Data plane verification in software-defined networking

Martin Rødvand

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 2017

Data plane verification in software-defined networking

Martin Rødvand

Data plane verification in software-defined networking

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

Printed: Reprosentralen, University of Oslo

Abstract

Network failures are costly and inconvenient to any business or customer. The troubleshooting tools available to the network operator have not evolved significantly since their introduction 30 years ago. Simple tools like ping, traceroute and tcpdump are still used in lieu of something better.

Software-defined networking, with its separate control plane and data plane is making its entry into enterprise networks and there is a need for more modern tools with a more holistic view of the network. Proactive network management is sought after in a hectic workday.

Two different methods for testing the forwarding logic of network devices in a software-defined network are tested in this thesis. The network path for each flow entry on a switch is predicted and verified with real network data to create real-world situations. The goal is to detect any change in forwarding logic, caused either by software bugs, or failures in the network such as link failures or misconfigured rules.

The two methods used both create a number of packets to test the network with real traffic. The first method creates a large number of test packets and whilst being very thorough, the generation of so many packets is slow and labour intensive. The second method creates test packets by selecting only certain packets from each flow entry and hence reduces the time and effort it takes to create the packets. This method is not as thorough and may not catch all cases where the forwarding logic fails.

Two failure scenarios were created for detection of link failure and wrong rule order. Both failure scenarios were detected by the software and provided output for the network operator to further troubleshoot.

This project has developed software that has proved itself suitable for testing in a virtual environment, but it is not applicable to a real-world network as it currently stands.

Acknowledgements

I would like to extend my thanks to the following people and organisations for their support during this period as a student at the Oslo and Akershus University College:

- **Anis Yazidi** - For bringing this topic to my attention and becoming my thesis supervisor. He has always been able to enlighten and motivate me.
- **Ramtin Aryan** - For supervising my thesis along with Anis, and always having good suggestions and relevant input to the problems faced.
- **Oslo and Akershus University College and NUUG** - For allowing me to travel to the LISA conference to get a real motivation boost and making me realise this is the work I want to do.

Contents

List of Figures

Listings

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Network connectivity has become a necessity in today's society and network failures can have a big impact on society both practically and financially. Important infrastructure relies on well connected and stable networks. With networks becoming increasingly complex, faults and outages are equally complex to troubleshoot. Downtime and outages cost businesses and society at large an estimated $700 billion every year[8]. Outages in the critical infrastructure of health care services could have debilitating or even fatal consequences.

Even though networks have grown in complexity, the same troubleshooting tools that were available in the early eighties are still being used today. The evolution of troubleshooting tools has not kept up with the complexity of the networks.

Outages and downtime is expensive, and the need for ways to verify a network's ability is required. While the implementation of networks can be audited, the verification and proof of a network's functionality is harder to prove. Network software is riddled with bugs that can impact network functionality. The ability to uncover and find these situations where an edge case or a bug impacts the forwarding logic of a network device is crucial to creating a more stable infrastructure.

With the introduction of software-defined networking, a more coherent method of controlling the network is available. This requires the creation of a brand new set of troubleshooting tools. The ability to rapidly verify that the network is functioning as intended: either on demand or continuously, is an area in rapid development.

In addition to the complexity of network growth, operators are asked to do more with less. Larger networks are being handled by fewer engineers and this requires better management solutions. Additional considerations need to be made with regard to security issues. With the Internet of Things making its way into today's networks, the need to have strong and well-managed security policies and methods of enforcing and verifying these policies is important.

1.2 Problem Statement

This thesis aims to assist in the troubleshooting of software-defined networks by verifying network policies implemented by controller software.

1. How can we ensure that network policies implemented by controller software act correctly in software-defined networking devices?

2. How can we test the implemented network policies in an efficient manner?

Software-defined networking devices are devices managed by controller software where the forwarding plane and control plane are separated.

The *network policies* are the policies defined by an organisation as to what is accepted traffic on the network. The high level policy is translated into an operational policy by network operators, and is usually input in a similar manner as firewall rules.

Controller software is a piece of software responsible for communicating with the network devices that are enforcing policies on the network. These devices will usually be switches or firewalls, but could extend to other devices as well. In order to implement, see, and verify the existence of a network policy the controller software needs to be programmable.

A network device *acts correctly* by performing the actions specified in the network policy.

Testing the policies in an *efficient manner* implies that the tests can be run at a frequent interval in order to semi-continually check the network operation.

Chapter 2

Background

2.1 Networking

The goal of any network is to deliver information from point A to point B. Different networks have different methods of transporting the data, but the overall goal is the same.

Since the migration to TCP/IP[6] in 1983 for the ARPANET, more and more of the networks around the world are based on the TCP/IP stack. The public Internet is based on IP and while there has been competing network technologies such as IPX and AppleTalk, these technologies are not in extensive use in public networks today.

The evolution of the Internet has led to many different types of devices being used on today's network: devices ranging from kitchen appliances to mobile phones. The basic functionality of the network is dependent upon the routers and switches. However, the increased complexity and increased traffic has required a transformation of the equipment used to serve the traffic, and has led to the development of new technologies. Increased security awareness prompted the introduction of firewalls, whilst the exhaustion of IP address has introduced features such as Network Address Translation (NAT) and private addressing.

2.1.1 Packets

Internet Protocol[20] (IP) packets have, since the transition to the TCP/IP stack, been the core of networks. The main point with an IP packet is to transfer data from A to B. To do that it includes a number of fields, the most important ones being the source and destination address. In addition to these fields the IP header has a field to denote how far into the network a packet should go before being dropped (Time to live-field) as well as a field to announce the IP version used (4 or 6). The IP header is limited to a minimum of 20 bytes and a maximum of 60 bytes.

2.1.2 Routers

Routers are the network devices responsible for routing the packet onwards to the correct destination. Networks commonly consist of internal routers

using an Interior Gateway Protocol (IGP) such as OSPF, RIP, IS-IS, or static routing. Organisations will commonly have one or multiple routers connecting them to the Internet and these devices are responsible for announcing the address prefixes assigned by the Local Internet Registry. The announcement of these addresses is done through the Border Gateway Protocol (BGP)[31].

2.1.3 Switches

Switches are used to increase the port capacity, either for server access in the data centre, or user access in the office. These network devices are known for their capability to push large amounts of traffic quickly. They are built with specialised hardware (ASIC) enabling them to move traffic without having to use CPU processing power. Traditionally switches have had layer 2 functionality only, but in the recent years switches have acquired layer 3 functionality as well, and in some cases switches are able to route traffic equally as good as a router.

Layer 2 switches are used for providing access within a network segment. An important function of the switch is the ability to use VLANs to divide a larger network into smaller segments, thereby minimising the failure domain as well as creating boundaries between network segments.

Hubs are the precursors to switches. The disadvantage of a hub is the requirement to forward all incoming traffic to all neighbouring ports. Switches are able to be more selective when forwarding traffic through the use of a MAC address table. Switches are also full duplex whilst hubs are only half duplex. This greatly increases the capacity and performance.

2.1.4 Firewalls

A firewall is a device used to inspect and protect areas of the network from specific traffic. This is traditionally done using the five tuple setup, meaning five fields of data used to inspect the traffic. These fields are: source IP, destination IP, source port, destination port and protocol. In today's networking, firewalls are found in virtually every network, and in most cases appear transparent to clients.

Firewalls have evolved significantly and today's firewalls have capabilities to do more than just the simple source/destination/port evaluation. They are able to move up the networking stack and into the application stack to gain visibility and the ability to stop more advanced threats. This makes the firewall a more advanced networking device, enabling it to deal with various issues as they arise. The increased feature set also requires more processing power and more efficient rule processing by the firewall.

2.1.5 Internet routing

To transfer a packet from point A to point B in a network, it is necessary to have some mechanisms in place. Each host on the network is in possession of an IP address, and each host communicating on the

Internet communicates using an unique IPv4 or IPv6 address. Routing of the packets is carried out by by network devices, **routers**, with routing protocols implemented. These routers are utilising the BGP to communicate with each other, and to decide where to send each data packet.

Currently the number of IPv4 prefixes on the Internet has passed 650,000, while the number of IPv6 prefixes is just below 40,000[16].

2.2 Software-Defined Networking

Software-defined Networking (SDN) is defined by the Open Networking Foundation as "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices"[29]. The network operation is moving away from managing multiple boxes one-on-one with different interfaces into managing a huge number of boxes through a single interface. Administrative changes may then be carried out at a higher level than previously possible, and consequently operators will move further away from the traditional command line interfaces.

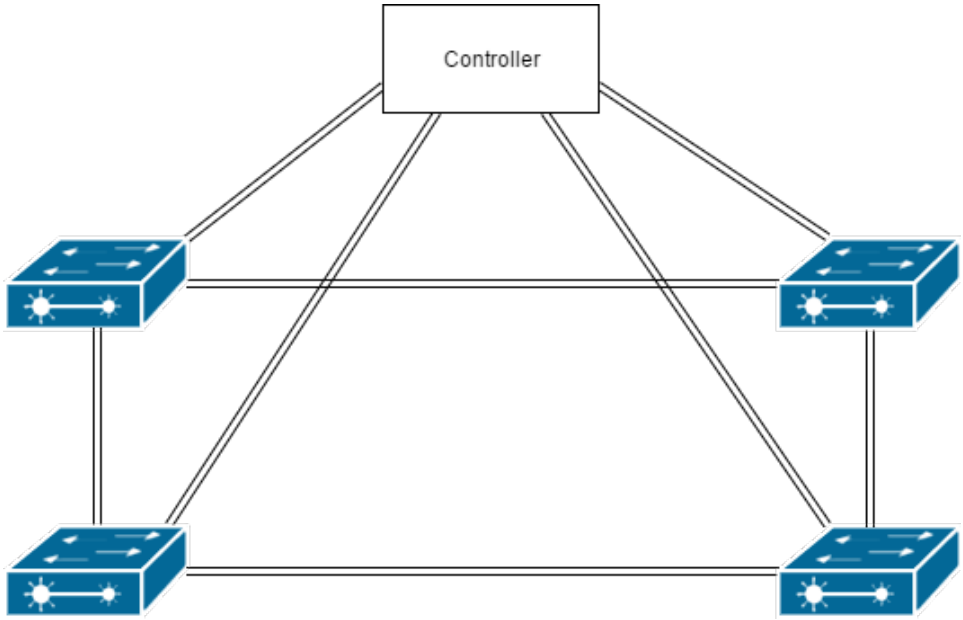

Figure 2.1: Simple view of a SDN topology with a controller

As with all terms gaining attention, the big industry vendors are pushing for all new appliances to adhere to the definition of SDN. However, in order to have SDN, it is not an absolute necessity to introduce new equipment in the network, it is only required to move the control plane logic from each separate device and create a coherent control plane. This means the principles of SDN can be introduced in traditional networks.

SDN brings several large advantages. Adding programmability to the network enables a quicker turnaround and faster deployment of

applications. Previously, operations teams could spend hours on deploying the infrastructure required for new hosts on the network, however with SDN this process can be automated and made ready in minutes. With programmability and consistent deployments the number of human errors decrease as well, making for a more stable network.

2.2.1 History of Software-Defined Networking

The principles behind SDN were introduced in a Stanford paper about Ethane[10], a precursor to OpenFlow. The idea of centralised management has been present for a long time in networking. The wish is for operators to manage and ensure the correct configuration is present across multiple devices.

This is not the first attempt made to automate the networking space, there have been multiple efforts to create more programmable networks. One of the technologies currently employed for monitoring, Simple Network Monitoring Protocol (SNMP)[11], has included methods for programming network devices since its introduction in the early 1990s. However, other technologies and approaches have had trouble with the lack of major vendor support. ForCES[3], an approach for standardising the communication between network elements did not gain enough traction with the major vendors to have an impact in the networking space. Vendors have instead attempted to introduce their own programming interfaces, like Juniper and Brocade who introduced NETCONF[14] as an RFC in 2011. The work on NETCONF started in 2002 with the NETCONF working group. This has enabled users of Brocade and Juniper equipment to automate various tasks using a well-defined interface. Other vendors have implemented the NETCONF technology as well, so this a technology used to increase the programmability of networks which has gained some popularity within the community.

NETCONF is based on the YANG data modelling language[5], a modelling language which other vendors have embraced lately. The structured language enables vendors to creates models based on devices where programmability and centralised management was not built into the device from the start. This increases the backwards compatibility and could enable a better transition for companies not ready to do a complete network refresh.

2.2.2 Architecture

The architecture of a network is based on the separation of control plane and forwarding plane. As figure 2.2 shows, the control plane and forwarding plane are separate. The network devices are acting on behalf of the controller software and are not taking any decision on their own. All of the logic behind the forwarding of data packets is located within the controller, whilst the network devices are just following instructions and pushing packets as based on the decisions made by the controller.

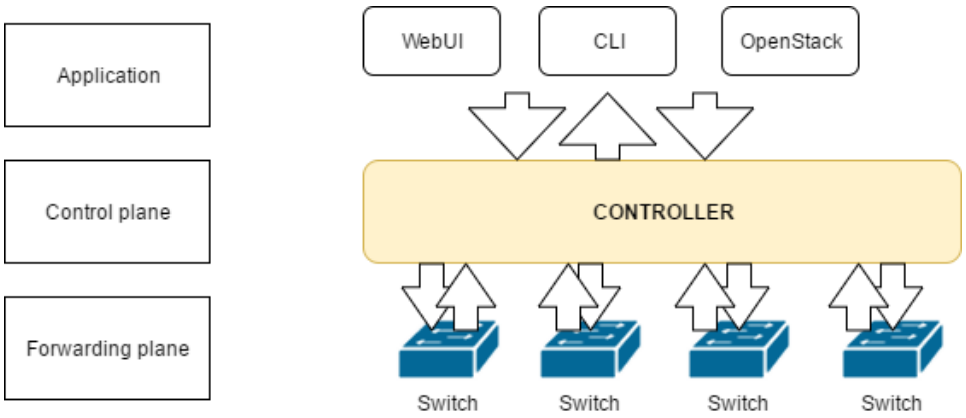

Figure 2.2: A simplistic view of the SDN architecture

Name	Company	Description
NOX[17]	OSS	One of the original OpenFlow controllers
POX[36]	OSS	Can also work as a OpenFlow switch
Ryu[32]	OSS	A component based SDN framework
Beacon[4]	OSS	Developed at Stanford University as a controller to support development
Floodlight[15]	OSS	Forked from Beacon. Big Switch Networks use it as a foundation for their controller
NorthStar	Juniper	Controller for Junos OS
APIC	Cisco	Controller software for Application Centric Infrastructure (ACI)
OpenContrail		A network virtualisation platform for the cloud
Kytos[23]	OSS	A controller developed by researchers at Sao Paulo State University

Table 2.1: SDN controllers

2.2.3 Controller software

Software-Defined Networks rely on a central controller used for managing policies. This controller is responsible for communication with the network devices and ensuring the consistency and correctness of the network.

