An Algorithmic Approach to OpenFlow Ruleset Transformation

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Richard Sanger

2020

Abstract

In an ideal development cycle for an OpenFlow application, a developer designs a pipeline to suit their application's needs and installs rules to that pipeline. Their application will run on any OpenFlow switch, whether software or hardware based. A network operator deploying this application would assess their network's requirements and purchase OpenFlow hardware to meet these requirements; such as bandwidth, port density, and flow table size. In reality, this level of interoperability does not exist as many OpenFlow switches are built on a fixed-function pipeline. Fixed-function pipelines limit the matches and actions available to rules depending on the table, but in doing so make more efficient use of expensive hardware resources such as TCAM.

This thesis investigates improving OpenFlow device interoperability by developing a method to rewrite existing rulesets to new complex fixed-function pipelines. Additionally, this thesis developed the tools to assess and verify the interoperability and equivalence of OpenFlow rulesets and pipelines.

This thesis developed a library and tools for working with descriptions of fixed-function pipelines, specifically, the Table Type Pattern description. This library provides a method to check if an existing ruleset is compatible with a new pipeline. Additionally, this thesis designed and implemented a pragmatic approach to compare if the forwarding behaviour of two OpenFlow 1.3 rulesets is equivalent. Equivalence checking provides a tool to verify that an OpenFlow application rewritten to program a new pipeline maintains the correct forwarding behaviour.

Finally, this thesis investigates the problem of algorithmically rewriting an existing OpenFlow ruleset, programmed by an existing application, to fit a different fixed-function pipeline. Solving this problem allows an OpenFlow application to be written once and run on any OpenFlow switch. This research aimed to solve this problem in a comprehensive manner that did not rely on the target pipeline supporting features such as OpenFlow metadata. This thesis developed and implemented a general method to convert an OpenFlow 1.3 to a complex constrained fixed-function.

Acknowledgements

First, I would like to thank my supervisors Richard Nelson, Matthew Luckie, and Bill Rogers, for guiding me through my PhD. Their guidance was invaluable, they always made time to help me with my research, and they helped me focus my research towards an achievable goal. They encouraged me to publish my research and supported me through the process. Additionally, I would like to especially thank my chief supervisor, Richard, for organising university formalities and funding, which allowed me more time to focus on my research.

I would like to acknowledge and thank Matthew Luckie, Richard Nelson, and Brad Cowie for their effort and own time spent proof-reading this thesis. In addition, I am incredibly grateful to Matthew for his time and perseverance guiding me through the academic publication process and helping me to present the best possible research.

I would like to thank the members of the WAND research group for being a wonderful group to work with. I wish Florin, Chris, and Dimeji all the best with their PhDs.

To all my friends and family thank you for your support, you have all been great to spend time with over the years. Thanks for checking on my progress and reminding me to finish.

This work was funded, in part, by a University of Waikato doctoral scholarship.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ACL	Access Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
BGP	Border Gateway Protocol
CAM	Content-Addressable Memory
CLI	Command Line Interface
CNF	Conjunctive Normal Form
DAG	Directed Acyclic Graph
DPDK	Data Plane Development Kit
FAWG	Forwarding Abstractions Working Group
FDD	Firewall Decision Diagram
HAL	Hardware Abstraction Layer
ILP	Integer Linear Programming
MPLS	Multiprotocol Label Switching
MTBDD	Multi-Terminal Binary Decision Diagram

NDM	Negotiable Datapath Model
NOS	Network Operating System
ODL	OpenDaylight
OF-DPA	OpenFlow Data Plane Abstraction
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OSSDN	OpenSourceSDN
OvS	Open vSwitch
PBB	Provider Backbone Bridging
POF	Protocol Oblivious Forwarding
RAM	Random Access Memory
ROBDD	Reduced Ordered BDD
SAT	boolean satisfiability
SDN	Software-Defined Networking
SRAM	Static Random Access Memory
t-bit	ternary bit
TCAM	Ternary Content-Addressable Memory
TTP	Table Type Pattern
VLAN	Virtual LAN

Publications

R. Sanger, M. Luckie, and R. Nelson, “Towards transforming OpenFlow rulesets to fit fixed-function pipelines,” in *Proceedings of the 2020 ACM Symposium on SDN Research*, 2020, p. 123–134. [Online]. Available: https://doi.org/10.1145/3373360.3380844

——, “Identifying equivalent SDN forwarding behaviour,” in *Proceedings of the 2019 ACM Symposium on SDN Research*. ACM, 2019, pp. 127–139. [Online]. Available: https://doi.org/10.1145/3314148.3314347

R. Sanger, B. Cowie, M. Luckie, and R. Nelson, “Characterising the limits of the OpenFlow slow-path,” in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NFV-SDN.2018.8725766

Chapter 1

Introduction

1.1 Problem Statement

Software-Defined Networking (SDN) is touted to remove vendor-lock, allowing network operators to buy network hardware based on price, performance, and features rather than having to buy from the same vendor to maintain compatibility with existing network devices. The defining feature of SDN that enables this interoperability is the decoupling of the control-plane and data-plane. Responsibility of the control-plane is given to the network operator, while the device vendor retains the responsibility of the data-plane. The interface for controlling the data-plane is standardised in a Software-Defined network, allowing different vendors to provide interoperable network hardware which implements this interface. OpenFlow is a popular SDN standard, which exposes a programmable match-action pipeline to the OpenFlow application.

SDN standards, such as OpenFlow [1], provide the necessary flexibility for a network operator to program a network's behaviour. However, in practice, it is the device vendor's implementation of a standard that dictates what the network operator can achieve. Vendors implement standards in different ways, restricting a network operator to using only the features implemented by an SDN device. A primary source of deviation from SDN standards is unavoidable due to constraints present in underlying hardware packet-processing pipeline designs. As a result, SDN application developers often tailor applications to a single SDN device, and many applications will not work with another device. This thesis is an investigation into how the forwarding behaviour an existing SDN application installs to a device can be rewritten to support a different SDN device with a constrained pipeline. We refer to this as the rule-fitting problem. More specifically, this thesis investigates converting an existing OpenFlow ruleset to a new ruleset which is compatible with a constrained fixed-function OpenFlow 1.3 pipeline.

1.2 Contributions

This thesis shows that it is feasible to convert an OpenFlow ruleset to fit different fixed-function pipelines algorithmically. This approach does not rely on OpenFlow metadata and can fit rulesets to pipelines with complex requirements. This thesis includes three fundamental contributions:

- A series of tools and a library for working with Table Type Patterns (TTPs). A TTP is a machine-readable representation of the features supported by a pipeline.
- A pragmatic approach to testing ruleset equivalence, which accounts for multi-table pipelines, overlapping rule matches, and complex actions.
- An algorithmic approach to the rule-fitting problem, which can convert an existing OpenFlow 1.3 ruleset to a complex constrained pipeline.

In addition, for all of these contributions, we release our implementations to the SDN and research community [2]. While our implementation targets OpenFlow, the key principles behind our approach apply more generally to match-action style pipelines. We provide an overview of each contribution below in the context of the rule-fitting problem and their standalone merit.

1.2.1 Table Type Pattern Tools

One problem in fitting a ruleset to a new pipeline is representing the constraints of that new pipeline in machine-readable format. We opted to use TTPs for this purpose. TTP was standardised [3] by the Open Networking Foundation (ONF) Forwarding Abstractions Working Group (FAWG) to represent OpenFlow forwarding pipelines. A TTP is typically stored as JSON and provides flexibility by allowing the user to add custom extensions. While the standard existed prior to this research, we found tools to create and interpret TTPs were not available.

We created a library to load and verify the TTP structure, and find valid placements of OpenFlow rules. Additionally, we created tools such as a TTP validator to detect and suggest fixes for issues found. Such issues include missing or mismatched identifier reference names, incorrect types, and header field value or mask overflow. These tools will be helpful to both vendors wanting to create TTPs for their devices and programmers who want to load information from TTPs.

1.2.2 Ruleset Equivalence

Another significant hurdle we needed to overcome was verifying that a ruleset transformation maintained its original forwarding behaviour. Finding a suitable representation was difficult; this representation needed to resolve overlaps in priority-ordered OpenFlow matches and multi-table pipelines in an efficient manner. OpenFlow can match on over 1000 bits of a packet header, i.e. more than 2^{1000} unique packet header values. Equivalent forwarding of a ruleset requires checking each unique packet header for equivalent forwarding behaviour. We found TCAM match style representations, like OpenFlow uses, could not represent shadowed rules efficiently. The calculation for shadowed rules resulted in a massive rule expansion, quickly becoming intractable.

Instead, we found that Multi-Terminal BDDs (MTBDDs) [4] provide a suitable representation. This thesis presents algorithms for building a MTBDD that represents a ruleset's forwarding behaviour and is a canonical representation. Additionally, MTBDDs naturally support set operations, which we use to find the difference in forwarding between rulesets. The difference between two rulesets is a useful measure of where inconsistencies between rulesets exist and can be used to evaluate how correct a solution to the rule-fitting problem is. The MTBDD representation is useful for researchers and programmers alike to check that any rewritten ruleset retains its original forwarding behaviour.

1.2.3 The Rule-Fitting Problem

Finally, we present a general approach to solving the rule-fitting problem for OpenFlow 1.3 rulesets and pipelines. Solving the rule-fitting problem combines both our TTP and ruleset equivalence checking contributions. We use the TTP library to represent and find valid rule placements in the target hardware's pipeline. While we use ruleset equivalence to check the final ruleset is valid.

In this approach, we build on two primary rule transformations: merge and split. A split transformation splits an OpenFlow rule into two or more rules in the target pipeline, which spreads the rule's matches and actions across multiple tables. A merge transformation takes two rules installed in different tables and combines them into a single equivalent rule.

From these two basic operations, our technique computes possible placements of rules from the input ruleset and creates a partial boolean satisfiability problem to select viable combinations of placements. This boolean satisfiability problem does not include all constraints, as precomputing all constraints is expensive. Instead, the technique improves upon the previous result by adding restrictions based on the difference between the candidate ruleset's forwarding behaviour and the original. This approach is capable of fitting rulesets into complex pipelines, and does not rely on OpenFlow metadata to create paths. We found this approach is suitable for constrained pipelines; however, has scaling issues with unconstrained pipelines as these allow many possible placements.

1.3 Thesis Structure

This thesis starts, in Chapter 2, by introducing the background for our contributions. This background includes an overview of SDN and OpenFlow. It then introduces constrained fixed-function pipelines and describes underlying network hardware design principles which explain these constraints. The background continues by introducing methods of representing hardware pipelines, as required to express the pipeline in the rule-fitting problem. Finally, the background introduces and compares two methods of representing OpenFlow matches, a fundamental component required to check a ruleset's equivalence.

To represent the target pipeline in the rule-fitting problem, we needed a machine-readable representation of the pipeline's constraints. Chapter 3 describes how we designed and developed a TTP library to represent and work with pipeline representations. This chapter details the tools we created and the algorithms we developed to fit an existing rule into a new TTP.

