bottomSwitch = self.addSwitch( 's3',
            protocols=["OpenFlow13"] )
         bottomHost = self.addHost( 'h3' )

        # Add host links
        self.addLink( leftHost , leftSwitch )
        self.addLink( leftSwitch , topSwitch )
        self.addLink( topHost, topSwitch )
        self.addLink( leftSwitch , bottomSwitch )
        self.addLink( bottomHost , bottomSwitch )

topos = {
    'simple': simple
}
```

C.6 Scaled topology

```
# Mininet topology
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.topo import Topo
from mininet.node import RemoteController, OVSSwitch

class simple( Topo ):
    def __init__( self ):
        # Initialize topology
        Topo.__init__( self )

        # name and neighbours
        allSwitches = {}
        allSwitches["s1"] = {"neighbours" : ["s3", "s4", "s5"]}
        allSwitches["s2"] = {"neighbours" : ["s3", "s4"]}
        allSwitches["s3"] = {"neighbours" : ["s1", "s2", "s4", "s5"]}
        allSwitches["s4"] = {"neighbours" : ["s1", "s2", "s3", "s6",
            "s9"]}
        allSwitches["s5"] = {"neighbours" : ["s1", "s3", "s6", "s8"]}
        allSwitches["s6"] = {"neighbours" : ["s4", "s5", "s9"]}
        allSwitches["s7"] = {"neighbours" : ["s5", "s8", "s9"]}
        allSwitches["s8"] = {"neighbours" : ["s5", "s7", "s10"]}
        allSwitches["s9"] = {"neighbours" : ["s4", "s6", "s7"]}
        allSwitches["s10"] = {"neighbours" : ["s8"]}
```

```
25
26          # loop once for generating switches
27          for k, v in allSwitches.items():
28              allSwitches[k]["switch"] = self.addSwitch(k ,
                    protocols=["OpenFlow13"])
29          # loop twice for adding links
30          for k, v in allSwitches.items():
31              for i in v["neighbours"]:
32                  self.addLink(allSwitches[k]["switch"] , i)
33
34          # adding some hosts
35          host1 = self.addHost("h1")
36          self.addLink(host1, allSwitches["s1"]["switch"])
37          host5 = self.addHost("h5")
38          self.addLink(host5, allSwitches["s5"]["switch"])
39
40  topos = {
41      'simple': simple
42  }
```

C.7 Dedicated topoloy

```
1   # Mininet topology
2   from mininet.cli import CLI
3   from mininet.log import setLogLevel
4   from mininet.net import Mininet
5   from mininet.topo import Topo
6   from mininet.node import RemoteController, OVSSwitch
7
8   class simple( Topo ):
9       def __init__( self ):
10          # Initialize topology
11          Topo.__init__( self )
12
13          # Left switch and host
14          leftHost = self.addHost( 'h1' )
15          leftSwitch = self.addSwitch( 's1', protocols=["OpenFlow13"] )
16
17          # Top right switch and host
18          topSwitch = self.addSwitch( 's2', protocols=["OpenFlow13"] )
19          topHost = self.addHost( 'h2' )
20
21          # Bottom right switch and host
22          bottomSwitch = self.addSwitch( 's3',
                protocols=["OpenFlow13"] )
23           bottomHost = self.addHost( 'h3' )
24
25          # Top left switch
26          topleftSwitch = self.addSwitch( 's4' ,
                protocols=["OpenFlow13"])
```

```
27
28          # Add host links
29          self.addLink( leftHost , leftSwitch )
30          self.addLink( leftSwitch , topSwitch )
31          self.addLink( topHost, topSwitch )
32          self.addLink( leftSwitch , bottomSwitch )
33          self.addLink( bottomHost , bottomSwitch )
34          self.addLink( topleftSwitch , leftSwitch )
35
36  topos = {
37      'simple': simple
38  }
```