A number of controller software exist on the market today, ranging from open source products supporting OpenFlow to vendor-proprietary products that only support the vendors' own technologies. A selection of available controller software is shown in table 2.2.3.

2.2.4 Network functions virtualised

NFV, network functions virtualised, is a shift on how operators organise networks[26]. Functions which have previously been run on dedicated and specialised hardware has been exchanged for commodity hardware with virtualisation functions running virtual switches, routers and firewalls.

Whilst NFV is not directly coupled with SDN, the push toward non-specialised hardware and open source is interesting and a big shift in the business.

A simple example of a possible implementation of NFV is the traditional branch office. Traditionally, the branch office has had a router, switch and a rack with servers providing access to files and backup services. If the branch was far from the main office or data centre, the branch may have had a WAN optimisation device as well. Today, a number of these functions can be moved into a single box, where the WAN optimisation, the servers, the firewall and the routers are virtualised. Being able to reduce the number of physical devices on site and running network functions on commodity hardware makes it easier to operate. With commodity hardware supported by multiple vendors companies have a greater opportunity to change away from a network which has traditionally been very much locked into a single vendor and specialised hardware from said vendor.

2.2.5 SD-WAN

Software-defined WAN is a subsection of software-defined networking. The idea behind SD-WAN is to make smarter routing decisions in wide area networks in order to achieve better application performance. Branch offices commonly have multiple WAN connections, each with a different service-level agreement and performance. Using solutions branded as SD-WAN, the traffic can be load-balanced or shifted across these connections in a way that improves performance.

The overall performance is increased because the solution is aware of its surroundings. If the WAN connection using the Internet is performing better than the MPLS connection provided, the application will run over the Internet instead. This can also help utilise multiple connections in a more cost effective way.

In short, SD-WAN enables dynamic routing based on application performance in the WAN.

2.3 Mininet

Mininet is a tool to create and emulate network topologies using minimal resources. Mininet enables users to create realistic virtual networks in little time. It creates networks with the capabilities of OpenFlow and with different configuration options to use different controllers: from built-in controllers to remote controllers.

A simple topology can be create with the command **mn**. This creates a topology with two hosts on one switch. Mininet has a Python API and custom topologies can be created using this. It supports different OpenFlow versions and different switches.

Mininet was developed as a tool to rapidly create network topologies without requiring large clusters of hardware to do so. The original paper detailing Mininet is called "A network in a laptop: rapid prototyping

for software-defined networks"[24] and highlights the small amount of resources needed to run Mininet.

2.4 OpenFlow

OpenFlow is a communications protocol used to separately manage policies and flows in switches. The protocol is based on the work on Ethane at Stanford University in the mid 2000s[10]. From 2011 onwards, the Open Networking Foundation has been responsible for its evolution and oversight.

2.4.1 Versions

As with all protocols, OpenFlow has gone through a process of maturing. This means the initial version 1.0[19] of OpenFlow is now superseded by other versions, with the current version 1.5.1[28] of the specifications being released in April 2015. The vendors spend different amount of time deploying newer versions. Many vendors keep supporting OpenFlow version 1.0, with a number of vendors now supporting version 1.3 as well.

It is not only the switches needing to support the OpenFlow version, the controller software is also required to act according to specifications. The switch and controller will negotiate on the version to be used between them, and most of the time agree on the newest version supported by them both.

2.4.2 Switches

The OpenFlow specification is a specification for a switch. OpenFlow switches can be developed by anyone as long as they follow the specifications listed. One of the more commonly used switches is the Open vSwitch developed by the Linux Foundation. Open vSwitch is currently the default switch in Mininet.

It is important to distinguish between physical and virtual switches, and Open vSwitch is a virtual switch. Hardware vendors have been working on creating physical hardware adhering to the OpenFlow standard. Whilst a number of smaller vendors have embraced the OpenFlow standard, the traction within the major vendors such as Cisco and Juniper varies.

Switches are not necessarily purely OpenFlow. They could be OpenFlow-hybrid switches, meaning they have capabilities outside of just the OpenFlow standard. In these cases it may be possible to activate OpenFlow on just certain ports, while the rest of the switch is performing more traditional tasks.

2.4.3 Flows

A core piece of OpenFlow switches is the flow entries. These are the forwarding tables used for decision making on the switches. A flow entry

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Table 2.2: OpenFlow flow table

can be compared to a mix of a firewall rule, policy based forwarding, and a mac address look-up.

The match fields are fields matching on packet meta data. This could be source network, destination network, protocol, port and similar data. Priority sets the precedence for evaluation of the flows. Instructions are actions to be taken on the packet. This could be a number of different events, like decrement TTL, add VLAN tag, or just simply drop the packet.

Flow entries are installed and manipulated by the controller software. Timeout values can be attached to the flows, making them present in the flow table for only a set time.

The default action when a packet does not match an entry in the flow table is to discard the packet. The flow table must support a table-miss flow entry: an entry to handle packets having missed all other entries. This flow entry is located at the end of the flow table with a priority of 0. This entry supports the actions like a normal flow entry, making it possible to adjust the entry to forward packets to the controller, or perhaps even to discard the packet.

2.5 Traditional Troubleshooting Tools

2.5.1 Ping

Ping is a tool used to check connectivity from one host to another. It utilises ICMP to send an ICMP Echo Request. The other host is expected to reply with an ICMP Echo Reply. Ping is ubiquitous and is generally available on all hosts, including routers and switches. This makes it a quick and well-known tool for checking network connectivity between hosts.

However, ping uses ICMP, which is a protocol used for diagnosis and troubleshooting. This leads to network operators not prioritising these packets, thereby making ping testing an unsuitable tool for testing network or application performance.

2.5.2 Traceroute

Traceroute is a tool used to check the path a packet traverses through the network. It does this by sending a packet with the destination IP address in the IP header, but starts off with a small TTL (time to live). This prompts the network device receiving the packet to return a ICMP Time Exceeded when the TTL gets to 0. Similar to ping, traceroute is present in almost all devices operating on a network.

2.5.3 NetFlow/sFlow

Netflow and sFlow are network protocols used to sample traffic data from a router. This helps operators gain insight into which ports/protocols are being used by specific users connecting through a router. It can also assist in cases with over-utilisation of a link and with capacity planning. Netflow is a Cisco proprietary protocol while sFlow is an industry standard, but both perform the same function.

A flow installation requires configuration on the device sending the flow data, as well as a central receiver of the data. The data being sent is a sample, selected at an interval specified by the operator. This means a flow installation will create additional bandwidth constraints on the network, due to the additional packets being sent to the monitoring system.

2.5.4 Tcpdump

Tcpdump[35] is a tool to capture network traffic on a host. It enables operators to capture raw traffic on the wire for analysis. This tool is used in troubleshooting scenarios where it is necessary to look at each packet in order to figure out the problem.

Tcpdump was developed in the late 1980s to assist with troubleshooting issues on the ARPAnet.

The most powerful functionality of tcpdump is the filtering ability. This enables operators to filter the traffic based on their previous troubleshooting efforts and limit it to the parts they have an interest in. In an environment with multiple network interfaces, tcpdump can be attached to any of the network interfaces and capture the traffic.

2.6 Related work

2.6.1 VeriFlow

VeriFlow[22] is a tool used for verifying network correctness before the rules and logic are implemented in the network devices. The tool will check the changes made to the network for correctness or anomalies before allowing the changes to be deployed.

A challenge with this approach is the demand for changes to be deployed without any latency. It is important that central changes on the controller are pushed to the network as soon as possible, as a delay may create issues with routing, failover or security policies.

To be able to verify the changes made by the centralised controller, VeriFlow is implemented as a layer between the controller and the network devices. This enables VeriFlow to intercept all communication between the controller and network devices, giving VeriFlow the opportunity to discard changes that do not adhere to a standard, or changes found to be dangerous to the network.

Faced with the timing challenge, VeriFlow is splitting the network into smaller pieces to be able to run the verification process on smaller network

segments. These smaller segments are called Equivalence Classes (EC), and each EC is a set of packets with the same forwarding action. To store these ECs, VeriFlow uses an ordered tree structure. By using ECs and a tree structure, VeriFlow can find the network segment affected by a new rule fast, and calculate the impact this rule will have on the network. In addition to the ECs, VeriFlow creates a forwarding graph to represent the forwarding actions of an EC.

Combining the EC and forwarding graph, VeriFlow is able to check rules when they are being installed for inconsistency or errors. A few example queries that can be run are listed:

- Basic reach-ability
- Loop-freeness
- Consistency
- VLAN separation

In the current implementation, performance based queries are not supported.

VeriFlow is a tool which promotes its ability to do the rule check in near real time. However, when large changes in the network happen, VeriFlow is unable to keep up and it is necessary to allow rules to be installed without verification. Instead, the verification process will run in parallel, at the same time as the rules are installed.

The tool is deployed as a proxy process, enabling it to be used without modifying the OpenFlow application. In addition to the proxy process, the authors implemented the tool within the NOX OpenFlow controller. The important part of both of these implementations is the ability to intercept the necessary messages for VeriFlow to run the verification queries selected. As an extension of the application, an API is developed enabling operators to create their own queries using VeriFlow.

The performance of VeriFlow is shown to be essentially dependent on the number of ECs created. It is able to handle a normal set of network data without much latency. The only issue which occurs is the processing time for major network changes. Big events requiring multiple convergence events in the case of link/node failures affect a big number of ECs and therefore cause an increase in the latency.

2.6.2 Automatic Test Packet Generator

ATPG[38] is a tool used to generate test packets to run through a network. This is useful for testing the forwarding rules as well as security rules throughout throughout the network. A general issue in traditional networking is being unable to test a certain policy without real world traffic. ATPG is an extension of the tools Hassel and NetPlumber presented in the "Header Space Analysis"[21] paper. These tools are used for verification of the control plane (Hassel does this offline, NetPlumber real-time).

The purpose of ATPG is to verify that the network is compliant with the policy created. The step from a written policy to the implementation in the networking equipment is not always perfect, making many policies invalid. It could also be that a policy has been implemented correctly, but that a later change has left the network in a non-compliant state. ATPG is able to run checks in a efficient and proactive way, giving network administrators the possibility to catch these errors.

ATPG uses the header space framework[21] to create its network model. Its network model consists of packets, switch rules, rule history and topology. Combined, these are the building blocks of the network model.

ATPG uses the network model to generate packets to match each forwarding rule by a minimum of one test packet. This does require test agents to be present at all end points in the network in order to verify the probes sent. The generator is able to create packets to test various functions of the network. It can test for correctly behaved forwarding (forward rule), correctly behaved forwarding over a specific link (link rule), as well as testing for correctly dropped packets (drop rule). In addition to these tests, ATPG can also be used to verify various performance based rules, such as congestion, available bandwidth and service priorities.

2.6.3 OFRewind

OFRewind[37] is a tool that enables network administrators to record and replay traffic patterns in the network. This application consists of two parts, Ofrecord and Ofreplay, used for recording traffic and replaying the traffic.

OFRewind is used as a tool to help localise an issue in order to assist the operators in resolving network problems. The application runs as a proxy between the controller and the network devices, thus enabling it to intercept any message to and from the controller software. When traffic is selected to be recorded, the application will send the necessary control messages to the switches ordering them to mirror selected traffic to a *Datarecord module*. The control traffic is recorded in order to keep a record for replay scenarios. This traffic is recorded on local storage attached to the OFRewind installation. If OFRewind is to record traffic from the switches, OFRewind will modify the messages going to the switches to order a duplication of the desired traffic pointing it to a *DataStore*.

A challenge with the recording data is the timing. Between Ofrecord and the DataStores it cannot be assumed that the time is synchronised. Therefore OFRewind utilises time binning markers and flow creation markers to make sure the replays will order the data plane events to control plane events.

To replay the data, Ofreplay re-injects the data into the network, with the control plane messages injected by Ofreplay, and the data plane traffic being replayed by the Datareplay component of the DataStores.

The amount of data required for recording is naturally dependent on the traffic marked for recording, but it also requires some data for the OpenFlow and sync marker messages to be able to reproduce the session. A test done by the authors on the Stanford network reveals the overhead to

be just over 1%. As the flow markers are synced between all the DataStores, this overhead should not be anything to stop the deployment of more DataStores in the network.

In comparison with traditional networking troubleshooting tools, the tool `tcpdump` is the application matching OFRewind closest. The difference is the centralised control nature of OFRewind. As OFRewind is placed close between the network devices and the controller, the controlling of the session recordings are much more centralised than the classic `tcpdump` where an event traditionally is recorded and analysed in the perspective of an end host. OFRewind also introduces the ability to see the control messages, and replay network events.

2.6.4 SDN Traceroute

SDN traceroute[2] attempts to do what the traditional traceroute tool does: trace the path of a packet. SDN traceroute will extend the functionality such that it will show the full path, not just layer 3 hops.

An important aspect with the SDN traceroute application is that it does not utilise the controller model to replicate the path, it actually sends packets to the switches and receives replies back, creating a trace using the actual forwarding logic on the data plane.

The way it does this is by assigning each switch in the network with a "colour". This is a tag enabling SDN traceroute to map the path. It is important that no switch is the same color as any of its adjacent switches. The way the "colouring" is done is by utilising the three bits VLAN priority field.

To be able to tell when a traceroute packet is received on the switch, a rule is installed matching all colours except the switch's own. This rule is given top priority, so if a packet arrives with one of the adjacent colours it will match the rule. The number of rules installed on the switch in relation to this part would be the same as the number of adjacent switches. As these rules are set to match on the "colour" tag, the normal production traffic is not affected.