Another fundamental requirement of the rule-fitting problem is checking if the resulting ruleset is equivalent. Chapter 4 describes the method we developed to check the equivalence of two OpenFlow rulesets. This equivalence checking method converts the ruleset to a canonical MTBDD representation. The chapter presents the algorithms to construct this MTBDD and provides an evaluation of the performance.

Thus far, Chapters 3 and 4 have introduced new tools and techniques which have applicability to other problems in their related area. These techniques are also the fundamental components required by the rule-fitting problem. Chapter 5 discusses an overview of the rule-fitting problem. Then provides the approach we took to designing a solution. Most importantly, Chapter 5 describes the high-level design overview of the rule-fitting solver we developed. This design splits the problem into two main stages: 1) creating transformations of each rule or path in the original ruleset, and 2) picking a valid combination of these transformations with the correct forwarding behaviour. Finally, Chapter 5 discusses related work to the rule-fitting problem and SDN interoperability more generally.

Chapter 6 details this first stage of the rule-fitting solver. The chapter provides methods of preprocessing the original ruleset to simplify the rule-fitting problem; including a description of ruleset compression, a preprocessing technique we developed to reduce the size of a ruleset drastically. Then, this chapter describes the transformations of a rule or path from the input ruleset that the rule-fitting solver builds in an attempt to place that rule or path in the target pipeline. Finally, the chapter outlines some transformations which remain as future work.

Chapter 7 details the second stage of the rule-fitting solver. In this stage, the rule-fitting solver attempts to find a valid combination of the transformations generated by the previous stage. For non-trivial rulesets, exploring all possible combinations of these transformations is an intractable problem. This chapter details how we used a boolean satisfiability solver to generate combinations of transformations. For each combination, the rule-fitting solver checks its equivalence against the original ruleset. This chapter describes the constraints we designed and expressed to the boolean satisfiability solver to constrain the combinations returned to search only where correct solutions are most likely.

Chapter 8 evaluates the rule-fitting solver we developed and focuses on evaluating the usefulness of the techniques introduced in Chapters 6 and 7. This chapter reports on the effectiveness of two preprocessing optimisations to the input ruleset: converting the ruleset to a single-table and compressing the ruleset. The evaluation also studies the effectiveness of the constraints we designed to filter combinations of transformations the boolean satisfiability solver returns. Finally, this chapter discusses the difficulties of fitting a real-world ruleset into a real-world pipeline and the limitations of our approach.

Finally, this thesis concludes in Chapter 9 and considers avenues for future work.

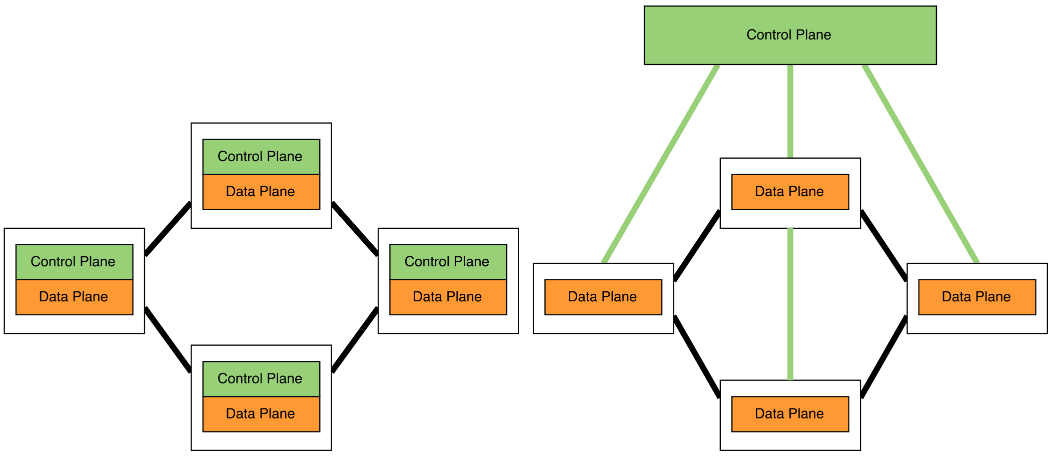
Chapter 2

Background

First, Section 2.1 presents the fundamentals of Software-Defined Networking (SDN) in comparison to traditional networking. Section 2.2 introduces the popular SDN standard OpenFlow and the factors guiding OpenFlow vendors' OpenFlow agent development. Section 2.3 highlights that fundamental hardware design issues that prevent a complete unrestricted implementation of an OpenFlow agent at the highest bandwidths. Due to considerations by vendors made when implementing OpenFlow agents, OpenFlow devices on the market have different capabilities and limitations. Section 2.4 discusses two methods of representing the capabilities and limitations of an OpenFlow pipeline.

Section 2.5 compares two match representations and provides the advantages and disadvantages of both. Both ruleset equivalence checking and translation need to compare and manipulate matches, (more generally sets of packets), and thus selecting an appropriate representation is critical. OpenFlow rules can partially or fully shadow a lower priority rule with overlapping matches. Essential operations such as representing the effective match need to be fast and compact, as to not exhaust memory, for both equivalence checking and ruleset translation.

Later, Chapter 5, presents a detailed background and related work specific to the rule-fitting problem.



(a) Traditional Network (b) Software-Defined Network

Figure 2.1: A comparison between a traditional and a Software-Defined network. In a traditional network, each network device runs its own instance of the control-plane. In the SDN model, the control-plane is separated from the network device, allowing the control-plane to run on a general purpose server and program the data-plane.

2.1 Software Defined Networking

SDN was coined in 2009 to describe the ideas around Stanford's OpenFlow [1] research; the definition has since broadened [5]. The defining feature of SDN is the separation of the network's control-plane from the data-plane. Figure 2.1 shows the difference between a traditional network and a Software-Defined network. In a traditional network, each network device runs its own control-plane which shares network information with all other network devices and then builds its view of the network. Traditional network devices independently make forwarding decisions based on their view of the network and independently program forwarding behaviour into their data-plane. Rather than running a decentralised control-plane across all network devices, Software-Defined networks often utilise a logically-centralised controller running on general purpose servers to centrally program the data-plane. In the SDN model, network devices no longer implement control-plane logic. Instead, they expose an interface that allows the controller to program the data-plane's forwarding behaviour.

The control-plane software running on traditional hardware is supplied and controlled by the hardware vendor. In SDN, the vendor still controls the software running on the network device, but it is no longer running control-plane logic, instead only providing an interface to program data-plane forwarding behaviour. SDN moves the control-plane logic to general purpose servers running SDN applications that program forwarding behaviour. Moving the control-plane off network hardware allows network operators to write programs to control their network [1].

The separation of the control-plane and data-plane offered by SDN promises the following advantages:

- Faster development - Traditional network development can be slow; it can take 5 to 10 years for a protocol to make it from development into deployments [6]. SDN moves the control-plane logic into software which a network operator can replace to suit their network's needs. Using software to control the network is also very useful for research and development, as a new protocol can be quickly prototyped and deployed on hardware. The standardisation gives developers the opportunity to write a single test suite for their network, which can test using both software switches for fast development, and hardware switches before deployment [7].

- Consistent management interface - In traditional networking, the vendor supplies the management interface. Typically these are Command Line Interface (CLI) interfaces that modify protocol settings. The commands vary between vendors, as does some of the default behaviour. This makes managing multiple devices from different vendors difficult. With SDN, the application provides the management interface independent of the hardware deployed, giving a network operator a centralised interface to make network changes that does not require manual configuration of individual network devices.

- Remove vendor lock - Traditional network devices work well with other devices from the same vendor. First, the consistent management interface provided by a single vendor is easier for a network operator. Second, vendors add extensions to protocols and may choose not to implement an extension resulting in interoperability issues. Ideally, with SDN, a single standard across all network devices is followed, which provides interoperability. The network operator selects and runs the controller software, which implements the required protocol support for their network.

- Fine-grained control - SDN allows fine-grained matching on arbitrary parts of a packet as required. Traditional networking is destination-based, whereas SDN is flow-based because a switch can match any field [6].

- More optimal network configurations - SDN allows a network operator to deploy a centralised controller which can make more optimal decisions based on its full network visibility. Additionally, as the controller is now running on standard servers, a compute cluster could be used to calculate optimal paths through the network now that there are no longer constraints on compute power imposed by each network device.

- Cost - SDN might become cheaper in the future because the network hardware will be simpler. Entities will no longer be paying their hardware vendor for the software license and stack to support all networking protocols they require. Instead, the hardware vendor will support a simpler SDN protocol allowing remote control of their hardware. More optimal network configuration could also result in better utilisation of links, reducing the number of devices needed for a deployment [8]. A simplified management interface may also reduce the time spent configuring a network.

A significant part of SDN is the "southbound" protocol; this is the protocol a controller uses to program a switch. A number of these protocols exist, including ForCES [9], OpenFlow [1], P4 [10] and Protocol Oblivious Forwarding (POF) [11]. A popular southbound protocol in use in 2019 is OpenFlow.

This protocol is used to connect the control-plane (controller) to the data-plane (forwarding elements). The server running SDN applications is known as the controller, and the network hardware are called forwarding elements or switches in the OpenFlow specification.

2.2 OpenFlow

OpenFlow [1] provides a standard protocol for programming SDN switches, using a match-action packet-processing model. OpenFlow primarily uses TCP as transport and defines a message structure to interact with the switch. These messages can add and delete rules, retrieve counters and even configure the pipeline of a flexible switch. Our research focuses on OpenFlow 1.3 [12], which 1) has good vendor and developer support due to new features added over prior versions that improve usability, and 2) supports multi-table pipelines.

In OpenFlow, a controller installs rules into a switch's flow tables to program the forwarding behaviour. Rules include three key components: matches, actions, and priority. OpenFlow matches support most traditional header-fields from Layer 2 Ethernet to Layer 4 TCP ports and allow arbitrary partial matching (aka masked matches) on most header-fields. The action applied to the packets can modify the packet, select the egress port, drop the packet, or send a packet to a new table for further processing. The priority of a rule determines which rule takes precedence when multiple rules match the same packets. For example, an OpenFlow switch may have both a rule matching IP:192.0.2.1 (a) and another matching IP:192.0.2.1, TCP_SRC:22 (b). If (a) had a higher priority than (b), (b) would see no traffic as (a) would match it all. If (a) had a lower priority than (b), then (b) would capture all traffic it matched and the remaining portion could be captured by (a).