When conducting the traceroute, the packet triggered either by the controller or via an API. The switch where it starts off is identified, and the colour of the switch is attached to the probe. The packet includes a destination as normal, and as the probe has the switch's own colour in it the rule installed previously will not match. It will therefore use the actual forwarding table of the switch making sure the probe is a real life test. As the probe reaches the neighbouring switch it will match the installed rule. The action of this rule will be to send the probe back to the controller, where the controller records the path taken. The controller will then modify the probe with the current switch's colour and send it back. This will repeat until a timeout occurs or a repeated route is recognised (routing loop).

SDN traceroute is a simple tool, but it is also very powerful as it utilises the actual forwarding table of the switch while at the same time being a recognisable tool for any network administrator.

2.6.5 NICE

NICE (No bugs In Controller Execution)[9] is a tool used for checking the network state for correctness when applying new instructions or changes in the network. To do this, the application utilises model checking and symbolic execution before identifying violations.

The model checking is done by identifying system states and the transitions between them. The controller, switches, and end hosts are modelled, but to minimise the size and complexity, the models are created in a way which removes the non-essential details. The system transitions are executed through the symbolic execution, a way of testing all possible outcomes of an event. To minimise the number of tests needed, only the inputs which would create different code paths are executed.

The models and symbolic execution is used together with correctness checks to provide the NICE tool. Nice has a library of correctness properties which can be used for various different OpenFlow applications. This includes modules able to check for no forwarding loops, no black holes, no forgotten packet, and more.

The implementation of this tool is done in Python, for the NOX controller platform[17]. The authors of this tool tested the tool in three different applications; a MAC-learning switch, a server load-balancer, and energy-aware traffic engineering. Using this tool they were able to uncover eleven bugs, ranging from flow entries that did not timeout (violating the "no back holes"-property) to forgotten ARP packets (violating the "no forgotten packets"-property).

2.6.6 NetSight

NetSight[18] is a tool for capturing packet histories to a database, enabling others to take advantage of useful information in case of network issues. By having recorded the flow of a packet through the network, tools can be built using the NetSight API to query the network for information. NetSight can be seen as a basis for network debugging tools.

NetSight is creating *postcards* from the packet histories. A postcard is a collection of the necessary data from a packet to create a packet history. This means that each switch will create a postcard when a packet arrives. The postcard consists of only the information needed, making the size as small as possible before forwarding it to the NetSight server. As well as including the necessary information about the packet in the postcard, it will also include the state version of the switch at the time. The purpose of this is to be able to replay a network event, not only with the packets, but also with the switch states. This ensures network events can be reconstructed and analysed.

The researchers behind the NetSight API created four applications to show the power of packet histories. The applications created are:

- **ndb:** Interactive Network Debugger
- **netwatch:** Live Invariant Monitor

- **netshark:** Network-Wide Path-Aware Logger
- **nprof:** Hierarchial Network Profiler

Each of these applications are utilising the packet histories generated by NetSight, and show the versatility in the API. The API exposed by NetSight is named PHF (packet history filter) and it is very similar to regular expressions on packet histories.

Using the previously mentioned applications, network administrators will be able to debug the network, get alerted when specific behaviour occurs, capture a complete path of a packet, and profile the network.

The researchers highlighted some issues with the method of using packet histories in your network. The postcard generation for each packet will lead to an added strain on the network, as well as on each switch. The postcards are compressed before moved onto the NetSight server, but it will still be putting more data on the links. The central server storing all postcard data is also an issue. How much data should be stored? This would vary from implementation to implementation as policies on data retention varies.

2.6.7 Anteater

Anteater[25] is a tool for debugging problems in the data plane. The tool analyses the data plane state of network devices. This is done by creating an overview of the network topology and the devices' forwarding tables. With this overview Anteater represents them as boolean functions and checks against invariants defined by the operator. The invariants could be loop-free forwarding, connectivity, and consistency as examples.

After having translated the network design and information to boolean functions, the comparison with the invariants are done through a SAT (satisfiability problem) solver. If the SAT solver finds a problem it will highlight the problem, as well as include an example that will not trigger the issue. This way the operator can more easily diagnose the network issue.

The authors tested the performance of Anteater on a network consisting of 384 routers and they found that check three invariants (forwarding loops, packet loss, consistency) took almost 30 minutes. The main portion of this time was spent on the consistency part of the check.

Anteater is able to check and find invariants in the network, and this can help the network operators in localising faults, as well as providing a functioning example to help with the troubleshooting part.

2.6.8 RuleScope

RuleScope[7] is a tool designed to accurately detect forwarding inconsistencies. This is done by checking the rules, not just the presence, but also the priority.

There are two different procedures for RuleScope, the detection and troubleshooting algorithms. The detection algorithm uses probing to

uncover a fault in forwarding, while the troubleshooting algorithm uncovers the actual flow table of the switch.

The architecture of RuleScope utilises NetSight's postcard method for rule installation and probing. Probe packets are generated based on the rules installed to test the forwarding rules.

The troubleshooting algorithm is developed into two different algorithms; online and semi-online troubleshooting algorithm. The online troubleshooting algorithm adapts its checking based on the previous probing done. The semi-online troubleshooting algorithm attempts to increase the efficiency of the troubleshooting by issuing the probe packets in batches.

The authors have currently implemented the prototype of RuleScope with the Ryu SDN controller[33] and Pica8 P-3297 switch.

2.6.9 Libra

Libra[39] is a troubleshooting tool to verify forwarding tables in large, switched networks. It is able to do this verification in a fast and accurate manner by using MapReduce[13].

MapReduce is used in the verification process. Libra assumes packet forwarding based on longest prefix matching to increase scalability.

To be able to run the verification process, Libra needs an overview of the network. This is done by creating snapshots of the network. The network control messages, routing messages, is gathered by Libra to create an overview of the network. The snapshot is taken in moments where the network is stable, leading to the name *stable snapshot*. If the network is stable for a moment defined by the operators, the network is stable and a snapshot can be taken. After creating the snapshot, the process for checking the network for correctness begins.

The snapshot is divided between multiple servers to be able to process the network and verify its correctness in a timely manner. Libra is then using MapReduce on the snapshot to create a forwarding graph to be used in the checks. As time goes by more snapshots are taken, but Libra is able to use incremental updates to avoid having to process everything every time.

Libra is interested in verifying a few important properties of the network: reachability, loop-freeness, black holes, and waypoint routing. With 50 machines running Libra and checking a network consisting of 11,260 (DCN) switches the jobs is finished in 57 seconds. The speed of the tool is connected to its narrow scope. It does a narrow piece of verification very good, but it has limitations to how much else that can be done. Some of the limitations are related to the way Libra slices the network, as well as a issue when dealing with NAT and other non-deterministic behaviour.

2.6.10 Linear-time verification of firewalls

Firewalls are essential in today's networking world. They are one part of the layered security perimeter. With the increase in bandwidth consumption firewalls are constantly being challenged to improve performance and

throughput. The paper from Acharya and Gouda[1] is attempting to improve the performance of the policy verification in firewalls.

A firewall consists of a number of rules where each rule is bundled with a accept/deny verdict. Within each rule there are a number of fields that need to match for the rule verdict to kick in. The common fields are source address, destination address, and port. With single entities in each field the firewall performs as many checks as there are rules making the complexity $O(n)$. However, firewalls today does not consist of these simple rules with single entities in each field. Rules include a range of source addresses, a range of destination addresses, and a port range. Converting these type of rules to single entities rules would give a complexity of $O(n^d)$ where n is the number of rules in a firewall and d is the number of fields. The proposed algorithm from Acharya and Gouda gives a time complexity of $O(nd)$.

This is achieved by having two passes over the firewall, a deterministic pass, and a probabilistic pass. If the deterministic pass does not produce a conclusion the probabilistic pass is run.

The input is firewall F and a property P.

The deterministic pass produces one of these outcomes:

1. F satisfies P
2. F does not satisfy P
3. No conclusion can be reached

The deterministic pass is guaranteed to produce outcome 1 or 2 if the firewall adheres to the following properties:

1. The rule includes a singleton property (only single entries)
2. The firewall is a two-phase firewall
3. The firewall is conflict-free (no shadowed rules)

Outcome 3 results in the need of a probabilistic pass as well. The deterministic pass will produce outcome 1 and 2 in more than 99% of the tests.

To increase the probabilistic pass the authors have used the concept of corner packets. Each rule with fields consisting of ranges, either IP addresses or ports, can be divided into multiple singleton rules. Singleton rules are rules with a single entity in each field. By identifying the start and end of each range the properties to be tested can be reduced considerably. To increase the probability of a match, the number of "corners" can be increased to not only use the start and end values. The authors refer to this as the K value. For high accuracy the authors have suggested a K value of 1024.

As the probabilistic pass is so much slower than the deterministic pass the goal is to keep the number of outcome 3 to as few as possible. From the probabilistic pass there are two outcomes:

1. F satisfies P with high probability
2. F does not satisfy P

Part II

The project

Chapter 3

Planning the project

3.1 Objectives

The main objective is to make sure the path defined in the flow entries is followed by the traffic. As a second objective, the verification of the rules and paths should be made as efficient as possible.

3.2 Design

To complete the objectives listed in the previous section, three parts have been identified as necessary in the project software:

- Path prediction
- Path verification
- Test generation

The path prediction is used to read the flow entries from the switches and calculate the path to be used for a specific packet. The prediction part is crucial as it is the "truth" as the switch sees it.

Path verification is the process of tracing a real packet through the network and comparing it with the prediction for the same packet.

Test generation is essential to the project. This involves generating packets based on the flow entries available, and using the path prediction and verification to compare results.

3.3 Implementation

3.3.1 Environment

The project is utilising a virtual environment based on the Mininet programme. With Mininet a complex topology and virtual network is simple to setup on a single machine. Nodes can be spun up in a simple manner and interacted with through an API, or through the command line interface.

The installation of Mininet is done on a virtual machine running on a Hyper-V hypervisor. The operating system is Ubuntu 16.10. Mininet is running version 2.

While the nodes with Mininet are running on a single virtual machine, the controller software is running on a different virtual machine. This is done for convenience and to separate the functions. The controller software is run on a machine with operating system Windows Server 2012.

3.3.2 Topology

The topology used in the project is based on the Stanford backbone network. This topology has been extracted and imported to Mininet by the ATPG project. The project is modelled after the real life topology of the Stanford network. The topology has been adapted from a "traditional" network based on Cisco routers. The rules have been adapted to OpenFlow entries. The topology from the ATPG project was created for an earlier Mininet version and has been updated to run under Mininet version 2.

The topology includes 16 switches and close to 4000 different flow entries.

Creating the topology is done by starting the Python programme mininet_builder.py imported from the ATPG project. As Mininet manipulates the Linux networking system root access to the operating system is required.

The topology as shown in figure 3.1 is built from a demo by the ATPG project. This demo was shown during their presentation of the project and is included in the project files. It is a Python programme which uses the topology files to build the topology in the controller.

```
sudo python mininet_builder.py -c 172.16.99.5 -p 6633
```

The programme is utilising the Mininet API for Python, creating the topology to replicate the Stanford topology. Each virtual switch is represented in the Linux networking system and can be interacted with either directly from the Mininet prompt, or by running commands in the Linux shell.

A list of the switches and its connections can be seen by issuing the command `ip -br link`:

```
s15-eth13@eth0    UP        f2:23:9f:76:2a:86
s15-eth14@eth0    UP        96:fd:2e:7e:60:0b
lo                UNKNOWN   00:00:00:00:00:00
s15-eth15@eth0    UP        b6:fa:52:7b:21:bd
eth0              UP        00:15:5d:00:50:21
s15-eth16@eth0    UP        be:52:ef:69:0e:9a
output truncated...
```

Each interface is listed on the hypervisor guest, but the end hosts are only accessible from within Mininet.

An example of interacting with the host h197 in Mininet:

```
mininet> h197 ifconfig
h197-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
```

Figure 3.1: The Stanford topology as pictured by the ATPG project

```
        inet 10.0.0.181  netmask 255.0.0.0  broadcast
    10.255.255.255
        inet6 fe80::4c7:76ff:fe1c:35c9  prefixlen 64  scopeid 0x20
    <link>
        ether 06:c7:76:1c:35:c9  txqueuelen 1000  (Ethernet)
        RX packets 2165539  bytes 111739928 (111.7 MB)
        RX errors 0  dropped 2163647  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The hosts in the topology are connected to 16 different switches. Initially the hosts are created without a default route in their routing table. To have the hosts talk to the switches a default route is inserted to each host. This is done by executing a shell script in the Mininet environment.

```
mininet> h197 route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Iface
10.0.0.0        0.0.0.0         255.0.0.0       U     h197-eth0
```

Listing 3.1: Initial routing table on host

```
#!/bin/bash
# Add default route on all hosts

for i in {17..256}
    do
        echo "h$i route add default dev h$i-eth0"
    done
```

Listing 3.2: Script for inserting a default route on each end host

The script in listing 3.2 is used to create a text file for insertion into Mininet. The script creates a file named ipconf and is loaded into Mininet.

```
mininet> source ipconf
```

Listing 3.3: Loading a file into Mininet

```
mininet> h197 route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Iface
default         0.0.0.0         0.0.0.0         U     h197-eth0
10.0.0.0        0.0.0.0         255.0.0.0       U     h197-eth0
```

Listing 3.4: Routing table after default route insertion

3.3.3 Controller software

The controller software used is the Beacon OpenFlow controller developed by David Erickson at Stanford University. This specific controller is used in

the project because of the already close connection with the ATPG project. However, the controller usage is not very complex and using a different controller in Beacon's place should not be a problem.

Beacon is run on a Windows server using Eclipse. The controller software creates a simple web page with an overview of the network attached to the controller. Through the web page it is possible to check the counters of each switch, and the connected devices through a MAC address table.

To enable Beacon to insert OpenFlow rules into Mininet, the Beacon bundle Mahak is installed. This reads OpenFlow entries from a specific directory and inserts them into Mininet when Mininet contacts the controller. These OpenFlow entries have been generated in the ATPG project using the Stanford backbone network as the source.

```
rodvand@atpg$ python generate_stanford_ip_fwd_tf.py
=== Reading Cisco Router ARP Table File ===
=== DONE Reading Cisco Router ARP Table File ===
=== Reading Cisco Mac Address Table File ===
=== DONE Reading Cisco Mac Address Table File ===
=== Reading Cisco Router Config File ===
=== DONE Reading Cisco Router Config File ===
=== Reading Cisco Router Spanning Tree File ===
=== DONE Reading Cisco Router Spanning Tree File ===
=== Reading Cisco Router IP CEF File ===
=== DONE Reading Cisco Router IP CEF File ===
=== Compressing forwarding table ===
 * Originally has 1825 ip fwd entries *
 * After compression has 869 ip fwd entries *
=== DONE forwarding table compression ===
 * Generating IP forwarding transfer function... *
=== Successfully Generated Transfer function ===
=== Saving transfer function to file ../work/
    tf_simple_stanford_backbone//bbra_rtr.tf ===
=== Transfer function saved to file ../work/
    tf_simple_stanford_backbone//bbra_rtr.tf ===
```

Listing 3.5: Generating transfer functions for a router in the Stanford backbone

```
rodvand@atpg$ python generate_stanford_openflow_rules.py
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/bbra_rtr.tf ===
=== Transfer function loaded from file ../work/
    tf_simple_stanford_backbone/bbra_rtr.tf ===
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/bbrb_rtr.tf ===
=== Transfer function loaded from file ../work/
    tf_simple_stanford_backbone/bbrb_rtr.tf ===
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/boza_rtr.tf ===
=== Transfer function loaded from file ../work/
    tf_simple_stanford_backbone/boza_rtr.tf ===
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/bozb_rtr.tf ===
=== Transfer function loaded from file ../work/
    tf_simple_stanford_backbone/bozb_rtr.tf ===
```

```
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/coza_rtr.tf ===
=== Transfer function loaded from file ../work/
    tf_simple_stanford_backbone/coza_rtr.tf ===
=== Loading transfer function from file ../work/
    tf_simple_stanford_backbone/cozb_rtr.tf ===
... output truncated ...
```

Listing 3.6: Generating OpenFlow entries based on the transfer functions generated

After running the OpenFlow entries generation the folder stanford_openflow_rules holds all the rules ready for insertion into the controller, separated into files for each router. These files are then transferred to the host where the Beacon controller is located. When the controller software is started the switches connect to the controller and the OpenFlow entries are pushed to each switch helped by the Mahak bundle.

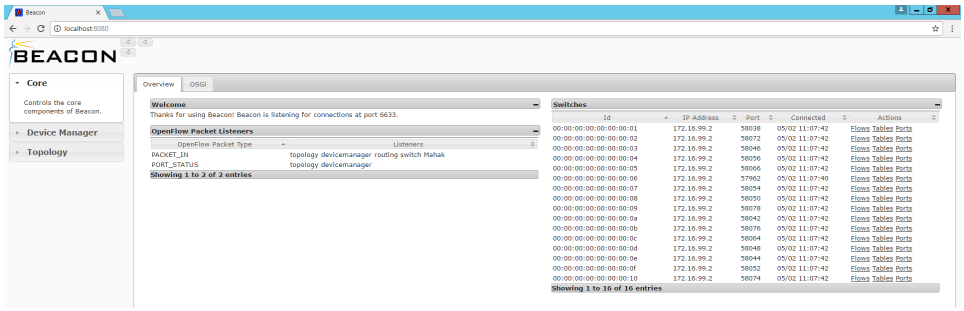

Figure 3.2: The web page presented by the Beacon controller

3.3.4 Programming

The tool developed in this project is based on the Python programming language[30]. Python is a programming language with a big community for user and library support. It is also an interpreted language and thus a fast language for development of a prototype.

Mininet has an API[27] written in Python, making it easy to generate switches and hosts programmatically.

For the topology setup and close connection to the ATPG project, the ATPG repository is forked and used as basis for this project.

3.3.5 Path prediction

Path prediction is an essential part of the software required to achieve the objective. The flow tables from the Open vSwitches are used to predict the path of a packet through the network.

Data from switches are extracted using the command line tool ovs-ofctl.

```
rodvand@atpg$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4): flags=[more]
```

```
3  cookie=0x0, duration=611591.961s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=60000,ip,nw_dst
     =128.12.0.2 actions=output:15
4  cookie=0x0, duration=611591.705s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59999,ip,nw_dst
     =128.12.0.17 actions=output:15
5  cookie=0x0, duration=611591.705s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59998,ip,nw_dst
     =171.64.0.17 actions=output:8
6  cookie=0x0, duration=611591.705s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59997,ip,nw_dst
     =171.64.0.18 actions=output:22
7  cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59996,ip,nw_dst
     =171.64.0.19 actions=output:8
8  cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59995,ip,nw_dst
     =171.64.0.24 actions=output:8
9  cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59994,ip,nw_dst
     =171.64.0.38 actions=output:4
10 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59993,ip,nw_dst
     =171.64.0.50 actions=output:8
11 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59992,ip,nw_dst
     =171.64.0.96 actions=output:18
12 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59991,ip,nw_dst
     =171.64.0.97 actions=output:12
13 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59990,ip,nw_dst
     =171.64.0.102 actions=output:4
14 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59989,ip,nw_dst
     =171.64.0.104 actions=output:8
15 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59988,ip,nw_dst
     =171.64.0.107 actions=output:14
16 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59987,ip,nw_dst
     =171.64.0.109 actions=output:14
17 cookie=0x0, duration=611591.704s, table=0, n_packets=0, n_bytes
     =0, idle_age=65534, hard_age=65534, priority=59986,ip,nw_dst
     =171.64.0.119 actions=output:4
18 ..output truncated..
```

Listing 3.7: Dumping flow entries from an Open vSwitch

The important fields for the path prediction are a priority, nw_dst, and actions. These three fields decide where the packet is sent. The flow table is ordered by priority making it easier to recognise the matching rule when

predicting the path.

```
Data: Host, IP address, port
Result: The predicted path
get device connected to host;
if device is switch then
    get flow entries for device;
end
add switch to path;
if flow entry contains IP address then
    if action equals drop or end host then
        add action to path;
        return path;
    end
    while action do
        run path prediction algorithm again;
    end
    return path;
end
```

Algorithm 1: Pseudo-code for predicting the path

The algorithm is simplified to illustrate the main idea behind it. The implemented algorithm includes adjustments for loop-detection. Loop-detection can be identified by checking the current switch in the path against the switches traversed previously.

The current implementation only records the switch a packet is set to flow through, not the egress or ingress ports.

The network topology is actively used in the code to be able to follow each flow through the network. Each flow entry has an action field associated with it. The most common action value is output:X where X is the port to forward the packet out. This data structure implements the network topology and is important for the ability to trace packets through the network.

In the code used to run path prediction, the topology has been loaded into the function to be able to see which switch or host is behind which port. This way the prediction can easily find the next hop in the chain by looking it up in its own data structure.

3.3.6 Path verification

The path verification complements the implementation and gives value to the solution by giving the software data to compare. The path prediction is responsible for returning a theoretical path through the network while the path verification returns the actual path taken by a packet through the network.

To perform the path verification the troubleshooting tool `tcpdump` is used. The data from all the switches are run through the tool and identified based on IP and port in the packet dump.

Tcpdump has multiple options for capturing multiple network inter-

faces. The option `-i <interface>` enables the capture of a single interface. The option can be paired with the interface `any` to capture all traffic on the host. However, due to some underlying issues in Linux the interface name is not available when using `any` as an option. To be able to identify the interface, and from the interface the switch packets are captured through, tcpdump has to be run with each interface as an option. To accomplish this a wrapper for tcpdump has been developed.

3.3.7 Test generation

The test packet generation is a function to verify the functionality of all OpenFlow flow entries. The generation of packets aims at hitting all the rules and comparing the predicted path with the path actually taken. This combines the path prediction and path verification functions. As a sub goal the packet generation is to be done as efficient as possible.

There are various measures done to avoid unnecessary packet generation and bandwidth consumption. As the tests are focusing on testing paths from host to host, the switches used in the test are called ingress switches. These switches are directly connected to end hosts and this reduces the number of switches and rules to be tested.

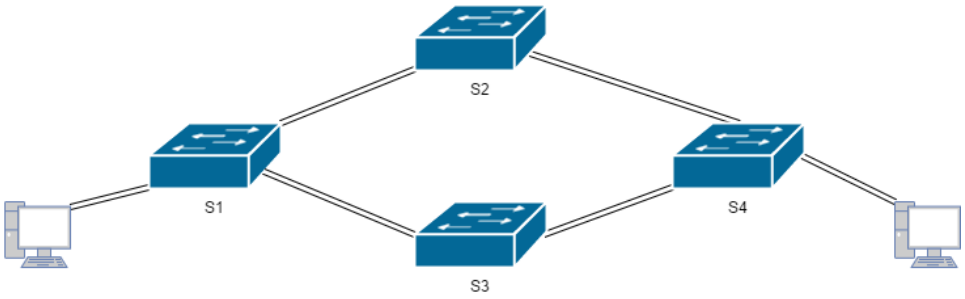

Figure 3.3: An example of ingress switches

Referring to figure 3.3 it is only needed to select S1 and S4 for packet generation from as they are the only switches connected directly to end hosts. S2 and S3 are purely transport switches and not connected directly to end hosts. For the objective of verifying the traffic path between end hosts this restriction does interfere with the task. While not a case in this project, various switches and network devices are also seen as clients in networks. In that case the assumption to only use ingress switches could be moot, and a path prediction and verification could be reasonable for all switches in the network.

The obvious method for packet generation is to convert each flow entry into test packets hitting the complete rule. This means covering all the fields defined in the flow entry completely. This method is called the full packet method throughout the project.

To improve the performance and efficiency of the test packet generation the theory from Acharya and Gouda's paper Linear-Time Verification of Firewalls[1] is adapted and used. This means the packets to test each rule

Destination	Priority	Action
128.12.0.2	60000	Output:15
128.12.0.17	59999	Output:15
171.64.2.0/26	59682	Output:3

Table 3.1: Flow entry in a table format

will be reduced from $O(n^d)$ to $O(nd)$ where n is the number of rules and d is the number of fields. With a limited number of fields the amount of tests can be done very efficiently. This method is called the corner packet method throughout the project.

```
cookie=0x0, duration=1100688.700s, table=0, n_packets=0, n_bytes
    =0, idle_age=65534, hard_age=65534, priority=59682,ip,nw_dst
    =171.64.2.0/26 actions=output:3
```

Listing 3.8: A single flow entry with an IP address range

To verify the forwarding functionality for the rule in listing 3.8 the flow entry is converted to two entries with the IP address field filled with the first and last address of the range. This is in contrast to converting the one flow entry into 64 singleton entries to be able to test all scenarios.

```
cookie=0x0, duration=1100688.700s, table=0, n_packets=0, n_bytes
    =0, idle_age=65534, hard_age=65534, priority=59682,ip,nw_dst
    =171.64.2.0/32 actions=output:3
cookie=0x0, duration=1100688.700s, table=0, n_packets=0, n_bytes
    =0, idle_age=65534, hard_age=65534, priority=59682,ip,nw_dst
    =171.64.2.63/32 actions=output:3
```

Listing 3.9: Converted into two flow entries

3.3.8 Sending the packets

The two different methods, full and corner, are used to create the necessary values for testing the network. The sending of these packets are done from Mininet using the Python library Scapy[34].

A separate Python programme named packet.py is created for sending packets.

To run any test from the hosts in Mininet the programme is required to be run from a host within, similar to the routing table adjustments done in section 3.3.2.

```
mininet> h197 python packet.py -h
usage: packet.py [-h] -d [DST] [-p {icmp,udp,tcp}] [-dp DPORT] [-
    sp SPORT]
                 [-t] [-v]

Craft a packet to test flow entries.

optional arguments:
  -h, --help          show this help message and exit
  -d [DST]            the destination address of the packet
  -p {icmp,udp,tcp}   the protocol used in the packet
  -dp DPORT           the destination port to be used in the packet
```

```
  -sp SPORT           the source port to be used in the packet
  -t                  show packet trace
  -v                  for verbose output. May be helpful under
    debugging
```

Listing 3.10: The Python programme packet.py

The packet generation is used to create a file for reading into Mininet. Listing 3.11 shows an example of the file contents.

```
h45 python packet.py -d 10.87.255.255
h45 python packet.py -d 10.84.0.0
h45 python packet.py -d 171.67.255.255
h45 python packet.py -d 171.64.0.0
h45 python packet.py -d 172.31.255.255
h45 python packet.py -d 172.28.0.0
h45 python packet.py -d 10.239.255.255
h45 python packet.py -d 10.232.0.0
h45 python packet.py -d 10.255.255.255
h45 python packet.py -d 10.240.0.0
h45 python packet.py -d 172.31.255.255
h45 python packet.py -d 172.16.0.0
h47 python packet.py -d 128.12.0.2
h47 python packet.py -d 128.12.0.17
h47 python packet.py -d 171.64.0.17
h47 python packet.py -d 171.64.0.18
h47 python packet.py -d 171.64.0.19
h47 python packet.py -d 171.64.0.24
h47 python packet.py -d 171.64.0.38
h47 python packet.py -d 171.64.0.50
h47 python packet.py -d 171.64.0.96
h47 python packet.py -d 171.64.0.97
```

Listing 3.11: Example file contents to be ran in Mininet

```
mininet> source send.source
```

Listing 3.12: Running the packet sending in Mininet