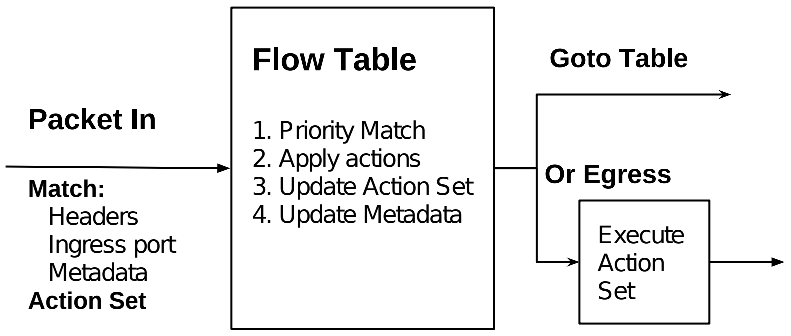


Figure 2.2: OpenFlow 1.3 pipeline packet processing through a table

2.2.1 OpenFlow Forwarding Pipeline

In an OpenFlow pipeline all packets begin processing in the first table. Figure 2.2 shows the processing applied to a packet through an OpenFlow table; multiple tables are chained together to form the full pipeline. First, an OpenFlow switch finds the highest priority rule that matches the packet and then applies the rule's actions. A rule can specify actions in two different ways: 1) *apply actions* a list which the switch executes immediately, thus allowing rules in the next table to match these modified fields and 2) *write actions* a list which the switch adds to the packet's *action set* and stores with the packet until the end of the pipeline. Then, if requested, the switch will update the packet's *metadata*. Finally, if the rule includes a *goto table* instruction, the switch sends the packet to that table. Otherwise, pipeline processing ends, and the switch applies the *action set*. When the switch adds to an *action set*, it replaces any existing actions of the same type. Optionally, a rule can include an instruction to clear the *action set* before updating it.

2.3 Diversity in OpenFlow Implementations

Unfortunately, OpenFlow implementations suffer all the common issues that standards have: differences allowed by optional or recommended features, ambiguity in interpretation, and undefined edge cases. However, to understand why we have diversity in OpenFlow switches, we first discuss the design considerations of both software and hardware OpenFlow switches.

2.3.1 Software Switch Design

Software OpenFlow switch implementations use standard general purpose servers to forward packets. The OpenFlow agent is a piece of code running on the CPU like every other application, and OpenFlow rules are stored in system memory in structures optimised for fast lookups. Because OpenFlow software switches use general purpose computing they present an entirely flexible OpenFlow implementation. Almost all features are available, and the number of rules and tables is limited only by available system memory which is inexpensive. Software switches employ many techniques to improve performance, avoiding the kernel networking stack by installing custom kernel modules or userspace networking libraries like Data Plane Development Kit (DPDK) and caches to improve lookup performance. For example, when the software switch Open vSwitch (OvS) sees a new network flow, it finds matching OpenFlow rules and determines a flattened representation which OvS installs into a fast hash lookup table [13]. Despite efforts in optimisation, software switches do not scale to the port density and high bandwidth that hardware switches do, and do not provide strong latency guarantees.

2.3.2 Hardware Switch Design

Hardware switches are designed to support high port densities and high bandwidth with a fixed low-latency, but lose flexibility in this process. To guarantee bandwidth and latency, hardware switches are organised in stages. A packet will take one clock cycle to be processed by each stage. One stage may perform a match based on the packet header while next may apply modifications to the packet. The bandwidth guarantee, in packets per second, is determined by clock speed, and the latency guarantee is determined by the number of stages. The most difficult part of this process is the match lookup because:

Masked lookups are expensive: To perform a masked lookup in a single cycle, a switch uses a special type of memory called Ternary Content-

Addressable Memory (TCAM). Unfortunately, the amount of TCAM a manufacturer can place on a chip is physically limited. TCAM uses many transistors and therefore space on silicon. TCAM also uses a lot of power as all TCAM entries must be powered in parallel to perform a lookup.

Exact matches are cheap: A switch can perform exact lookups in cheaper Content-Addressable Memory (CAM) or Random Access Memory (RAM) by using a data-structure such as a hash table.

Wide matches are expensive: Lookup tables are configured to match a number of bits (i.e. header fields), the more bits a table matches the fewer rules can fit in a lookup table without using more transistors. For example, an IPv4 address is a quarter the size of an IPv6 address. Therefore, a specialised TCAM would be able to fit 4 times more IPv4 matches in the same space a IPv6 match uses.

Lookup stages are limited Adding more lookup stages increases latency and uses more space on the silicon. This places a limit on the number of tables an OpenFlow controller can use.

OpenFlow places no limitations on the rules a controller can install; they can match any header field. To fully accommodate OpenFlow, a switch must support very wide lookups as the switch must be able to match every OpenFlow header field, and those lookups need to be installed in TCAM, as those header fields can be masked. OpenFlow requires wide TCAM matching, which is the worst case for hardware, and significantly limits the total number of rules. As a result, some OpenFlow switches only expose a single small table with full OpenFlow functionality. Others offer more rules by restricting matches in multiple large, but specialised tables, for example, a narrow exact-match on VLAN and Ethernet Address to perform Layer 2 forwarding.

The hardware supporting specialised tables may either be programmable or fixed-function. A fixed-function pipeline is designed to perform specific network functions and provides specialised tables that cannot be reconfigured. An example of a fixed-function pipeline is merchant silicon, the cheap off-the-shelf option many switches use. A programmable pipeline instead provides a pipeline that can be reconfigured. Pipeline reconfiguration is generally done at startup or upon a controller connecting, as it is usually not possible to reconfigure without interrupting forwarding. Programmable pipelines are still limited by a maximum number of lookup stages and the total amount of TCAM and RAM which is shared between all lookup stages.

2.3.3 Software vs. Hardware Switches

Software switches provide complete flexibility but do not scale to the high throughput, low-latency, and high port densities that hardware switches can provide. Software switches have nearly unlimited scalability of forwarding rules as they are stored in RAM which an operator can easily and cheaply upgrade. In contrast, hardware switches can provide low latency and high throughput guarantees coupled with higher port densities. However, lookups are made in a single clock cycle in TCAM for masked priority ordered lookups to achieve these guarantees. TCAM is expensive, both in silicon space and power consumption and is, therefore, a limited resource, placing a limit on the rule scalability of hardware switches. To more efficiently use TCAM, wide masked matches should be avoided where possible. All hardware switches have these limitations whether fixed-function or flexible.

This research focuses on fitting an existing ruleset into a new fixed-function pipeline, in which the silicon design predetermines table order and restricts the matches and actions available in each table.

2.3.4 OF-DPA: A Fixed-Function Pipeline

A motivating example of a fixed-function pipeline is Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) pipeline [14]. Broadcom built the OF-DPA abstraction on top of their proprietary SDK which exposes much of their

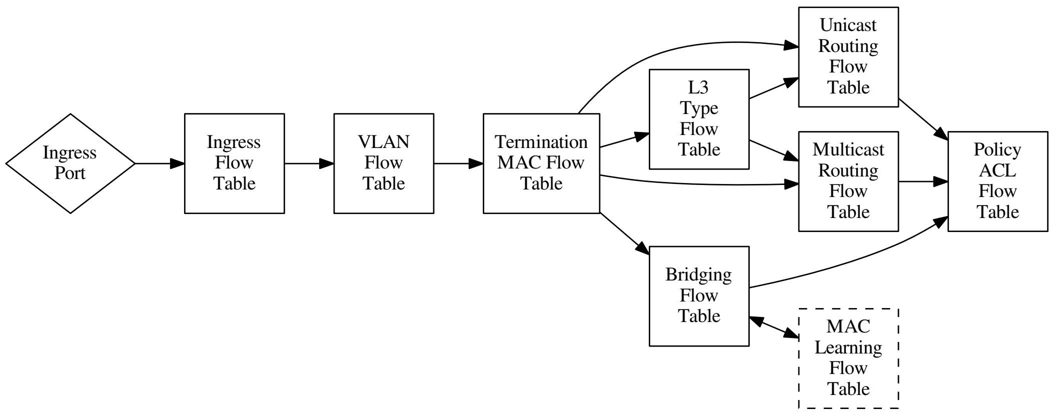


Figure 2.3: Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) 2.0 bridging and routing pipeline. OF-DPA 1.0 presents a similar view of these same tables.

StrataXGS series switching chip's underlying hardware capabilities and pipeline layout in a manner compatible with OpenFlow 1.3. The StrataXGS series switching chips are very popular merchant silicon and are used in switches by many vendors including Edge-Core, Quanta, Pica8, Dell, and HPE [15].

While the OF-DPA pipeline and code is open-source, unfortunately, building the code requires proprietary Broadcom SDKs which are only available to switch vendors under strict confidentiality agreements. Therefore, a switch vendor needs to supply OF-DPA firmware for their switch. Edge-Core and Quanta, among others, provide OF-DPA firmware for many of their switches. The OF-DPA pipeline is also generally representative capabilities of the underlying Broadcom StrataXGS chip which a switch vendor can expose themselves in OpenFlow. For example, Pica8's PicOS Network Operating System (NOS) exposes the routing and bridging tables which look just like those in OF-DPA.

There have been two major releases of OF-DPA: 1.0 and 2.0. OF-DPA 2.0 exposes more of the underlying chip's capabilities compared to OF-DPA 1.0. The OF-DPA 2.0 pipeline has a total of 33 OpenFlow tables. The tables exposed by the OF-DPA pipeline are specialised to their function, which allows for larger table sizes by limiting the matches and actions available. Not all tables in the OF-DPA pipeline relate to an underlying lookup table in silicon, and some tables exist to best express the underlying pipeline in the OpenFlow abstraction. For example, Broadcom has placed Layer 3 type tables before the unicast and multicast routing tables with built-in rules directing traffic based on whether the IP destination is a multicast address. However, no rules can be installed or modified in these tables.

The OF-DPA pipeline has specialised tables for Layer 2 switching and Layer 3 routing which allow for tens of thousands of rules. Figure 2.3 shows the tables in the OF-DPA pipeline used for basic bridging and routing. There are many other tables in the OF-DPA pipeline that support features such as MPLS, QoS, and egress processing, which are not shown in Figure 2.3. The size of the OF-DPA tables vary by the Broadcom chipset used and the memory configuration selected by the switch vendor. As a typical example illustrating the difference in table sizes, consider an EdgeCore AS5710-54X-EC: the Bridging table holds 160K entries, Unicast Routing holds 80K entries, Multicast Routing holds 72K entries, and Policy ACL holds 2K entries [16]. Many OpenFlow agent implementations only use the Policy ACL table, and are therefore limited to 2K rules, but, support nearly all match fields and actions. However, this means that much of the hardware's capability goes unutilised.

This thesis investigates algorithmically translating existing rulesets to fixed-function pipelines like OF-DPA to both improve switch interoperability of SDN applications and use the specialised tables to improve scalability. To better show the complexity involved in this process, we describe the rules an application needs to install to configure the OF-DPA pipeline for bridging and routing. To configure routing and bridging, an OpenFlow application must configure five key tables: VLAN, Termination MAC, Unicast Routing, Multicast Routing, Bridging, and Policy ACL.

The first table in the OF-DPA pipeline is the *Ingress Port Table* which allows an application to apply Quality of Service based on a packet's ingress port. On a table-miss, the *Ingress Port Table* sends a packet to the *VLAN Table* by default. An application uses the *VLAN Table* to add or modify a packet's VLAN tag. On a table-miss, the *VLAN Table* sends packets to the *Policy ACL Table* by default. An SDN application must install a rule for every VLAN it accepts by matching the ingress-port and VLAN-tag, and sending it to the *Termination MAC Table.* The OF-DPA pipeline does not allow untagged VLAN packets past the *VLAN table*; an application must tag all untagged packets with a VLAN in the *VLAN Table.* Rules in following tables such as the *Bridging* and *Policy ACL Tables* require an exact VLAN match. Next, the *Termination MAC Table* accepts rules that direct packets based on their Ethernet-destination, ingress-port, and VLAN-tag to either a *Routing Table* or the *Bridging Table.* A packet can either be routed or bridged, but not both; as such the pipeline can process both bridging and routing in parallel to reduce total pipeline latency.

The routes an application can install into the *Unicast Routing Table* must match an IP-destination prefix, and must write actions to decrement the packet's TTL and forward it to a Layer 3 interface group. A Layer 3 interface group rewrites Ethernet addresses and forwards the packet to its next-hop. The *Multicast Routing Table* only accepts rules that match an exact IP-destination, VLAN, and optionally IP-source, and must include actions to decrement the packet's TTL and forward the packet to a Layer 3 multicast group. A Layer 3 multicast group will duplicate a packet to a series of Layer 3 interface groups. An SDN application installing Layer 2 bridging rules into the *Bridging Table* must match an exact Ethernet-destination and VLAN, and must write the egress interface to the packet's action-set using a Layer 2 interface group. The OF-DPA pipeline can also be configured to mirror the *Bridging table* into a *MAC Learning table* that performs an Ethernet-source match sending unmatched packets, those without a known MAC address, to the controller to learn new port mappings. Mirroring tables in this way is an extension to the OpenFlow 1.3 specification.

In the OF-DPA pipeline, both the Routing and Bridging tables write actions to the packet's action-set rather than applying the action immediately. The actions in the packet's action-set will only be executed at the end of the

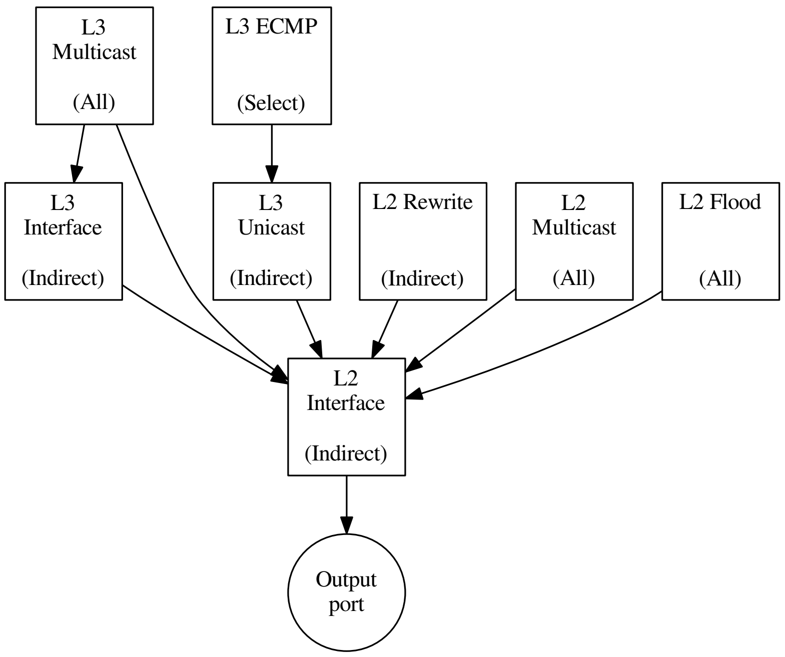


Figure 2.4: The OpenFlow group hierarchy in OF-DPA. An arrow from a group to another indicates that the group's buckets must contain that group as an action. All groups eventually include a Layer 2 Interface group which includes an output port action. Rules in the OF-DPA pipeline are required to include a group action and cannot output a packet directly except to the controller.

pipeline, and the actions can be modified or removed by a rule in any following table. In particular, a rule in the *Policy ACL table* can clear a packet's action-set, which would drop the packet. Additionally, OpenFlow output actions are not available directly to any rule in the OF-DPA pipeline (except to send a packet to the controller); instead, OF-DPA uses indirection via OpenFlow groups for all outputs to ports. Figure 2.4 shows OF-DPA's group hierarchy used in basic bridging and routing. A group's buckets must contain a group action that the arrow points to. The Layer 2 Interface group is the only group shown which contains the output port action. Different rules accept different group types as actions, which can be the Layer 2 Interface group directly. Groups such as Layer 3 and Layer 2 multicast have a group type of all, so will duplicate a packet to all buckets to multicast a packet. Some groups also include additional actions to rewrite a packet's fields. For example, the Layer 3 Interface indirect group rewrites a packet's source and destination Ethernet address and then outputs the packet to a port indirectly via a Layer 2 Interface group.

The final table in the pipeline is the *Policy ACL (Access Control List) table*, which is the most generalised table accepting rules which match almost any header field, and can apply almost any action or instruction. The *ACL table* is well suited for implementing a simple stateless firewall, as it can drop packets by clearing the actions set by the Routing or Bridging table. The *ACL table* can also modify the bridging or routing of any packet or direct the packet to the controller. The *ACL table* has almost complete support for all OpenFlow features and is often the only table used in many OpenFlow agent implementations because it supports the majority of the OpenFlow standard. However, the downside of this flexibility is that the number of rules supported is smaller compared to specialised tables. Compounding to the limited number of rules, programming a single-table pipeline typically requires more rules than a multi-table pipeline would require; in the worst case it is the Cartesian product of all tables. This worst case is common, simple Layer 2 switching in a multi-table pipeline uses a source learning table and destination forwarding table; each Ethernet address is installed once in both tables, requiring two rules for every address. Whereas, Layer 2 switching in an equivalent single-table pipeline requires a rule for each source and destination combination, therefore the number of rules required is the number of addresses squared.

The complexity of the OF-DPA pipeline makes it a particularly hard target to fit rules because matches and actions are constrained. An OpenFlow application needs to split components of the overall forwarding behaviour it requires throughout multiple tables in the pipeline. Additionally, the placement of bridging and routing before the *ACL table* means that a packet dropped by the pipeline often first requires the “wrong" routing or bridging actions before the *ACL table* clears them. Even resolving output actions to the correct groups is non-trivial. In Chapter 5, we introduce our method of algorithmically fitting rulesets into new pipeline’s like OF-DPA and address these complexities further.

2.4 Representing OpenFlow Pipelines

OpenFlow pipelines built on fixed-function silicon, such as OF-DPA as described in Section 2.3.4, have complex limitations on the rules which a controller can install. In order to successfully install transformed rules into a fixed-function pipeline, our algorithm requires a machine-readable method of representing the pipeline's requirements. To most optimally fit rules we not only need to be able to compute valid placements of rules, but we also need to consider the size of each flow table, so that placements do not exceed table size and can scale to large networks. This section discusses two existing pipeline representations: OpenFlow feature messages [12] and Table Type Patterns (TTPs) [3].

OpenFlow features request messages allow a controller to query a switch about the OpenFlow features it supports at run-time [12]. In OpenFlow 1.3 there are three types of messages: table features, meter features, and group features. A switch responds to an empty meter, group, or table features request message with a features response describing the supported number and types of groups, meters, or rules per table. Additionally, a controller can use the table features request to configure the pipeline on the switch; this is not possible with group and meter features request messages. OpenFlow features response messages are strictly structured and do not offer a fine-grained description. In particular, they cannot represent mutually exclusive features. Instead, OpenFlow features response messages represent the superset of all features available even though some combinations may be unavailable.

The Table Type Pattern (TTP) specification was designed to help solve interoperability issues between switches and applications by being an abstract OpenFlow pipeline description that both vendors and developers could develop against [3]. Unlike feature messages, a TTP is a textual representation of the pipeline which is shared offline and not at run-time via OpenFlow. A TTP can represent everything that the OpenFlow features message can, but also supports much more complex requirements including mutually exclusive features. TTPs support more fine-grained requirements including limits on the value or mask allowed in a rule's match. The major downside of TTPs is the lack of widespread adoption and use.

2.4.1 OpenFlow Feature Messages

OpenFlow includes the group features message since version 1.2, and meter and table features messages since version 1.3 [12]. Both group and meter queries are read-only requests, whereas a controller can configure a switch's pipeline with the table features message. Both group and meter requests are simple fixed-sized messages, which return bitmaps of the supported feature. A group features response from a switch contains a description of the supported group types (indirect, all, failover, select), the number of groups, and actions available in its buckets. Similarly, a meter features response returns a description of the meter capabilities and the maximum number of meters, colours, and bands that a switch supports.

A table features message is more complex and has a variable length to support different numbers of tables, and different features available to rules within each table such as matches and actions. A table features message allows a controller to either query or optionally set the capabilities of a switch's tables [12], i.e. the switch's pipeline. Setting a switch's table features is primarily designed for programmable pipelines so that a switch can allocate its hardware resources efficiently. The controller configures the entire pipeline at once by requesting the features required of every table. The controller can configure the following features available to rules in a table: instructions, next tables, write actions, apply actions, matches, write set-fields, and apply set-fields. A table features response from a switch returns with the approximate size of the table, along with the full table configuration.

Table features make a distinction between regular rules and a table-miss rule. A controller can specify the features available to the table-miss rule separately to regular rules. A controller can configure if a match is maskable or requires an exact match. By restricting a match to an exact match a switch can often place the match in a cheaper lookup table. Additionally, a controller can configure a match such that it is valid to omit, by adding it to the wildcards list, otherwise, all rules must include the match. Unlike matches, one must assume a rule can omit actions, set-fields, and instructions. However, there is no way to represent if an action, set-field, or instruction is required.

The most significant limitation of the OpenFlow features model is that the model cannot represent mutually exclusive features. Instead, a switch must return an all-encompassing set of features. For example, a controller or switch cannot accurately represent a rule which is either output to a group action or output action but not both using table features. A switch would most likely return that it supports both, yet return a run-time error if the controller attempted to install a rule using both. Conversely, a controller would need to request the pipeline supported both outputting to a group action and an output action together, even if the controller never installs a rule with both.

2.4.2 Table Type Patterns

The Open Networking Foundation (ONF) Forwarding Abstractions Working Group (FAWG) created the Table Type Pattern (TTP) specification to help solve interoperability issues between switches and applications. A TTP [3] is a structured machine and human-readable description of a logical OpenFlow pipeline. The pipeline is described logically as the OpenFlow operations supported by the pipeline rather than being tied to any particular hardware implementation. A TTP is a connect between the controller and switch; a switch must implement all features in the TTP, and a controller must ensure it only uses those supported features.