```
[Interface:s1-eth36:] 19:18:09.223696 IP 10.0.0.29.151 >
    128.12.0.2.1108: Flags [S], seq 0, win 8192, length 0
[Interface:s1-eth15:] 19:18:09.223996 IP 10.0.0.29.151 >
    128.12.0.2.1108: Flags [S], seq 0, win 8192, length 0
```

Listing 3.13: Tracing a single packet using dump.sh

3.4 Experiments

3.4.1 Packet generation

The packet generation experiment is divided into two parts. The first part involves creating test packets for all possible scenarios. The second part involves using the corner packet theory as outlined in the implementation section. This is to see the difference in time for packet generation.

Both experiments use the ingress switches only when generating packets. The ingress switches hold a total of 3840 flow entries. From these

flow entries a number of test packets is generated based on the method chosen in the following sections.

As opposed to a firewall these flow entries do not include a destination port field. This gives one less field to calculate and reduces the number of test packets to be generated.

When using the flow entries to generate packets and divide larger rules into smaller sets, some assumptions and decisions are made to reduce the scope. Special IP addresses[12] such as loopbacks and APIPA addresses have been excluded due to their special nature. These addresses are not commonly routed across networks. As they span a large IP network the generation of packets would become impossible to deal with. A default route (0.0.0.0/0) would lead to the generation of test packets for all IP addresses in existence and this is obviously a bad idea. These are individual adjustments that have to be tailored to the network tested.

3.4.2 Complete rule set

The generation of packets for the complete rule set involves dissecting each rule with ranges of IP addresses into singleton rules. This will create a number of packets from the rules to test with. The advantage with this approach is the thoroughness of the test. Each field within a rule will be iterated over and packets generated. This means all combinations of the rule will be tested and verified.

The calculation for all the flows in the flow table is: $O(n^d)$.

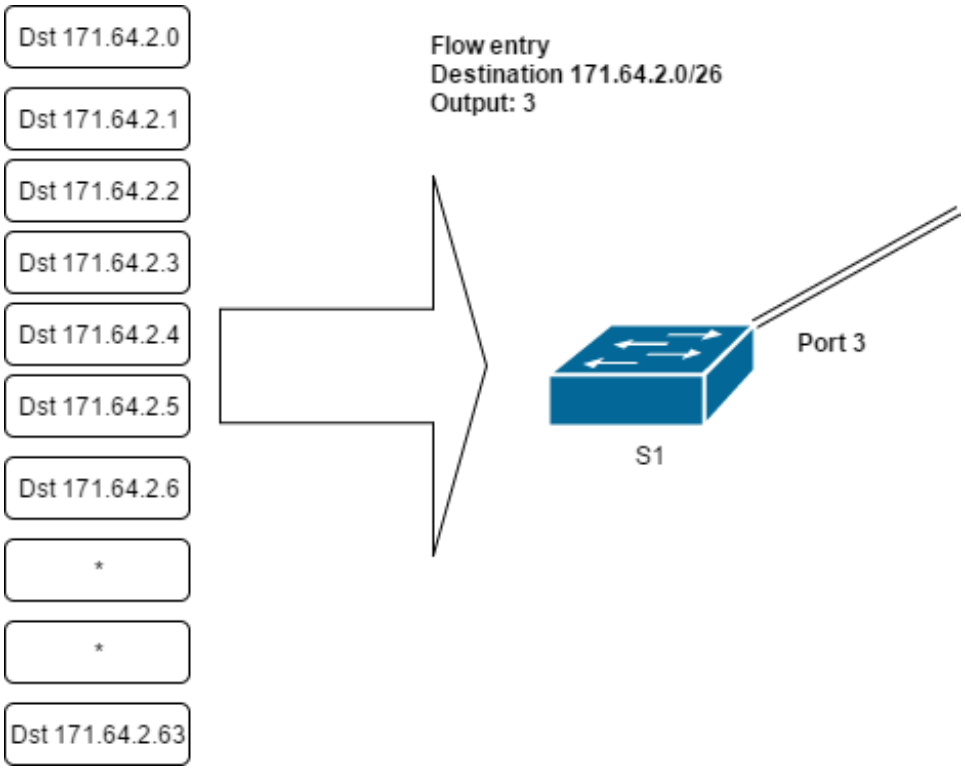

Figure 3.4: Testing with all packets

3.4.3 Corner packets

Rules with IP address ranges are divided into subsets of the range and packets generated. The theory is that if the start and end IP addresses behave in the same manner through the switch, the addresses in-between would get the same treatment.

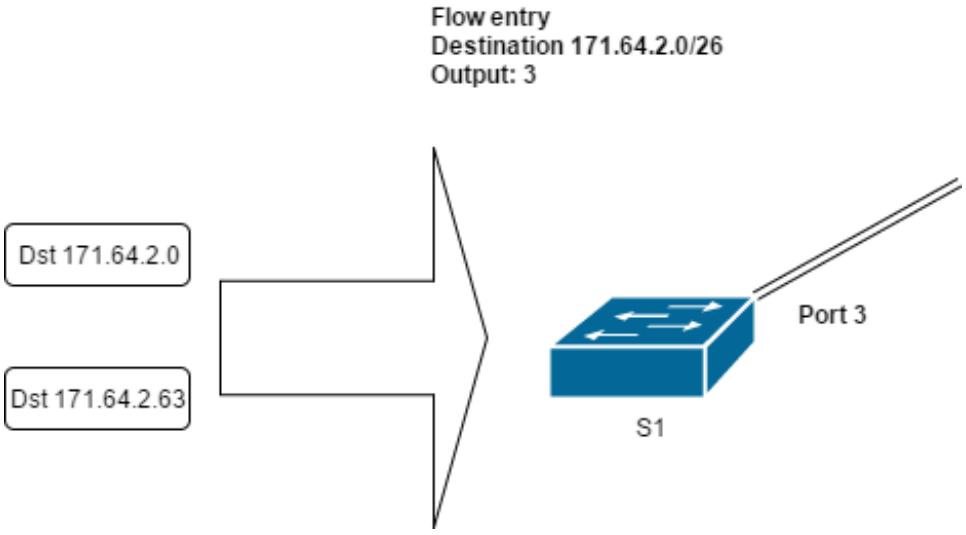

Figure 3.5: Testing with the corner packet theory

3.4.4 Introduction of faults

The objective is to make for a more stable and resilient network by enabling network administrators to run tests and verify the functionality. To simulate an environment in change, errors and faults are introduced to the network. The software needs to be able to recognise these faults, as well as locate and pinpoint where the error took place.

To be able to observe the errors the path prediction and path verification should be run separately. The time between path prediction and path verification may naturally be ran at different times due to the time it takes to predict and verify the path. To ensure that the results are consistent and easily generated the two functions are separated in the code. The run scheduling is up to each network administrator to decide.

Link failure

The most common error in a network is link failures. Throughout the networking world a link failure can have a massive impact on the network's stability. Ranging from a bad cable connected to an end user's desktop to undersea cables connecting different continents, link failures happen all the time. Being able to identify and pinpoint these failures are helpful for network administrators.

Destination	Priority	Action
~~128.12.0.2~~	~~60000~~	~~Output:15~~
128.12.0.0/30	59999	Output:15
171.64.2.0/26	59682	Output:3

Table 3.2: Flow entry where a rule is missing

Simulating link failures in Mininet is done by issuing commands on the Linux host manipulating the network interfaces.

```
1 rodvand@atpg$ sudo ip link show s9-eth14
2 1225: s9-eth14@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    qdisc noqueue master ovs-system state UP mode DEFAULT group
    default qlen 1000
3     link/ether d2:01:80:ee:1b:c3 brd ff:ff:ff:ff:ff:ff link-
    netnsid 141
4 rodvand@atpg$ sudo ip link set dev s9-eth14 down
5 rodvand@atpg$ sudo ip link show s9-eth14
6 1225: s9-eth14@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue
    master ovs-system state DOWN mode DEFAULT group default qlen
    1000
7     link/ether d2:01:80:ee:1b:c3 brd ff:ff:ff:ff:ff:ff link-
    netnsid 141
```

Listing 3.14: Shutting down port 14 on switch 9

Rule order

Another error to consider is the adjustment of the rule order in the flow table. Between path prediction and path verification the rule order may change and the tests should be able to detect this. Changes in rule order may come from incorrect input from the controller, or rules timing out. Each flow has a hard and idle timer associated with it adjusted by the controller. If the flow entry times out, the entry is removed from the switch and this could lead to a different action than previously.

Rule order is also affected by the priority of the flow entry. Each flow entry has a priority attached to it. Flow entries with a higher priority is located further up in the flow table on the switch. The controller could change the priorities of rules and as a result change the path taken by the packets.

The problem with finding errors when the rule order changes is if the rules have the same action. When this happens it is not possible to identify that the rule order has been changed and a fault in the network may still exist.

As illustrated in table 3.2 the second flow entry in the flow table has the same action as the first entry. In a flow table where rules shadow each other this could be an issue where a rule order change is missed by the troubleshooting software. If the flow table is checked and found to be conflict-free beforehand this should not show up as an issue and any change in rule order is recognised.

Part III

Conclusion

Chapter 4

Results

4.1 Implementing packet generation

4.1.1 Generating singleton rules based on OpenFlow entries

The first piece of code is getting the number of singleton entries required from the OpenFlow flow entry table. This is to compare the number of packets generated between the all-covering method and the corner packet method.

```
def slow_packets(flows):
    '''
    Return the number of packets to generate and what IP address
    to use when generating.
    Returns a tuple with (number of packet, IP addresses)
    '''
    import ipaddress
    count = 0
    for flow in flows:
        if 'nw_dst' in flow:
            if flow['nw_dst'] in BAD_NETWORKS:
                break
            network = ipaddress.ip_network(flow['nw_dst'])
            list_network = list(network)
            count = count + network.num_addresses
    return (count, list_network)
```

Listing 4.1: Python code to count the number of all-covering singleton packets

```
def corner_packets(flows):
    import ipaddress
    count = 0
    address = []
    for flow in flows:
        if 'nw_dst' in flow:
            if flow['nw_dst'] in BAD_NETWORKS:
                break
            network = ipaddress.ip_network(flow['nw_dst'])
            if network.prefixlen < 32:
                count = count + 2
                address.append(str(network.broadcast_address))
                address.append(str(network.network_address))
```

```
14              else:
15                  count = count + 1
16                  address.append(str(network.network_address))
17      return (count, address)
```

Listing 4.2: Python code to count the number of corner packets

After having created the number of packets to be generated with the two different algorithms, the path prediction and path verification can start.

4.2 Running the path prediction

The path prediction is run in Mininet using the 'flow-predictor.py' program.

```
1  rodvand@atpg:$ sudo python flow-predictor.py -h
2  usage: flow-predictor.py [-h] [-S SWI,SWI [SWI,SWI ...]] [-s SRC]
       [-d DST]
3                           [-p P]
4
5  Predict the packet flow through the network
6
7  optional arguments:
8    -h, --help            show this help message and exit
9    -S SWI,SWI [SWI,SWI ...]
10                         the switches to check the packet flow
       through
11   -s SRC                the source address
12   -d DST                the destination address
13   -p P                  run the prediction
```

Listing 4.3: Running flow-predictor with the help option

An example run to predict a packet path for a single IP address:

```
1  rodvand@atpg:$ sudo python flow-predictor.py -p -d 128.0.0.0 --
       host h197
2  Running prediction...
3  From h197 to 128.0.0.0
4  Predicted path: s1006 s1 s3 h240
```

Listing 4.4: Running flow-predictor to predict path of a single packet

The output of listing 4.4 is for one packet, and when doing the path prediction for multiple packets in preparation for verification with thousands of packets the output format is different and more optimised to programmatically be able to compare the prediction and verification part.

The reason a host is necessary for the prediction part is to simulate a data flow and an entry point in the network. When predicting packets for the whole network a start point is needed for when each flow table is tested. Testing a flow table from a host located two hops away is not realistic and may lead to wrong results.

As illustrated in 4.1 each switch has its own flow table. If all the tests are run from one host, the actions taken by each switch could be wrong and not represent the real network. It is therefore important to adjust the host from where the test is performed depending on which switch is tested.

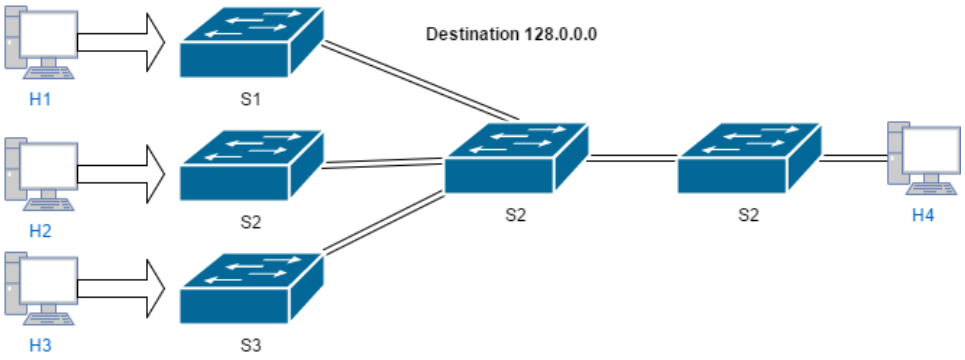

Figure 4.1: Different actions depending on the hosts tested from

```
rodvand@atpg:$ sudo python flow-predictor.py -P
Running prediction for whole network...
```

Listing 4.5: Running flow-predictor to predict path of all ingress switches

```
h45,s1,h31
h45,s1,h31
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1001,drop
h45,s1,s1001,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
h45,s1,s1007,drop
...output truncated...
```

Listing 4.6: File output from prediction

4.3 Running the test packet generation

Both of the tests are run at the same time to time and compare them.

```
rodvand@atpg:$ sudo python flow-predictor.py -T
Test packet generation
Number of all packets: 13152090
Number of corner packets: 6948
All packets: 45.63088274
Corner packets: 0.262818813324
```

Listing 4.7: Running flow-predictor with the -T option

The tests are ran in sequel and on stable conditions for the virtual machine where the test programme is running.

4.3.1 Packet generation for full packet set

Running the packet generation test for the full packet algorithm is listed in table 4.3.1. The first test is completed using the full network, all ingress

Full packet method			
Run	Flow entries	Number of packets	Time to generate
1	3840	13152090	42,5 seconds
2	3840	13152090	45,6 seconds
3	3840	13152090	45,0 seconds
4	3840	13152090	41,7 seconds
5	3840	13152090	42,4 seconds
6	3840	13152090	46,7 seconds
7	3840	13152090	43,7 seconds
8	3840	13152090	42,2 seconds
9	3840	13152090	41,9 seconds
10	3840	13152090	44,4 seconds

Table 4.1: Test packet generation with the full packet method

switches. A second test is ran calculating the run time for each switch with a different number of flow entries and test packets generated. In table 4.3.1 the distribution of flow entries and packets generated using the different methods are listed.

4.3.2 Packet generation for corner packet set

Running the packet generation test for the corner packet algorithm is listed in table 4.3.2.

4.4 Fault detection

An important part of the project is being able to detect faults in the network. The ability to detect and point the operator to the error is integral to the project.

The comparison between predicted path and actual path takes place after having generated test packets. The comparison is ran and discrepancies between the the two data sets are noted.

4.4.1 Link failure

A link failure is simulated by taking down the link from s1 to h29. The predicted path is h45 - s1 - h29. When taking down the link the path stops and should result in a h45 - s1 trace.