The TTP specification proposes the following lifecycle for a TTP. Anyone can create a TTP including an application developer to describe their application's requirements, a switch vendor to describe their switch's feature support, and FAWG to create standard TTPs for common use cases. To use a TTP, an application developer must ensure their application only uses the features described, and a device vendor must ensure that their switch supports all features in the TTP. A device vendor with a fixed-function pipeline can describe their pipeline in a TTP, whereas a vendor with a programmable pipeline can optimise placement of rules to use hardware resources efficiently based on a TTP's requirements.

TTPs describe the OpenFlow features available on a switch, most often encoded in JSON. TTPs can represent all the features included in OpenFlow feature messages. Beyond describing just the OpenFlow matches, instructions, or actions available, a TTP can restrict valid values of the match, instruction, or action. Additionally, a TTP supports representing mutually exclusive options through the usage of meta-members. Meta-members enclose lists and restrict the valid combinations depending on their type. For example, a meta-type of *exactly one* allows a TTP to describe mutually exclusive options. TTP table descriptions can include built-in rules, to represent rule behaviour which is built into a pipeline. Built-in rules are particularly useful for representing fixed-function pipelines as they often have non-modifiable table miss behaviour.

TTPs are extensible as they are not a fixed format like OpenFlow feature messages. The TTP author can include additional information by adding a new member to a TTP object description. For our purposes, this is interesting as it means we can store the table size or any other optimisation constraint in the TTP.

Adoption of the TTP standard has been limited; few vendors ship TTP representations of their pipelines, and there are very few tools for creating and working with TTPs. Incentivising switch vendors to support a TTP for an application is also difficult as it requires the vendor to assign business resources to add pipeline support. However, the OF-DPA pipeline, a key target of our research, does ship with a TTP representation of its entire pipeline.

Overall, we decided to use a TTP representation of hardware pipelines. In part due to already having Broadcom's OF-DPA pipeline in this format, and the additional flexibility TTPs have over OpenFlow feature messages. While the lack of tools for working with TTPs is problematic, the situation is not too dissimilar with feature messages, as the common use cases of these messages does not match with our use case. OpenFlow applications use table feature messages to configure a flexible pipeline or to check the required features are supported rather than constructing rules for the pipeline as we are in this thesis. As TTPs can describe all the requirements in OpenFlow feature messages, we expect the conversion to a TTP would be trivial and could be automated. We discuss the structure of TTPs further in Chapter 3 along with the tools we developed to work with them.

2.5 Representing an OpenFlow Rule Match

An OpenFlow rule has three key components: a match, actions, and a priority. A switch evaluates packets received against a rule's match fields and, if matched, applies the rule's actions. If multiple rules match, the rules's priority is used to select one. This section explores representing an OpenFlow match. This section is based on work we have published during the completion of this thesis [17].

Picking an appropriate representation of an OpenFlow match is essential because we need to be able to determine reachability between rules, both of rules at different priorities within the same table, and rules which goto another table. This reachability information identifies dependencies between rules, and can simplify a ruleset by removing unreachable rules. An ideal match representation also allows programmers to determine equivalences between matches, which is a key component in determining equivalent rule transformations and rulesets.

Thinking of an OpenFlow match as the set of packets matched is a useful abstraction, as this allows us to use set operations such as union, intersection,

No. Matches	1	2	3	4
Matches	1***	1*1* 1*0*	10*1 11** 1**0	11*0 10** 1*01 1111

Table 2.1: Equivalent TCAM representations of the match 1*** expanded into multiple matches that are challenging to combine. In the two match case, we see one differing bit which can be merged and replaced with *. However, once three or four matches are involved the relation to 1*** is not apparent. No two wildcards differ by one bit, and no wildcards are redundant, all three or four matches must be considered together to find that they can be merged.

and difference in calculations. The difficulty lies in selecting an efficient set representation as the number of unique packets an OpenFlow rule can match is in excess of 2^{1000}, which is infeasible to work with uncompressed as 2^{1000} items will not fit in memory.

This section introduces the OpenFlow match and the alternative representations: Header Space and Binary Decision Diagram (BDD) representations which both support set operations. We evaluate both representations against two basic operations we require: calculating rule reachability within a table and rule reachability between tables. Rule reachability within a table determines which packets, if any, will reach a given rule. We calculate reachability within a table by subtracting all higher priority rule matches from a given rule match. Rule reachability between tables determines which packets are processed by two rules in different tables, and is calculated as the intersection of the match set between a rule and another rule in the next table[1].

2.5.1 An OpenFlow Rule Match

An OpenFlow rule match can match zero or more header fields. OpenFlow stores match fields as value-mask pairs. Each field can match either a specific value or can be arbitrarily masked to allow a range of values. Most fields in OpenFlow support masking, which a programmer can use to exclude an arbitrary selection of bits from the match field. A packet must satisfy all fields

[1]Reachability within a table is also a factor, but we cover that separately.

included in a match. By default, every packet matches all fields omitted from a match, as such all packets match an empty match. OpenFlow specifies a list of header fields that can be matched, these include all commonly used network protocols from Layer 2 to Layer 4.

This match format conforms to the abilities of the numerous hardware switches that use TCAM to match packets. In general, we refer to this type of per-bit matching as TCAM-style. TCAM matches a series of bits which must all be satisfied. The switch configures each bit to match either a 0, 1 or * (a do not care). This conformity helped drive the adoption of OpenFlow by providing a direct mapping to hardware.

There is more than one way to represent the same match in TCAM. A simple example of this is that any match containing a * bit can be split into two more specific matches for the 0 and 1 case. TCAM matches can overlap each other, so OpenFlow includes a priority with each rule to select precedence. Minimisation of TCAM-style matches has been proven to be an NP-hard problem [18]. We give an example to illustrate the difficulty of simplifying TCAM matches in Table 2.1. Table 2.1 shows three representations of the match 1*** as combinations of matches. While the minimisation of two matches is simple as both wildcards differ by a single bit, it is non-trivial for the three and four match cases. We cannot merge any two rules into a single rule. Instead, all matches must be considered together to find the simplification back to a single rule.

TCAM-style matches support set logic, and we discuss this next with Header Space. Header Space uses a TCAM representation with a more efficient bitwise packing compared to OpenFlow's value-mask pairs.

2.5.2 Header Space

Kazemian et al. [19] introduce Header Space Analysis, a model to represent packets and network boxes as transfer functions. In this section, we focus on Header Space Analysis's representation of packet sets along with the al-

Match	Header Space encoding
0	01_b
1	10_b
*	11_b
z	00_b

Table 2.2: A conversion from a bit match to Header Space. Z is an "annihilator" value which represents an empty set if found in any bit during a calculation.

gorithms to perform union, intersection, difference and complement set operations. Kazemian et al. [19] define Header Space as encompassing all possible packet header values. Header Space only models influential bits of a packet, e.g. only header fields, not payload. Their representation is protocol independent; Header Space flattens fields to a series of bits allowing it to represent any protocol. In Header Space a packet is represented as a point in the space $0, 1^L$, where L is the header length, and a rule matches a region of Header Space.

Header Space objects (or regions) represent a portion of Header Space and a set of packets. The basic building block to represent Header Space regions is a wildcard expression, which is a sequence of bits where each bit can be 0, 1 or *. Wildcard expressions are TCAM-style and analogue an OpenFlow match. Table 2.2 shows the encoding used for wildcards, two bits represent a bit in the packet header. Header Space uses a special value z mapped to 00_b as an "annihilator" to represent an empty set. If any bit becomes z during a calculation the entire set is empty. This encoding enables fast calculation of wildcard intersection using the bitwise AND operation. A wildcard expression alone cannot represent some regions of Packet Space, so Header Space objects are made of a union of wildcards.

Below are the basic set operations for wildcards, some operations result in a set of wildcards which Header Space objects combine as a union. Later we discuss how to apply these operations to Header Space objects.:

- Intersection: The intersection of two wildcards are the packets contained in both wildcards. Due to the encoding used intersection is calculated as the standard bitwise & AND operation of two wildcards, $A \cap B = A\&B$. E.g. 1**0 $\cap$ **1* $=$ 1*10 and 1*** $\cap$ 0*** $=$ z*** $= \emptyset$.

- Union: The union of two wildcards are the packets contained in either wildcard. In general, a single wildcard cannot express a union of two wildcards, so both wildcards are combined in a Header Space object to represent the union.

- Complement: The complement of a wildcard are all other packets in Header Space that are not in that wildcard. The complement operation creates a union of new wildcards for each 0 or 1 bit in the original wildcard with a single bit complemented and all other bits set to *, $\neg A = \neg(A_0 ... A_n) = \{*_0 ... *_{i-1} \neg b_i *_{i+1} ... *_n : (b, i) \in A \wedge b \in \{0, 1\}\}$ where b the value of a bit and i is the bit's offset.
  E.g. $\neg$(1*01) $=$ 0*** $\cup$ **1* $\cup$ ***0 and $\neg$(****) $= \emptyset$

- Difference (aka subtraction): The difference between two wildcards are the packets in the first wildcard but not in the second. The difference is calculated using the existing intersection and complement operations $A - B = A \cap \neg B$.
  E.g. 1**1 - *101
  $=$ 1**1 $\cap$ $\neg$*101
  $=$ 1**1 $\cap$ (*0** $\cup$ **1* $\cup$ ***0)
  $=$ 10*1 $\cup$ 1*11 $\cup$ 1**z
  $=$ 10*1 $\cup$ 1*11

An intersection operation of wildcards always results in a single wildcard or an empty wildcard and is very efficient to compute using bitwise AND in the Header Space representation. We use intersection to compute reachability of rules between OpenFlow tables, so it is crucial intersection performs well. Unions are also efficient as they are generally incompressible so are simply added together in a Header Space object. However, to keep sizes down at least some form of simple elimination of overlapping wildcards is beneficial, which increases compute time.

In contrast, the complement operation and by extension the difference operation results in a substantial expansion of the number of wildcards. Every exact bit match (0 or 1) results in a new wildcard, a match containing a MAC address would expand by 48 times and an exact IPv6 match 128 times. We use difference to calculate the set of packets reaching a rule in a priority-ordered table and to detect unreachable rules, so it is essential difference performs well. To calculate the reachability of a rule in a table, we must subtract all higher priority rules from it. A rule is unreachable if the resulting set is empty.

Recall that Header Space objects contain a union of wildcards, we say $A = a_1 \cup a_2 \cup ...a_n$ where A is the Header Space object and a_x is a wildcard. The intersection of two Header Space objects is the intersection of Cartesian product pairs $A \cap B = \{(a \cap b) : a \in A, b \in B\}$, this results in at worst $|A| * |B|$ wildcards. Some wildcard intersections may result in the empty set which is superfluous in a Header Space object union. The union of two Header Space objects is trivial as the objects are concatenated, in the worst case resulting in $|A| + |B|$ wildcards. The complement of a Header Space object is calculated as the intersection of all wildcards complemented $\neg A = \neg a_1 \cap \neg a_2 \cap ... \neg a_n$. As the complement of a single wildcard results in b new wildcards one per exact bit, the worst case for complementation is $b^{|A|}$. Recall that the difference between two sets is $A - B = A \cap \neg B$, so if we combine intersection and complementation space complexity we find the worst case is $|A| b^{|B|}$.