```
1 rodvand@atpg:$ sudo python flow-predictor.py -C
2 h45,s1 does not act like the prediction of h45,s1,h29
3 Number of errors detected: 1
4 Number of checks completed: 340
```

Listing 4.8: Running flow-predictor with the -C option

The comparison function compares the two files created during the prediction part and the tracing part. The result from each of these files are compared and any discrepancies are output for the network operator to

Time for packet generation per switch					
Switch	Flows	Full	Corner	Full time	Corner time
1	869	5565848	1646	19.33	0.10
2	843	5824710	1590	18.54	0.04
3	202	65135	361	0.31	0.009
4	174	65106	334	0.19	0.009
5	187	209034	288	0.56	0.007
6	124	208971	225	0.59	0.005
7	166	142446	298	0.45	0.007
8	145	142170	276	0.44	0.007
9	123	55464	218	0.18	0.005
10	108	55449	203	0.15	0.006
11	103	70140	182	0.20	0.008
12	90	70127	169	0.19	0.004
13	203	239211	319	0.68	0.008
14	141	239405	257	0.78	0.006
15	247	104993	371	0.32	0.01
16	115	93881	211	0.27	0.005
Times are averages from 10 runs.					

Table 4.2: Flow entries per switch

Packets per flow					
Switch	Flows	Full	Corner	Full ratio	Corner ratio
1	869	5565848	1646	6404	1.89
2	843	5824710	1590	6909	1.88
3	202	65135	361	322	1.78
4	174	65106	334	374	1.91
5	187	209034	288	1117	1.54
6	124	208971	225	1685	1.81
7	166	142446	298	858	1.79
8	145	142170	276	980	1.90
9	123	55464	218	450	1.77
10	108	55449	203	513	1.87
11	103	70140	182	680	1.76
12	90	70127	169	779	1.87
13	203	239211	319	1178	1.57
14	141	239405	257	1697	1.82
15	247	104993	371	425	1.50
16	115	93881	211	816	1.83

Table 4.3: Packets per flow

Corner packet method			
Run	Flow entries	Number of packets	Time to generate
1	3840	6948	0,2 seconds
2	3840	6948	0,2 seconds
3	3840	6948	0,2 seconds
4	3840	6948	0,2 seconds
5	3840	6948	0,2 seconds
6	3840	6948	0,2 seconds
7	3840	6948	0,2 seconds
8	3840	6948	0,2 seconds
9	3840	6948	0,2 seconds
10	3840	6948	0,2 seconds

Table 4.4: Test packet generation with the corner packet method

continue troubleshooting. A natural starting point for the operator would be the last hop where the discrepancy is detected on the trace.

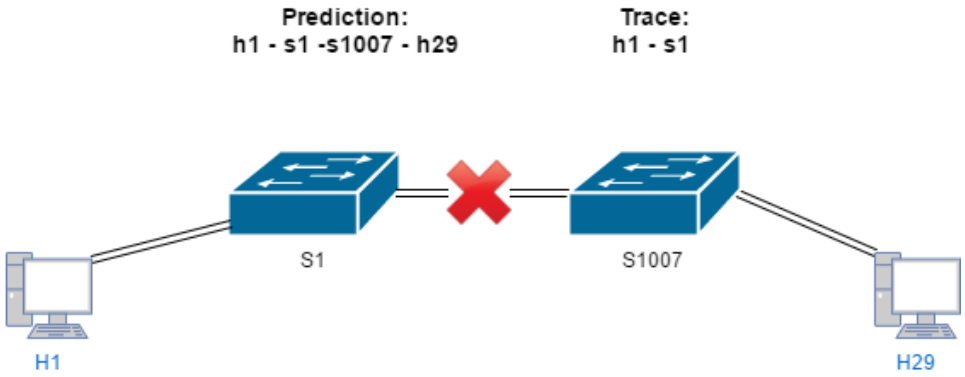

Figure 4.2: Link failure on the network

4.4.2 Rule order

A change in rule order can lead to a different action taken when traffic traverse the network. Such an action would be detected by the same algorithm detecting a link failure, but it is dependent on the action being different.

If the action taken forwards the packet the same way, there is no current way of recognising this rule order change. In such case a change in rule order could go undetected.

Chapter 5

Discussion

5.1 Algorithm comparisons and evaluation

The two different algorithms tested in the project have a different impact on the network. The full packet algorithm is slow, but very thorough whilst the corner packet algorithm is fast, but not as thorough. The question is: what are we trying to accomplish?

Proactive network troubleshooting is what every company aims for. Being able to adjust the network and anticipate failures are valuable contributions. Having a network troubleshooting tool that can give operators a 'heads up' before something happens is the holy grail of troubleshooting.

The tool developed in the project does seem to fit the part of a testing environment tool. The methods used are not directly applicable on a production network where equipment are separated over multiple different physical devices. This impacts the evaluation of the tool and the algorithms used.

The corner packet algorithm is a good and fast algorithm to use for packet generation. It makes it easy and fast to test the network policies. The drawback of this algorithm is the fact that it only use corner packets. When testing the network policies the most likely reason for discrepancy between the prediction and the verification is a software bug and link failures. And while link failures happen all the time and are a challenge, the prediction and verification is commonly run in succession, if not almost parallel. This means the prediction and verification algorithms must run almost constantly to be able to find a link failure when it happens. A more likely reason for discrepancies in prediction and verification is software bugs. These bugs are likely to spawn in very unlikely scenarios where edge cases can trigger faults in the code. The problem with using the corner packet method for a flow entry with a /26 destination is that the packets generated are only 2/64. If a packet in the middle of this subnet is triggering a code fault, the corner packet method would not capture it. The full packet method would be able to capture this when running the packet generation. The drawback would be the time taken for the packet generation as illustrated in the results section.

Combining these two methods could be a good solution. Running the full packet method less frequent than the corner packet method would make the full packet method locate any forwarding bugs whilst the corner packet method could recognise any link failures and other issues with discrepancies between prediction and verification. When evaluating the solution it is also important to note that the path prediction algorithm only needs to run when changes happen to the network. Such a notification could be requested from the controller.

5.1.1 Flow entries and number of packets

As seen from the results in table 4.3.1 the number of packets needed for the test increases with the number of flow entries. This is natural to a certain extent, but it is not necessarily the same increase in every scenario. In extreme cases the number of packets could relate on a 1:1 ratio to the number of flow entries. In such a case the full packet method and corner packet method would perform the same. To get such a scenario each flow entry would have to consist of a single destination IP address.

Because of the way the packet number is calculated from the flow entries it is impossible to correlate the number of flow entries to the number of packets needed by the full method and the corner method. Each network is special and has its own configuration with different flow entries.

From the breakdown of each switch in the network it is clear that the number of packets generated are mainly from s1 and s2. This is related to the number of flow entries present, but the ratio of packets/flows is lower on s1 than s2 even though the number of flows are larger on s1. Another example is between s8 and s14. The number of flow entries are close to the same with 145 on s8 and 141 on s14. As seen in 4.3.1 the packet ratio for the corner packet method is also very similar with 1.9 packets per flow for s8 and 1.8 packets per flow for s14. With the full packet method the ratio is different with s8 having 980 packets per flow and s14 having 1697 packets per flow. This shows the differences which can occur, all depending on the individual flow entries.

The corner packet method ratio is consistent around 1.7-1.9 with some outliers like s13 and s15 showing 1.5 packets per flow. A ratio close to 2 implies that the flow entries on the switch contains mostly destination IP address ranges and not single IP addresses. The ratio for the corner packet method can maximum be 2 in this case where the only corner packets selected are the start and end addresses. To increase the coverage the number of addresses selected with the corner packet method could be increased.

5.1.2 Packet tracing

The packet tracing functionality of the software developed is a simple wrapping script for tcpdump. A challenge with this is the to identify the correct packet when tracing it through the network. As this is a virtual network the amount of traffic is low and filtering traffic on queries enables

the tracing of packets. In a bigger environment the tracing feature would have to be tweaked to get a good result. Currently the filtering mechanism is done on the destination port being unique. In a bigger network this filter may not work as well and multiple fields might have to be combined to achieve an accurate result.

5.1.3 Fault detection

Faults are detected and presented through the software, but the comparison algorithm is very simple. The comparison runs through two files, the prediction and the tracing and presents differences between the two. In the case of an error being present late in the file, it will take a while to present this to the user. The planning and comparison method could use some optimisation for the error to be presented as fast as possible.

5.2 Environmental setup

Mininet and the tools needed to create the virtual network setup through this project are simple and powerful tools. The ability to create a large virtual network with a relatively small effort makes the creation of a mirroring network viable. In the project a virtual network of 16 switches and almost 4000 flow entries were created and run in an environment with modest specifications. The tests and performance of the setup shows that larger networks can be replicated without demanding massive amount of resources.

The resources used has not been a priority in the project, and it is certain that there are ways to improve the performance by adjusting the installation.

The controller software used in the project, Beacon, has not been updated or been in active development for some time. The reason for using this software lies mostly with a familiarity from the ATPG project and the setup from it. Transferring this project to a real world network the choice of controller has to be evaluated, as there are a number of actively developed controllers available today.

5.3 Challenges

The project has met multiple challenges throughout and it has been necessary to make adjustments underway. The initial ATPG project which much of this work is based on had an older code base published online. This urged updating various pieces of old code to fit with newer releases of supporting software.

Another issue when creating a software solution is the introduction of bugs and proofing the code. Making sure the code works as intended and making sure the results generated are valid, have also been challenging and time consuming work. A significant amount of time was spent debugging and making sure the code performs the tasks as intended.

In addition to these two issues, the project has encountered various revelations as to what is possible and impossible within a short time frame. A number of paths within the project turned out impossible to complete and had to be rejected. This included a wish to manipulate the Open vSwitch software to incorporate testing agents. This would have removed the need for testing agents on end hosts and made the tool better for real world scenarios. In this scenario the network devices would run their own testing software directly on the device.

Time management and devoting enough time and effort into the various parts of the project have been challenging. In hindsight the comparison method between prediction and verification could have received more attention. Being able to identify the rules used in the project would be a valuable contribution and allow for more detailed error detection.

5.4 Future work and improvements

Network troubleshooting is not something that will disappear from the business. Good and efficient tools will be a necessity when even more devices get connected to the network. With the boom of Internet of Things network administrators will have a tough time troubleshooting issues if the only tools available are ping and traceroute. The low level of entry for devices will put a bigger burden on the people administrating the network.

Troubleshooting tools should be built into the devices where network traffic traverse to enable thorough and accurate troubleshooting. While the tools built in this project are handy and valuable for its use cases, the implementation of this in a real-world scenario is impossible. The software fits for testing and evaluating a copy of a production network, but it does not work in a real-world scenario with the tools built and the methods currently used.

To improve on the work completed, a closer interaction with the controller software and network devices is needed. Using the controller to validate and compare flow entries, in combination with data directly from the network devices (switches), will increase the value of the tool. The current method of tracing and looking at data is not thorough and detailed enough to be implemented in real-world scenarios.

The current way of storing and comparing data from the prediction process and packet tracing is simple and not scaleable. The current file structure should be changed to use a database. This way the historical data could also be included without much effort, and the tool could be adjusted to list recent changes within the network. It would also make it easier to fetch statistics from changes and errors detected.

A way of locating and identifying exactly which rule is being tested would enable a more detailed error report during the detection phase.

Interaction with the network devices directly to get access to real-time network traffic should be pursued. This would enable an implementation of the tracing capability in the real-world. In this scenario the added bandwidth consumption is something to consider. This consideration

is the basis of the algorithm using corner packets. While the network consumption in a packet test in a virtual environment is no big deal, the bandwidth consumption in the real-world is a major concern and obstacle to a tool like this.

The software developed is currently very disjointed and requires a number of manual processes to complete one run. A more coherent and automatic process would make the tool more attractive for use in a network.

A separate and different topic would be the creation of a virtual network. To utilise the tool as a mirrored network of a production network there is a need to be able to automatically transfer a production network into a virtual network such as Mininet. Having an easy way of transferring a real-world production network into a virtual network ready for testing promotes the use of these mechanisms.

Chapter 6

Conclusion

The objectives of the project have been to confirm the data plane actions performed by the network devices in a network, and test these actions as efficiently as possible. To accomplish this, a virtual environment mirroring a production network has been setup for this purpose.

A virtual network has been established as a mirror of the Stanford backbone network. This network is used in different projects and provides a real world example of a network with numerous different switches and a flow table reflecting a real network. By using this network as an example, the results and data gathered in the project may be compared to other projects using the same example network.

The tool developed in this thesis is used to enable troubleshooting and packet tracing through a network created in Mininet. The software has been divided into three different parts:

- Path prediction
- Path verification
- Test generation

In order to have a basis for comparison the prediction part is important. This and the path verification ensures the first objective in the problem statement is completed. These two functions enable the verification of the data plane actions taken by the network devices.

The test generation covers the second part of the problem statement. Two different methods have been used for creating the test packets necessary to perform the network testing. The first method uses the complete IP range when testing a flow entry and creates a packet for every IP address. This is a very thorough test but also very time consuming. To find a more efficient approach, a corner packet method has been developed and implemented as an alternative testing method. This approach considerably reduces the number of tests needed, whilst still enabling error detection in the network. However, with the reduction of test packets, the potential to uncover software bugs and other issues decreases.

Using this newly developed troubleshooting tool will allow the operator to efficiently troubleshoot and identify issues within a software-defined network. The tool enables error detection and has been proven to identify link failure issues and changes to rule order, but is equally applicable to other issues where the data plane forwarding action does not adhere to the network policy.

References

[1] H. B. Acharya and M. G. Gouda. 'Linear-time verification of firewalls'. In: *Proceedings - International Conference on Network Protocols, ICNP* July (2009), pp. 133–140. ISSN: 10921648. DOI: 10.1109/ICNP.2009.5339691.

[2] Kanak Agarwal, John Carter and Colin Dixon. 'SDN traceroute : Tracing SDN Forwarding without Changing Network Behavior'. In: *HotSDN 2014* (2014), pp. 145–150. DOI: 10.1145/2620728.2620756.

[3] T Anderson. *Forwarding and Control Element Separation (ForCES) Framework*. Tech. rep. IETF, 2004, pp. 1–40.

[4] *Beacon Controller*. URL: https://openflow.stanford.edu/display/Beacon/Home.

[5] M Bjorklund. 'YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)'. In: *RFC6020* (2010), pp. 1–173. URL: https://tools.ietf.org/html/rfc6020.

[6] R. Braden. *Requirements for Internet Hosts - Communication Layers*. Tech. rep. October. IETF, 1989, pp. 1–116. URL: https://tools.ietf.org/html/rfc1122.

[7] Kai Bu et al. 'Is Every Flow on The Right Track ?: Inspect SDN Forwarding with RuleScope'. In: ().

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

[9] Marco Canini et al. 'A NICE Way to Test Openflow Applications'. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), p. 10.

[10] Martin Casado et al. 'Ethane: taking control of the enterprise'. In: *Sigcomm '07* (2007), pp. 1–12. ISSN: 01464833. DOI: http://doi.acm.org/10.1145/1282380.1282382. URL: http://doi.acm.org/10.1145/1282380.1282382.

[11] J Case and J Davin. *A Simple Network Management Protocol*. Tech. rep. MIT, 1990, pp. 1–36.

[12] M Cotton. *Special Use IPv4 Addresses*. Tech. rep. draft-iana-rfc3330bis-03. 2008, pp. 1–11. URL: https://tools.ietf.org/html/rfc5735%20http://www.ietf.org/internet-drafts/draft-iana-rfc3330bis-03.txt.

[13] J Dean and S Ghemawat. 'Simplified data processing on large clusters'. In: *Communications of the Acm* 51.1 (2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://dmkd.cs.wayne.edu/TUTORIAL/Bigdata/Papers/CACM08.pdf.

[14] R Enns. *Network Configuration Protocol (NETCONF)*. Tech. rep. IETF, 2011, pp. 1–113.

[15] *Floodlight OpenFlow Controller*. URL: http://www.projectfloodlight.org/floodlight/.

[16] Geoff Huston. *BGP report*. URL: https://bgp.potaroo.net/index-bgp.html.

[17] Natasha Gude et al. 'NOX: towards an operating system for networks'. In: *SIGCOMM Computer Communication Review* 38.3 (2008), pp. 105–110. ISSN: 01464833. DOI: 10.1145/1384609.1384625. URL: http://portal.acm.org/citation.cfm?id=1384609.1384625&coll=DL&dl=GUIDE&CFID=113040128&CFTOKEN=60814186%5Cnpapers2://publication/doi/10.1145/1384609.1384625.

[18] 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 2014)* (2014), pp. 71–85. URL: http://blogs.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol.

[19] Brandon Heller. 'OpenFlow Switch Specification 1.0.0'. In: *Current* (2009), pp. 1–36. ISSN: 09226389. DOI: 10.1002/2014GB005021.

[20] 'INTERNET PROTOCOL'. In: (1981). URL: https://tools.ietf.org/html/rfc791.

[21] Peyman Kazemian et al. 'Real Time Network Policy Checking Using Header Space Analysis'. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), pp. 99–112. URL: http://dl.acm.org/citation.cfm?id=2482626.2482638%5Cnhttp://yuba.stanford.edu/~peyman/docs/net_plumber-nsdi13.pdf.

[22] Ahmed Khurshid et al. 'Veriflow'. In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), p. 467. ISSN: 01464833. DOI: 10.1145/2377677.2377766. URL: http://dl.acm.org/citation.cfm?id=2377766.

[23] *Kytos SDN Platform*. URL: https://kytos.io/.

[24] Bob Lantz, Brandon Heller and Nick Mckeown. 'A Network in a Laptop : Rapid Prototyping for Software-Defined Networks'. In: (2010), pp. 1–6.

[25] Haohui Mai et al. 'Debugging the data plane with anteater'. In: *ACM SIGCOMM Computer Communication Review* 41 (2011), p. 290. ISSN: 01464833. DOI: 10.1145/2043164.2018470.

[26] Rashid Mijumbi et al. 'Network function virtualization: State-of-the-art and research challenges'. In: *IEEE Communications Surveys and Tutorials* 18.1 (2016), pp. 236–262. ISSN: 1553877X. DOI: 10.1109/COMST.2015.2477041.

[27] *Mininet Python API Reference*. URL: http://mininet.org/api/.

[28] Open Networking Foundation. 'OpenFlow Switch Specification 1.5.1'. In: *Current* (2015), pp. 1–36. ISSN: 09226389. DOI: 10.1002/2014GB005021.

[29] Open Networking Foundation. *Software-Defined Networking (SDN) Definition*. URL: https://www.opennetworking.org/sdn-resources/sdn-definition.

[30] *Python Programming Language*. URL: https://www.python.org/.

[31] Yakov Rekhter, Tony Li and Susan Hares. *A Border Gateway Protocol 4 (BGP-4)*. Tech. rep. IETF, 2006, pp. 1–04. URL: https://tools.ietf.org/html/rfc4271.

[32] *Ryu*. URL: https://osrg.github.io/ryu/.

[33] *Ryu SDN framework*. URL: https://osrg.github.io/ryu/.

[34] *Scapy*. URL: http://www.secdev.org/projects/scapy/.

[35] *tcpdump Man Page*. 2017. URL: http://www.tcpdump.org/manpages/tcpdump.1.html.

[36] *The POX controller*. URL: https://github.com/noxrepo/pox.

[37] Andreas Wundsam et al. 'OFRewind: Enabling Record and Replay Troubleshooting for Networks'. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (2011), p. 29.

[38] Hongyi Zeng et al. 'Automatic Test Packet Generation'. In: (), pp. 241–252. DOI: 10.1145/2413176.2413205.

[39] Hongyi Zeng et al. 'Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks'. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* Section 3 (2014), pp. 87–99. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng.

Appendices

Appendix A

Code

A.1 Device class