However, due to this exponential expansion, subtracting just five Ethernet addresses from a catch-all rule to calculate rule reachability within a table quickly exhausts system memory. To highlight this problem, consider a table containing five matches on Ethernet addresses (A, B, C, D, E) and a low priority default rule (F). To find the set of packets reaching F we calculate $F - (A \cup B \cup C \cup D \cup E)$. As Ethernet addresses contain 48 exact match bits, we calculate the worst case expansion to be $|F| b^{|A \cup ... E|} = 1 \cdot 48^5 = 254,803,968$.

Calculation (Cumulative)	Ethernet Address	Actual No. Wildcards	Theoretical No. Wildcards
F	**:**:**:**:**:**	1	1
-A	d8:2c:07:cc:53:ed	48	48
-B	c2:09:4c:bc:7c:e0	1,932	2,304
-C	6b:aa:94:41:4b:a5	64,649	110,592
-D	42:48:5e:3e:e5:16	1,298,260	5,308,416
-E	82:e2:e6:6f:8c:b8	Mem Error	254,803,968

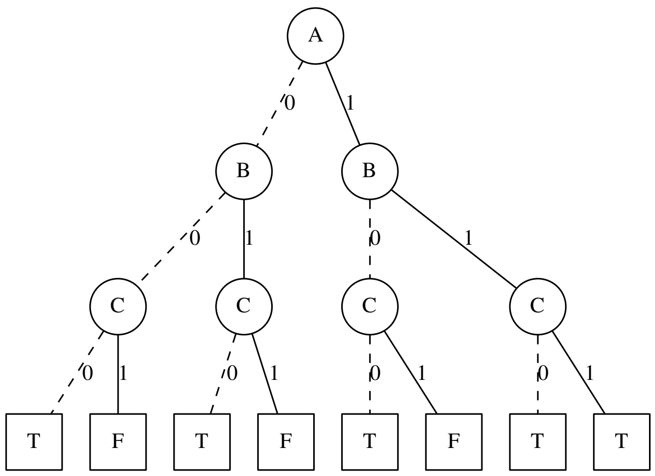
Table 2.3: A demonstration using Header Space objects to subtract five higher priority rules (A-E) to compute the reachability of a lower priority catch-all rule (F). We show that even subtracting just five Ethernet addresses from a catch-all rule quickly exhausts system memory. While removing duplicate wildcards greatly reduces the actual size vs. the theoretical, the scaling is still primarily dominated by the theoretical exponential growth.

We ran this experiment using five random MAC addresses. We show the resulting number of wildcards in Table 2.3. The resulting number of wildcards differs from the theoretical as we have cancelled out duplicate wildcards and empty sets that occur during the calculation. Overall, we find the exponential growth to be the dominating factor and that even attempting to subtract only five addresses quickly runs the system out of memory. Unfortunately, this makes Header Space unsuitable for reachability calculations within a table as these commonly contain thousands of rules. Minimisation of TCAM-style matches is NP-hard [18], so while there is certainly still room for more compression of these wildcards, compressing wildcards quickly becomes prohibitively computationally expensive.

Kazemian et al. [19] also found the difference operation to be expensive and that it slowed down performing network verification, such as verifying host reachability. They improved performance by using lazy evaluation, only evaluating the difference after applying all transfer functions between switches. Lazy evaluation allowed Kazemian et al. [19] to find terms which cancel out before evaluating the difference. However, in our example there is no way to simplify the MAC addresses further as they cannot be merged together. For computing reachability within a table, we find, in practice, lazy evaluation is not helpful as most often no further simplification can be made.

A	B	C	$(A \wedge B) \vee \neg C$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	T
F	T	T	F
F	T	F	T
F	F	T	F
F	F	F	T

(a) Truth Table



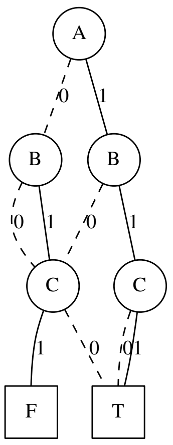
(b) Ordered Binary Decision Diagram

Figure 2.5: Representations of the boolean equation $(A \wedge B) \vee \neg C$

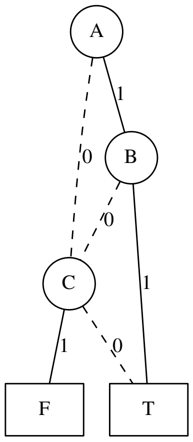
2.5.3 Binary Decision Diagrams

A BDD [20, 21] is a directed acyclic graph used to represent boolean logic. A BDD's layout naturally supports set operations efficiently. A BDD has a single root node, and nodes have a label corresponding to the boolean variable they represent. Each node has two child branches, named low and high, corresponding with the decision made if that variable is false or true. At the end of the graph are terminal nodes which represent the final decision made, either true or false. The truth of a boolean expression for a given set of input values is found by following a path through the BDD from the root node by walking the edge corresponding to the variable's value until encountering a terminal node which holds the overall truth value.

Figure 2.5 shows two different representations of the boolean expression $(A \wedge B) \vee \neg C$. Where $\wedge$ is logical AND, $\vee$ is logical OR, and $\neg$ is logical negation (NOT). Figure 2.5a shows a truth table representation; each row shows the output of $(A \wedge B) \vee \neg C$ for selected input values of the variables A, B and C. Figure 2.5b shows a BDD representation of $(A \wedge B) \vee \neg C$, as pictured A is the root node and the square nodes are used to represent the **T**rue and **F**alse terminal nodes. We draw the low branch as a dotted edge labelled 0 and the high branch as a solid edge labelled 1. If we follow a path from the



(a) BDD with duplicate subgraphs removed



(b) Reduced Ordered BDD

Figure 2.6: Steps applied to reduce the BDD in Figure 2.5b, a representation of the boolean expression $(A \wedge B) \vee \neg C$.

BDD root based on the input values to a termination node, we find each path matches with the truth table. Figure 2.5a is an ordered BDD, which is to say the ordering of nodes down the graph is consistent, i.e. always A, B then C.

Bryant [22] introduces a more useful type of BDD the Reduced Ordered BDD (ROBDD) which adds restrictions to create a more condensed graph. The process of building a reduced BDD obeys two simple rules 1) merge all duplicate subgraphs into one and, 2) remove a node if both of its children are the same subgraph. Figure 2.6 shows these two steps applied to the Ordered BDD in Figure 2.5b. If we first consider merging duplicate subgraphs in the ordered BDD, we observe that the three leftmost C subgraphs are identical, all map C's low branch to a true terminal node and C's high branch to false. We also cannot have duplicate terminal nodes as these are in themselves subgraphs, as such a reduced BDD contains only one copy of each terminal node. Figure 2.6a shows the output after merging duplicate subgraphs. Next, we consider removing nodes with identical subgraphs; these are easy to identify after merging duplicate subgraphs as both branches of a node point to the same subgraph. These nodes are redundant as their value does not affect the terminal node reached. We can see in Figure 2.6a that the leftmost B node and rightmost C node have identical subgraphs. Figure 2.6b shows the final reduced ordered BDD with these nodes eliminated.

ROBDDs are used in practice as they result in much smaller graphs reducing resource requirements of both memory and compute time. Bryant [22] proved that a reduced BDD is the minimal representation, requiring the fewest nodes, for a selected node ordering. Additionally, a reduced BDD is a canonical representation for a selected node ordering [22]. In practice, BDD implementations allocate all BDDs subgraphs from a shared pool [23], as this is more efficient and allows for comparison between BDDs. This implementation detail is particularly useful when checking BDD equivalence, as equivalent BDDs are the same graph and therefore share the same root node in memory. In the remainder of this thesis, we use BDD to mean ROBDD unless otherwise stated.

BDDs can be used to represent sets by mapping items in the set to true or otherwise false. TCAM-style matches are logical expressions in which each bit is a variable, and all bits' matches must be satisfied (i.e. are anded together). Mapping from a TCAM-style match to a BDD is a direct 1 bit to 1 node mapping; wildcarded bits do not affect the decision so do not add a node.

BDDs can easily be combined using set operations. A common base for all BDD operations is Bryant [22]'s *apply* procedure. Bryant [22]'s *apply* procedure takes two BDDs and creates a resulting BDD by applying the operation of choice ($\diamond$) to the terminal nodes. If we consider a BDD node (v, l, h) where v is the variable label, l is the low branch and h is the high branch. Then the *apply* operation can be described recursively to two BDDs a and a' [24]:

$$a \diamond a' = \begin{cases} \mathit{terminal}(\mathit{truth}(a) \diamond \mathit{truth}(a')) & \text{if } a \text{ and } a' \text{ are terminals} \\ \mathit{node}(v, l \diamond l', h \diamond h') & \text{if } v = v' \\ \mathit{node}(v, l \diamond a', h \diamond a') & \text{if } v < v' \\ \mathit{node}(v, a \diamond l', a \diamond h') & \text{if } v > v' \end{cases}$$

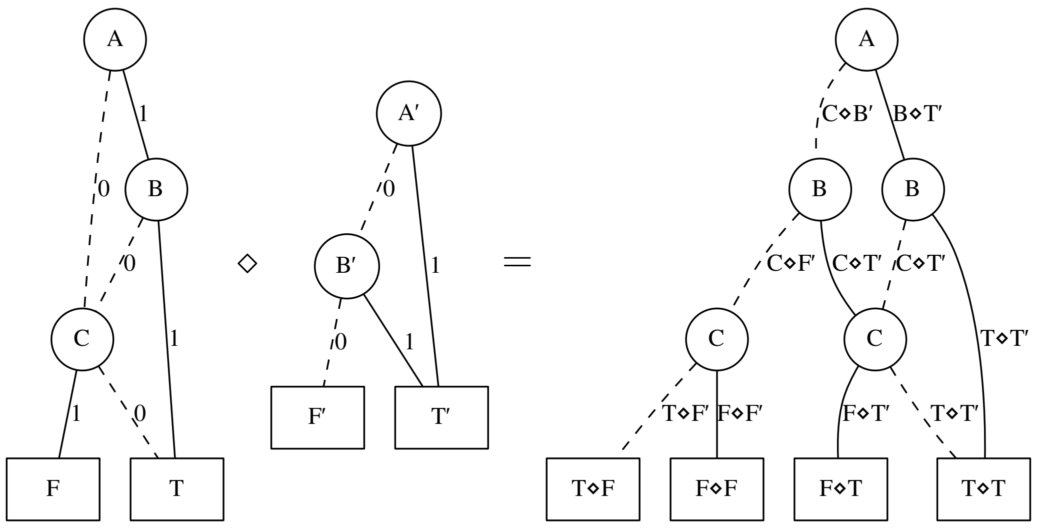


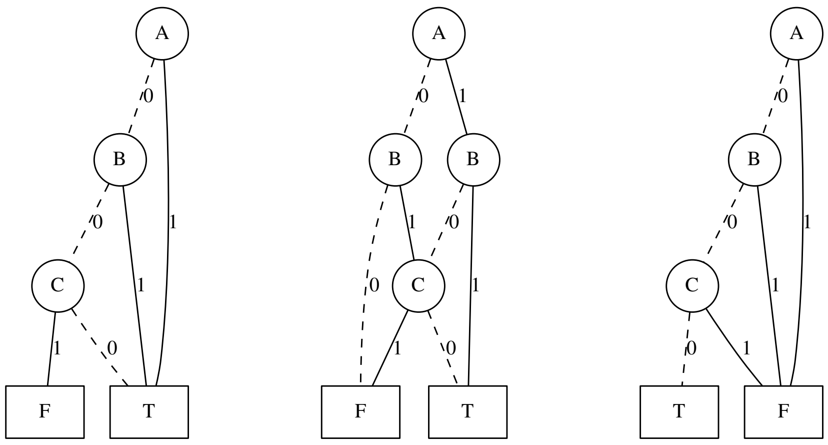
Figure 2.7: A demonstration of a general ◇ operation applied to two BDDs. We have marked the edges in the resulting BDD with the subgraphs combined to create the child node, with the root being the result of $A \diamond A'$. Notice that two branches combine $C \diamond F'$, these always result in the same subgraph, thus are deduplicated. Note that depending on the operation all combinations of the terminal nodes will compute to either true or false further reducing the graph.