```
class Device(object):
    '''
    A class to represent a Device (host/switch) in the network
    '''

    def __init__(self, name):
        self.connections = {}
        self.name = name
        self.ingress = False

    def add_connection(self, port, device):
        connections = self.connections
        connections[port] = device
        self.check_ingress()

    def list_connections(self):
        connections = self.connections
        if len(connections) > 0:
            for p in connections:
                print(self.name + " is connected to " + p + " on
    port " + connections[p])

    def is_ingress(self):
        return self.ingress

    def check_ingress(self):
        conn = self.connections
        if len(conn) > 0:
            for p in conn:
                if "h" in p:
                    self.ingress = True
                    break

    def __eq__(self, other):
        return self.name == other.name

    def get_connection(self):
        '''
        If device only have one connection, return it
        '''
```

```
        conn = self.connections

        if len(conn) == 1:
            return conn

        return False
    def get_connected_device(self, port="0"):
        '''
        Return a link to a device on the port
        '''
        new_port = "eth" + port
        conn = self.connections
        if len(conn) > 0:
            for p in conn:
                if conn[p] == new_port:
                    return p
        return False

    def get_random_host(self):
        '''
        Return a host connected to the switch.
        '''
        if not self.is_host():
            conn = self.connections
            for p in conn:
                if "h" in p:
                    return p

    def is_host(self):
        return "h" in self.name
```

A.2 Packet creation script

```
#! /usr/bin/env python
# Script to craft a packet for testing flow entries

import sys
import argparse
import logging
import random
import datetime
import os
from scapy.all import *
from subprocess import call

# Commandline arguments
parser = argparse.ArgumentParser(description='Craft a packet to
    test flow entries.')
parser.add_argument('-d', nargs='?', required=True, help='the
    destination address of the packet', metavar='DST')
parser.add_argument('-p', nargs=1, choices=['icmp', 'udp', 'tcp'],
     default=['tcp'], help='the protocol used in the packet')
parser.add_argument('-dp', nargs=1, help='the destination port to
    be used in the packet', metavar='DPORT')
parser.add_argument('-sp', nargs=1, help='the source port to be
    used in the packet', metavar='SPORT')
parser.add_argument('-t', action='store_true', help='show packet
    trace')
```

```
20 parser.add_argument('-v', action='store_true', help='for verbose
       output. May be helpful under debugging')
21
22 args = parser.parse_args()
23
24 dest = args.d
25 prot = args.p[0]
26 trace = args.t
27
28 if args.dp:
29   port = int(args.dp[0])
30 else:
31   port = '-1'
32
33 if args.sp:
34   sourceport = int(args.sp[0])
35 else:
36   sourceport = random.randint(1, 65535)
37
38 # Logging
39 verbose = args.v
40
41 if verbose:
42   logging.basicConfig(level=logging.DEBUG)
43 else:
44   logging.basicConfig(level=logging.INFO)
45
46 # Start crafting a packet with SCAPY
47 logging.debug("Attempting to craft packet")
48 logging.debug("CMD arguments: " + str(args))
49
50 if prot == 'icmp':
51   logging.debug("We're in ICMP")
52   packet = IP(dst=dest)/ICMP()/"Hello World"
53
54 if prot == 'tcp':
55   logging.debug("We're in TCP")
56   logging.debug(port)
57   if port == '-1':
58     logging.warning("Port is not defined. Port is required for TCP
       .")
59     sys.exit(1)
60   packet = IP(dst=dest)/TCP(sport=sourceport,dport=port)
61
62 if prot == 'udp':
63   print
64
65 logging.debug("Going for a send")
66
67 # Start the trace with ./dump.sh
68 if trace:
69   # Create a time stamp for logging
70   timestamp = '{:%Y%m%d%H%M}'.format(datetime.now())
71   outfile = open("logs/" + timestamp, 'w')
72   logging.debug("Timestamp: " + timestamp)
73   logging.debug("Current directory " + os.getcwd())
74   logCode = call(["./dump.sh dst host " + dest + "> logs/" +
       timestamp + " &"], shell=True)
75   logging.debug("Call Error Code: " + str(logCode))
```

```
76
77   if logCode != 0:
78     logging.info("TCPdump returned with error code " + str(logCode
     ))
79
80 sr(packet, timeout=1)
```

A.3 Main programme

```
1  #!/bin/env python
2  '''
3  Script to go through the flow tables of
4  a number of switches, and predict the flow of a packet.
5
6  Takes for input:
7      - packet source and destination
8      - Optional: specific switches
9  '''
10
11 from functions import *
12
13 def main():
14     '''
15     Script to predict the flow of a packet through the switches.
16     Input:
17         switches
18         source
19         destination
20         port
21     '''
22     import argparse
23
24     # Commandline arguments
25     parser = argparse.ArgumentParser(description='Predict the
       packet flow through the network')
26     parser.add_argument('-S', nargs='+', help='the switches to
       check the packet flow through', metavar='SWI,SWI')
27     parser.add_argument('-s', nargs=1, help='the source address',
       metavar='SRC')
28     parser.add_argument('-d', nargs=1, help='the destination
       address', metavar='DST')
29     parser.add_argument('-p', action='store_true', help='run the
       prediction')
30     parser.add_argument('-P', action='store_true', help='run the
       prediction against all switches')
31     parser.add_argument('-T', action='store_true', help='run the
       packet generation')
32     parser.add_argument('-C', action='store_true', help='compare
       the prediction and tracing')
33
34     args = parser.parse_args()
35     source = args.s
36     dest = args.d
37     #print(args)
38     if args.S == None:
39         switches = get_interfaces(sort=True)
40     else:
41         switches = args.S
```

```
42      topology_database = {}
43      topo = create_topology(topology_database)
44      switches = get_ingress_switches(topo)
45
46      if args.p:
47          '''
48          We are predicting the path.
49          Get destination IP and run through algorithm
50          '''
51          if not args.P:
52              '''
53              Single packet prediction. Prettify the output
    accordingly.
54              '''
55              print('Predicting path...')
56              print('IP to predict path from: ' + str(dest[0]))
57              cond = (None, dest)
58              predict_path(topo, 'h197', cond)
59          else:
60              '''
61              Generation of prediction across all switches.
62              Output to file in "easy" to read format
63              for the path verification.
64              '''
65              fi = open('prediction.txt', 'w')
66              cond = (None, dest) # We only have a destination
    address
67              print('Running prediction for whole network...')
68              for switch in switches:
69                  host = switch.get_random_host()
70                  if host is not None:
71                      '''
72                      We have the host object - predict from it
73                      '''
74                      flows = dump_flows(switch.name) # Get the flow
     table from the switch
75
76                      (num_corner, list_corner) = corner_packets(
    flows)
77                      for adr in list_corner:
78                          path = []
79                          cond = (None, adr)
80                          pred = predict_path(topo, host, cond, path
    )
81                          pred_out = ','.join(pred)
82                          fi.write(pred_out + '\n')
83                  host = ""
84
85
86              fi.close()
87
88      if args.C:
89          '''
90          Compare prediction and tracing.
91          '''
92          pr = 'prediction.txt'
93          tr = 'tracing.txt'
94
95          compare(pr, tr)
```

```
96
97      '''
98      Do the test packet generation - with tracing
99      '''
100     if args.T:
101         import timeit # To calculate execution time
102
103         time_corner_tot = 0
104         time_full_tot = 0
105         corner_packets_count = 0
106         full_packets = 0
107
108         for switch in switches:
109             host = switch.get_random_host() # Host to test from
110             if host is not None:
111                 # Get the flows
112                 flows = dump_flows(switch.name)
113
114                 '''
115                 Create the corner packets
116                 '''
117                 time_corner = timeit.default_timer()
118                 (num, corner) = corner_packets(flows)
119                 create_sends(host, corner)
120                 corner_packets_count = corner_packets_count + num
121                 corner_elapsed = timeit.default_timer() -
    time_corner
122                 corner_ratio = float(num)/float(len(flows))
123                 print(switch.name + " & " + str(len(flows)) + " &
    " + str(num) + " & " + str(corner_elapsed))
124                 time_corner_tot = time_corner_tot + corner_elapsed
125
126                 '''
127                 Create the full set packets
128                 '''
129                 time_full = timeit.default_timer()
130                 (full_num, full_set) = slow_packets(flows)
131                 full_packets = full_packets + full_num
132                 full_elapsed = timeit.default_timer() - time_full
133                 full_ratio = full_num / len(flows)
134                 print(switch.name + " & " + str(len(flows)) + " &
    " + str(full_num) + " & " + str(full_elapsed))
135                 time_full_tot = time_full_tot + full_elapsed
136
137         print("Number of all packets: " + str(full_packets))
138         print("Number of corner packets: " + str(
    corner_packets_count))
139         print("All packets: " + str(time_full_tot))
140         print("Corner packets: " + str(time_corner_tot))
141         sys.exit(0)
142
143
144     #condition = (source, dest)
145     #path = predict_path(topo, 'h204', condition)
146     slow_count = 0
147     corner_count = 0
148     flow_count = 0
149     for switch in switches:
150         flow = dump_flows(switch.name)
```

```
151         flow_count = flow_count + len(flow)
152         #print(flow)
153         slow_num, slow_test = slow_packets(flow)
154         corner_num, corner_test = corner_packets(flow)
155         slow_count = slow_count + slow_num
156         corner_count = corner_count + corner_num
157     print(slow_count)
158     print(corner_count)
159     print(flow_count)
160
161 if __name__=="__main__":
162     main()
```

A.4 Helper functions

```
1  #!/bin/env python
2
3  from device import *
4  import sys
5
6  BAD_NETWORKS = ["0.0.0.0/8", "127.0.0.0/3", "169.254.0.0/16"]
7
8  def get_interfaces(sort=False):
9      '''
10     Get the interfaces on the host.
11     Parse them and return a dictionary with the switches.
12     '''
13     import netifaces
14     interfaces_old = netifaces.interfaces()
15     interfaces = []
16
17     for interface in interfaces_old:
18         if 'lo' in interface:
19             continue
20         elif 'ovs' in interface:
21             continue
22         elif 'eth' in interface:
23             continue
24
25         split_int = interface.split("-")
26         switch = split_int[0]
27
28         if switch not in interfaces:
29             interfaces.append(switch)
30     if sort:
31         return sorted(interfaces)
32
33     return interfaces
34
35 def dump_flows(switch):
36     '''
37     Dump the flow entries from one switch
38     and return a list with dictionary items.
39
40     command run: ovs-ofctl dump-flows <switch>
41     '''
42     import subprocess
43
```

```
44      command = "ovs-ofctl dump-flows " + switch
45      proc = subprocess.Popen(command, stdout=subprocess.PIPE, shell
        =True)
46      flows = proc.stdout.read()
47      flow = flows.split("\n")
48      entries = []
49      for line in flow:
50          if 'NXST' in line:
51              continue
52
53          entry = {}
54          att = line.split(" ")
55          entry['switch'] = switch
56
57          if len(att) > 1:
58              att.pop(0) # Remove first element (it's just an empty
        space)
59              el = att[-2].split(",")
60              att.pop(len(att) - 2)
61
62              for e in el:
63                  att.append(e)
64
65              for line in att:
66                  new = line.split("=")
67                  if len(new) == 2:
68                      entry[new[0].strip(",")] = new[1].strip(",")
69              entries.append(entry)
70
71      if len(entries) == 0:
72          return None
73
74      return entries
75
76  def match_rules(flows, match_conditions):
77      '''
78      Input: a dictionary of flows
79      Output: rules matched, first match listed first (list). If no
        rule matched, return None
80      '''
81      import ipaddress
82      if match_conditions[1]:
83          print(match_conditions[1])
84          dest = ipaddress.IPv4Address(unicode(match_conditions[1]))
85
86      if flows is None:
87          return None
88
89      matches = []
90
91      for flow in flows:
92          source_match = True
93          dest_match = True
94          if 'nw_src' in flow:
95              dest_match = ipaddress.ip_address(source) in ipaddress
        .ip_network(unicode(flow['nw_src']))
96          if 'nw_dst' in flow:
97              dest_match = ipaddress.ip_address(dest) in ipaddress.
        ip_network(unicode(flow['nw_dst']))
```

```
98
99          if source_match and dest_match:
100             matches.append(flow)
101
102     return matches
103
104 def print_flows(flows):
105     '''
106     Function to prettify the print of a flow.
107     Input: flows
108     Output: prints the flows in a readable manner
109     '''
110     for flow in flows:
111         if 'nw_dst' in flow and 'nw_src' in flow:
112             print("Switch: " + flow['switch']
113                     + " Priority: " + flow['priority']
114                     + " Source: " + flow['nw_src']
115                     + " Destination: " + flow['nw_dst']
116                     + " Action: " + flow['actions'])
117         elif 'nw_dst' in flow and 'nw_src' not in flow:
118             print("Switch: " + flow['switch']
119                     + " Priority: " + flow['priority']
120                     + " Destination: " + flow['nw_dst']
121                     + " Action: " + flow['actions'])
122         elif 'nw_src' in flow and 'nw_dst' not in flow:
123             print("Switch: " + flow['switch']
124                     + " Priority: " + flow['priority']
125                     + " Source: " + flow['nw_src']
126                     + " Action: " + flow['actions'])
127         else:
128             print("Switch: " + flow['switch']
129                     + " Priority: " + flow['priority']
130                     + " Action: " + flow['actions'])
131
132 def get_port(string):
133     local = string[0].split('-')
134     external = string[1].split('-')
135     device = external[0]
136     port = local[1]
137
138     return (device, port)
139
140 def get_device(dev, search):
141     '''
142     Return Device object after searching for device
143     '''
144
145     for device in dev:
146         if search in device.name:
147             print("Found dev: "+ search + " - " + device.name)
148             return device
149     return None
150
151 def create_topology(conn, f='topology.txt'):
152     '''
153     Definition to create a topology of the network.
154     '''
155     o = open(f, 'r')
156     topology = []
```

```
157
158     for line in o:
159         new_line = line.split()
160
161         # Loop through the list
162         count = 0
163         for entry in new_line:
164             if count == 0:
165                 device = Device(entry)
166
167             count = count + 1
168             sp = entry.split(':')
169
170             if len(sp) == 1:
171                 continue
172             elif not sp[1]:
173                 continue
174             port, ext = get_port(sp)
175
176             device.add_connection(port, ext)
177
178         topology.append(device)
179
180     return topology
181
182 def predict_path(topo, start, condition, path=[]):
183     '''
184     Predict the path between start and end by analysing the
185     flow tables on the switches and create a path.
186
187     Returns list with path
188     '''
189     device = get_device(topo, start)
190
191     if device.is_host():
192         if len(path) > 1:
193             return path
194         '''
195         If we are not at the end, get the switch where
196         the start host is connected.
197         '''
198         conn = device.get_connected_device()
199         '''
200         If the device already is in the path we have a loop
201         '''
202         path.append(device.name) # First device in path
203         path.append(conn)
204         return predict_path(topo, conn, condition, path)
205     else:
206         flow = dump_flows(device.name)
207         if flow is None:
208             path.append("drop")
209             return path
210
211         matches = match_rules(flow, condition)
212         actions = get_path(matches)
213         output = get_output(actions)
214         if output == "drop":
215             path.append("drop")
```

```
216             return path
217
218         # With multiple outputs
219         # WIP
220         for out in output:
221             next_dev = device.get_connected_device(out)
222             if next_dev in path:
223                 return path
224             path.append(next_dev)
225             return predict_path(topo, next_dev, condition, path)
226
227     return path
228
229 def get_path(rules):
230     next_hop = []
231     if len(rules) > 1:
232         for rule in rules:
233             next_hop.append(rule['actions'])
234             return next_hop
235     else:
236         next_hop.append(rules['actions'])
237
238     return next_hop
239
240 def get_output(next_hop):
241     if next_hop == "drop":
242         return ["drop"]
243     hops = []
244     for hop in next_hop:
245         if hop == "drop":
246             return "drop"
247         line = hop.split(',')
248         for sp in line:
249             line2 = sp.split(':')
250             hops.append(line2[1])
251     return hops
252
253 def test_packets(topo):
254     '''
255     Generate test packets based on topology
256     '''
257     switches = get_ingress_switches(topo)
258     for switch in switches:
259         print(switch.name)
260
261 def get_ingress_switches(topo):
262     '''
263     Return a list with only the ingress switches
264     (switches which have hosts connected)
265     '''
266     ingress = []
267     for switch in topo:
268         conn = switch.connections
269         if switch.is_ingress():
270             ingress.append(switch)
271
272     return ingress
273
274 def slow_packets(flows):
```

```
275     '''
276     Return the number of packets to generate and what IP address
        to use when generating.
277     Returns a tuple with (number of packet, IP addresses)
278     '''
279     import ipaddress
280     count = 0
281     for flow in flows:
282         if 'nw_dst' in flow:
283             if flow['nw_dst'] in BAD_NETWORKS:
284                 break
285             network = ipaddress.ip_network(flow['nw_dst'])
286             list_network = list(network)
287             count = count + network.num_addresses
288     return (count, list_network)
289
290 def corner_packets(flows):
291     import ipaddress
292     count = 0
293     address = []
294     for flow in flows:
295         if 'nw_dst' in flow:
296             if flow['nw_dst'] in BAD_NETWORKS:
297                 break
298             network = ipaddress.ip_network(flow['nw_dst'])
299             if network.prefixlen < 32:
300                 count = count + 2
301                 address.append(str(network.broadcast_address))
302                 address.append(str(network.network_address))
303             else:
304                 count = count + 1
305                 address.append(str(network.network_address))
306     return (count, address)
307
308 def create_sends(host, adr):
309     '''
310     Create output to use in Mininet for sending of the test
        packets.
311     '''
312     fi = open('send.source', 'a+')
313     for a in adr:
314         fi.write(host + ' python packet.py -d ' + a + '\n')
315     fi.close()
316
317
318 def send_packet(dest):
319     '''
320     Function to craft a packet using scapy and
321     using the destination address send it through the network
322     '''
323     from scapy.all import *
324     import random
325
326     port = 80
327     sourceport = 1108
328     packet = IP(dst=dest)/TCP(sport=sourceport,dport=port)
329     sr(packet, timeout=1)
330
331 def send_packets(adr):
```

```
332     '''
333     Input: IP addresses
334     '''
335     for a in adr:
336         send_packet(a)
337 
338 def compare(pred, trace):
339     '''
340     Compare the prediction file with the trace.
341     '''
342     pr = open(pred, 'r')
343     tr = open(trace, 'r')
344 
345     pr_line = pr.next()
346     tr_line = tr.next()
347     count = 0
348     error_count = 0
349 
350     while pr_line and tr_line:
351         '''
352         Check if lines are the same.
353         If not we have a discrepancy
354         '''
355         count += 1
356         if pr_line != tr_line:
357             print(tr_line.strip() + " does not act like the
    prediction of " + pr_line)
358             error_count += 1
359         try:
360             pr_line = pr.next()
361             tr_line = tr.next()
362         except StopIteration:
363             # No more values in the files
364             break
365 
366     pr.close()
367     tr.close()
368     print("Number of errors detected: " + str(error_count))
369     print("Number of checks completed: " + str(count))
```

A.5 GitHub repository

https://github.com/rodvand/atpg

Appendix B

Data set

B.1 Data set for packet generation

Corner packet method												
Switch	Flows	Packets	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	869	1646	0.087	0.086	0.10	0.10	0.096	0.099	0.104	0.096	0.093	0.093
2	843	1590	0.041	0.041	0.04	0.039	0.041	0.052	0.043	0.040	0.040	0.045
3	202	361	0.009	0.009	0.009	0.009	0.009	0.009	0.009	0.011	0.009	0.009
4	174	334	0.008	0.008	0.009	0.011	0.009	0.008	0.008	0.008	0.034	0.010
5	187	288	0.007	0.013	0.009	0.010	0.008	0.008	0.007	0.007	0.013	0.009
6	124	225	0.005	0.007	0.005	0.005	0.005	0.008	0.005	0.006	0.006	0.005
7	166	298	0.008	0.016	0.008	0.007	0.011	0.008	0.007	0.007	0.008	0.007
8	145	276	0.006	0.011	0.007	0.009	0.008	0.009	0.008	0.007	0.007	0.007
9	123	218	0.005	0.008	0.005	0.005	0.012	0.008	0.005	0.006	0.005	0.005
10	108	203	0.005	0.009	0.005	0.006	0.010	0.011	0.005	0.005	0.005	0.005
11	103	182	0.004	0.008	0.005	0.097	0.036	0.009	0.005	0.004	0.004	0.005
12	90	169	0.004	0.011	0.004	0.005	0.019	0.008	0.006	0.006	0.006	0.004
13	203	319	0.015	0.012	0.009	0.017	0.020	0.020	0.012	0.016	0.009	0.011
14	141	257	0.015	0.009	0.006	0.010	0.014	0.009	0.009	0.012	0.006	0.006
15	247	371	0.014	0.022	0.011	0.010	0.014	0.011	0.028	0.011	0.011	0.012
16	115	211	0.007	0.013	0.008	0.005	0.007	0.005	0.005	0.005	0.005	0.005

Full packet method												
Switch	Flows	Packets	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1	869	5565848	18.51	17.86	18.53	18.65	19.36	18.47	18.33	18.37	19.07	18.34
2	843	5824710	18.48	18.00	18.27	18.60	18.67	18.99	18.68	18.81	18.36	19.46
3	202	65135	0.31	0.31	0.36	0.32	0.31	0.31	0.32	0.38	0.30	0.32
4	174	65106	0.19	0.28	0.19	0.20	0.19	0.20	0.19	0.18	0.28	0.21
5	187	209034	0.57	0.78	0.57	0.59	0.60	0.65	0.57	0.57	0.79	0.56
6	124	208971	0.60	1.15	0.60	0.62	0.64	0.59	0.60	0.59	0.62	0.67
7	166	142446	0.40	0.71	0.40	0.41	0.51	0.56	0.49	0.40	0.40	0.40
8	145	142170	0.43	0.64	0.41	0.43	0.79	0.59	0.45	0.41	0.41	0.43
9	123	55464	0.16	0.21	0.16	0.30	0.31	0.30	0.16	0.17	0.16	0.17
10	108	55449	0.14	0.23	0.14	0.23	0.28	0.31	0.16	0.14	0.13	0.14
11	103	70140	0.19	0.29	0.19	0.29	0.53	0.36	0.21	0.27	0.20	0.20
12	90	70127	0.25	0.26	0.18	0.28	0.49	0.37	0.27	0.26	0.18	0.23
13	203	239211	1.05	1.01	0.64	0.74	1.35	1.14	0.90	1.32	0.65	0.72
14	141	239405	1.18	1.07	0.66	0.78	1.09	0.90	0.81	0.95	0.66	0.87
15	247	104993	0.41	0.48	0.31	0.32	0.41	0.34	0.32	0.38	0.32	0.36
16	115	93881	0.33	0.55	0.31	0.38	0.33	0.30	0.34	0.29	0.28	0.27