Figure 2.7 applies this recursive process to two BDDs. The *apply* procedure walks both BDDs in unison, following the same decision path from both root nodes until reaching two terminal nodes, recursively to consider all paths. The *apply* procedure applies the operation to these terminal nodes to calculate the new terminal node in the resulting BDD. In the case the walk encounters mismatched nodes, the lowest ordered node is selected, in this way forming a shared path which maps to a single terminal value in both BDDs.

If we consider our BDD as a set and we *apply* some operation (◇) $A \diamond A'$. Then our ◇ operation is given two boolean inputs corresponding to an item's existence in both sets A and A' and should return true if the item is in the resulting set otherwise false. The logic for these set operations are as follows:

Union $(A \cup A')$ creates a new set with the combined items of both sets. We apply the logical OR operation, $A \vee A'$.

Intersection $(A \cap A')$ creates a new set containing items present in both sets. We apply the logical AND operation, $A \wedge A'$.



(a) Union $A \cup A'$ (b) Intersection $A \cap A'$ (c) Difference $A - A'$

Figure 2.8: The union, intersection, and difference set operations applied to A and A' from Figure 2.7. The resulting BDDs are smaller (contain fewer nodes) than the general result pictured in Figure 2.7 because the operation replaces the four terminal nodes with either True or False.

Complement $\neg A$ creates a set of all items not contained in the BDD. Complementation simply requires swapping the true and false terminals or alternatively *applying* logical XOR against the true terminal.

Difference/Subtraction($A - A'$) creates a new set of the items containing items present in the first set but not in the second set. We apply the logical operation $A \wedge \neg A'$.

Figure 2.8 shows the resulting BDD after *applying* the three basic set operations to $A \diamond A'$ shown in Figure 2.7.

In the worst case, the space complexity of any *apply* BDD operation is the product of the nodes in each BDD [22], as in the worst case every node would need to be combined with every other. However, the worst case complexity is most often not met, Knuth [24] notes that this complexity in many cases is closer to the sum of the nodes. An *apply* operation often attempts the same combination of nodes in different subgraphs, e.g. the two branches combining $C \diamond F'$ in Figure 2.7; implementations maintain a lookup cache to avoid calculating the resulting subgraph again. This caching makes the time complexity of an apply operation the same as the space complexity. The node ordering

Calculation (Cumulative)	Ethernet Address	Actual No. Node	Theoretical No. Node
F	**:**:**:**:**:**	1	1
-A	d8:2c:07:cc:53:ed	50	50
-B	c2:09:4c:bc:7c:e0	93	2,500
-C	6b:aa:94:41:4b:a5	138	125,000
-D	42:48:5e:3e:e5:16	181	6,250,000
-E	82:e2:e6:6f:8c:b8	224	312,500,000

Table 2.4: A demonstration using BDDs to subtract five higher priority rules (A-E) to compute the reachability of a lower priority catch-all rule (F). Notice F contains a single node, which is a true terminal, while A-E contain 50 nodes corresponding to 48 bits and two terminal nodes. While the theoretical worst case space complexity is exponential, the actual scaling we observe is linear. The results show that combining exact matches scales linearly, which is a typical pattern in flow tables.

chosen for a BDD affects its size [22, 24].

We too have found combining flow table rules most often results in linear expansion, rather than the worst case exponential expansion. Now we consider the problem of computing reachability between rules in different tables which uses intersection. Recall that the intersection of two TCAM-style wildcards results in at most one wildcard and that wildcards have a 1:1 mapping from exact bit matches into BDD nodes. So the intersection of two TCAM-style matches results in, at worst, the sum of the nodes with an upper bound of the total number of matchable bits. This does not always hold true when taking the intersection of two BDDs in general, only when intersecting TCAM-style matches.

Now we reconsider the problem of testing rule reachability within an OpenFlow table, the same as we did for Header Space in Section 2.5.2 which quickly expanded in size and exhausted memory. Consider a table containing five matches on Ethernet addresses (A, B, C, D, E) and a low priority default rule (F). To find the set of packets reaching F we calculate $F-(A \cup B \cup C \cup D \cup E)$. As Ethernet addresses contain 48 exact match bits, and as a BDD contains 50 nodes; the 2 extra are the true and false terminal node. The theoretical worst case expansion is 50^n, where n is the number of MAC addresses subtracted.

Table 2.4 shows our experimental results of computing reachability within a OpenFlow table using BDDs. The theoretical worst case scaling is exponential; however, we find the actual scaling to be linear. BDD's scaling is much better suited to our usage than TCAM-style representations like Header Space (Table 2.3), which were dominated by the worst case exponential scaling.

2.5.3.1 Variations of Binary Decision Diagrams for Networking

Many variations of Binary Decision Diagrams exist, often specialised for a specific usage while others, here we highlight some of those which have been used in network research.

Firewall Decision Diagrams (FDDs) [25] are a variant of BDDs in which each node represents a header field instead of a single boolean decision, i.e. a single bit. Nodes can have multiple branches rather than only two as in a traditional BDD. Each edge is labelled with a set of integers corresponding to the field's value, such that all edges leaving one node are non-overlapping sets and their union encompasses all possible values of a field. In an FDD terminal nodes map to either a drop or accept decision. FDDs efficiently encode firewall rules as a series of integer ranges per header field, which maps to the style of filtering rules firewalls accept. Gouda and Liu [25] designed algorithms using an FDD to compress firewall sets into fewer rules while maintaining equivalent behaviour.

Thus far, the BDDs explored map to a binary result, either true or false (accept or drop). Clarke et al. [26, 4] generalise the BDD to allow any finite set of terminal nodes forming the Multi-Terminal Binary Decision Diagram (MTBDD). MTBDD retain the canonical and minimal characteristics of a standard BDDs. MTBDDs are useful for representing non-binary decisions such as an action associated with a match. We further discuss using this canonical MTBDD representation for forwarding ruleset equivalence checking in Chapter 4.

Smolka et al. [27] and Arashloo et al. [28] used BDD structures to compile high-level languages into physical network topologies. Smolka et al. [27] introduced the Forwarding Decision Diagram, and Arashloo et al. [28] extended the Forwarding Decision Diagram to encode stateful operations. Both found the FDD structure is efficient and offers chances for optimisation through removing equivalences. A Forwarding Decision Diagram differs from an MTBDD as a decision node considers an entire value of the field rather than a single bit and cannot represent the arbitrary masking found in OpenFlow.

Yang and Lam [29] use a BDD representation of packet-space to create atomic predicates to verify forwarding behaviour, including reachability, loop detection and black hole detection. They found their approach was fast enough to run in real-time. BDDs and MTBDDs have also been used to represent firewall ACL rules for verification checking [30, 31] and as a structure for fast packet classification [30, 32, 33].

2.5.4 Summary

We have found representing OpenFlow match sets as TCAM-style matches, such as Header Space, to be suitable for calculating simple intersection operations, however, is unsuitable for calculations requiring the difference to between match sets due to exponential scaling in practice. Header Space intersection is very fast as it simplifies to the bitwise AND operation.

BDDs provide an alternative representation, which does not suffer from exponential scaling in practice, despite exponential scaling being the theoretical worse case complexity. BDDs also have an additional advantage over TCAM-style matches, they are canonical. Overall, we have found the performance and scaling characteristics of BDDs to be sufficient for our usage.

Our research makes use of both Header Space and BDD representations where appropriate. Merging rules between tables requires intersection, so we use a Header Space representation. While to check ruleset equivalence we use an MTBDD as it is a canonical representation, and, in practice, scales almost linearly with the number of rules.

Chapter 3

Working With Table Type Patterns

Representing a switch's pipeline in a machine digestible format is essential to being able to determine where a controller can install an OpenFlow rule. Determining where a rule can be installed is an integral component in our ultimate goal of fitting an existing ruleset to a new pipeline. The pipeline representation needs to be able to represent the requirements of a switch accurately, and yet be as convenient as possible to both create and use.

We chose to use a Table Type Pattern (TTP) [3] as input to our rule fitting algorithm to describe the target OpenFlow pipeline. TTPs provide a flexible representation of an OpenFlow pipeline, which is both machine and human-readable. A TTP can represent detailed requirements such as the values a rule can match in addition to the header field that can be matched. Complex dependencies between requirements can be represented including mutually exclusive options and all-or-nothing options. Additionally, Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) pipeline included a TTP description which is a crucial target for our rule fitting algorithm due to the prevalence of Broadcom switching chips in OpenFlow hardware.

The largest problem we found with using TTPs was the ecosystem is undeveloped. There is a lack of tools for working with TTPs, there are few developers publishing TTPs, and there are errors in published TTPs. We found that many errors in TTPs are typographical in nature, including issues such as using the wrong match field name (MAC_DST vs. the correct ETH_DST), which a developed ecosystem of tools would have been discovered.

In addition to using a TTP to represent a pipeline for rule fitting, we created two tools for working with TTPs and a library with an extension to fit rules into a TTP. The tools we created to assist with viewing and verifying TTPs are useful not only to us but to the entire TTP ecosystem to improve the quality of TTPs and speed up the process of creating and using TTPs. We have also created a library to find valid rule placements in a TTP, which could be used to assist the Software-Defined Networking (SDN) application development process. An application developer could include a rule verification check against a TTP in their automated testing. Thus allowing multiple OpenFlow switches to be tested without the need to have access to hardware while also providing near instant feedback after every code change. For researchers, our TTP library is a useful base to build network modelling and verification tools.

In this chapter, we detail the structure of Table Type Patterns and the tools we developed to assist with creating and interpreting TTPs.

3.1 Contents of a TTP

A TTP is a text file that describes the capabilities of an OpenFlow pipeline. A TTP consists of objects which contain members (a key-value mapping) and other basic types such as lists, strings and numbers. In practice, JSON is used to encode a TTP, and all examples in the specification use JSON encoding. The standard also allows the TTP author to use other machine-readable encodings such as YAML and XML. However, we have only encountered JSON encoded TTPs. An author writes a TTP for a specific version of OpenFlow, which determines the matches and actions available. The tools we have created are

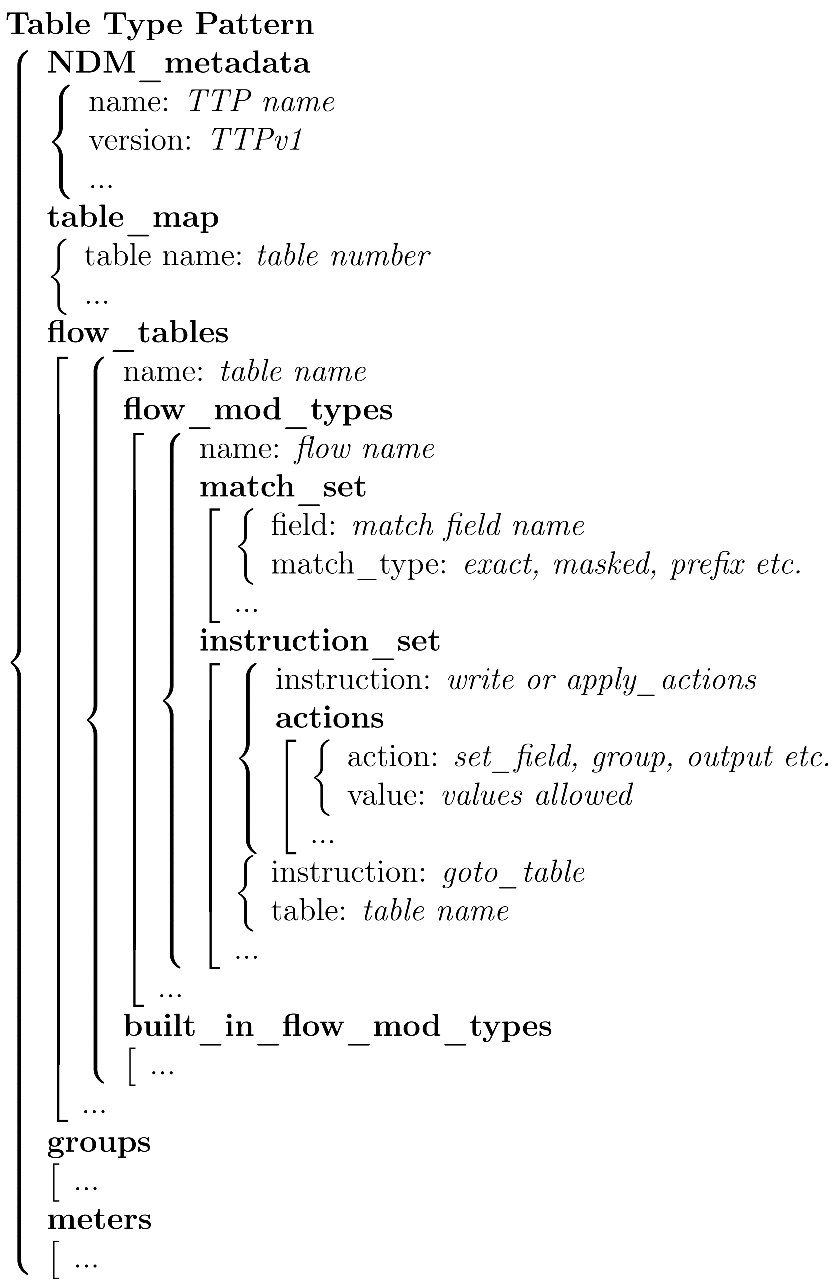


Figure 3.1: The basic hierarchy of a Table Type Pattern where a curly bracket ({) represents an object and a square bracket ([) represents a list of objects. Objects are deeply nested in this hierarchy, for example, *actions* are nested eight levels deep. These actions can further reference other top-level objects such as groups, which a program also needs to interpret.

designed to work with a JSON encoding and OpenFlow 1.3.

Figure 3.1 shows the condensed TTP hierarchy of the most important features for OpenFlow 1.3; square brackets show lists ([) and curly brackets ({) show objects. The top level of a Table Type Pattern is an object containing up to eleven possible members; we discuss four of the most important. The *NDM_metadata* member supplies information about the TTP; the TTP version, OpenFlow version and the TTP's author. The *table_map* member maps human-readable table names to their number in the pipeline. Throughout the TTP, objects reference these human-readable table names, such as in a goto_table instruction, whereas an OpenFlow application must use the table number to install rules. The *flow_tables* member includes descriptions of all flow tables and includes descriptions of the rules the table will accept. Additionally, tables can include built-in rules which cannot be modified and are commonly used to represent a fixed-function pipeline's table-miss behaviour. Rule descriptions describe the valid priority, match field and instruction combinations of a rule. Within a rule's instructions, a distinction exists between apply-actions and write-actions. Apply-actions and write-actions contain a list of TTP *actions* describing the constraints. This nesting within the TTP structure is deep in places; *actions* are at least eight levels deep within lists and objects, meta-members can make this even deeper.

The *groups* top-level member contains a list of group descriptions. Each group description can constrain the type of group and the actions supported in each *bucket*. In OpenFlow groups are an indirect way to execute actions, groups contain zero or more buckets which each contain a set of actions. The type of group determines which buckets the switch executes. For example, when a switch executes a group of type *all* it executes all buckets while for a group of type *select* it executes one bucket based on the selected load-balancing algorithm. A group output action elsewhere in the TTP can include a reference to a group description by name to restrict the available groups for the action. Similar top-level member descriptions exist for other OpenFlow features like *meters*.

When a controller is evaluating the validity of a rule, it must meet all requirements described by the TTP. For example, if a *flow mod type* lists two exact matches in its *match set* for Ethernet destination and IPv4 destination, a controller can only install a rule that contains both an exact match on Ethernet destination and IPv4 destination. The controller cannot exclude one or both of these matches.

A TTP author can describe optional and more complex requirements by adding meta-members around lists which describe how many items in the list are required. Meta-members can be placed anywhere within a TTP. Meta-members can be of the type *all*, *one or more*, *zero or more*, *zero or one*, and *exactly one* which correspond to the number of items which must be satisfied[1]. The default type is *all* meaning every item in the list must be satisfied; conversely, *zero or one* makes every item in the list optional. Meta-members can be nested to create further combinations. In addition to meta-members, objects can include an *opt_tag*, which is a named optional feature which should either be included or excluded in its entirety. In addition to the meta-member and opt_tag mechanism available to all objects in the TTP, some objects have member values which make them optional. For example, match field descriptions have a type which is one of *exact*, *mask*, *prefix* or *all_or_exact*. For all match types, except *exact*, the field can optionally be omitted from a rule's match unless another requirement exists.

While meta-members are flexible, it quickly becomes complex to correctly parse nested meta-members at different levels of the TTP hierarchy and reduces the human readability. Figure 3.2 shows a simplified excerpt of complex meta-member usage within the OF-DPA TTP. Notice that all actions are within the outermost list which has a default *all* requirement, meaning all items within that list must be satisfied. However, the rule's action-set does not need to include any actions as both innermost lists are wrapped within meta-members with zero as an option. An action-set cannot include more than one GROUP action as these are limited to selecting either *zero or one*. In contrast, the *zero or more* meta-member allows the action-set to include

[1]The TTP standard makes a distinction between meta-member types as being either a *use meta-member* or *support meta-member*, to differentiate between what a switch must support vs. what an application can use. However, in practice this differentiation is ignored by the TTPs we have encountered, we treat all as *use meta-members*.

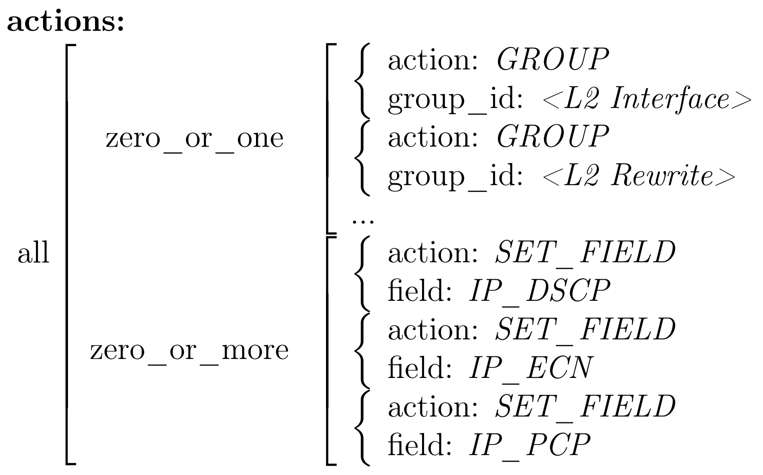


Figure 3.2: Example of meta-member usage in the OF-DPA TTP taken from the IPv4 VLAN rule entry in the ACL table.

any number of SET_FIELD actions.

While TTPs are a machine-readable standard, some complex, less common or non-standard requirements are often delegated to a documentation string. We have found the TTP descriptions of rules that a pipeline accepts are almost entirely machine-readable. Most non-machine readable descriptions are due to non-standard additions to OpenFlow or any other hardware quirks. The human-readability of a TTP is cumbersome as they are often large and the hierarchy results in deeply nested objects which are hard to follow. The Forwarding Abstractions Working Group (FAWG) did not release any tools with the TTP standard, this was left to the community, and as a result, very few tools exist for working with TTPs. As part of our research, we have created a set of tools and a library to assist with checking a TTP is valid, and finding valid placements for a rule within a TTP.

3.2 Our TTP Library and Tools

We developed a Python library for working with TTPs, which loads a TTP as a normalised representation. Additionally, the library's loading process also validates the TTP, detects errors, missing components, and contradictions, allowing the input to be corrected. On top of this base library, we have built three tools: one for validating a TTP and suggesting corrections, one for viewing a TTP, and an extension to our library to fit rules into a TTP.

Our library retains the original TTP hierarchy and layout, built from subclasses of a base TTPObject. For example, a TTPTable object stores the TTP table description and is a subclass of TTPObject. Additionally, lists including TTPActions and TTPMatches all inherit from a base TTPList class. A TTPList stores the meta-member type and can be nested. We have designed the library so that a developer can add methods to the base TTPObject and individual subclasses to perform a procedure on the TTP. By adding methods directly to the TTPObject procedures can recursively walk the TTP hierarchy while still performing different operations depending on the object type. The ability to recursively walk a TTP is essential as even a simple task like finding all the match fields supported by a pipeline requires walking through all rules in all tables while accounting for nested meta-members (TTPLists) which are allowed anywhere within the hierarchy. The library includes basic methods to find all instances of an object type (such as matches), resolve a name to an instance (e.g. a table name to the TTPTable instance), and printing objects.

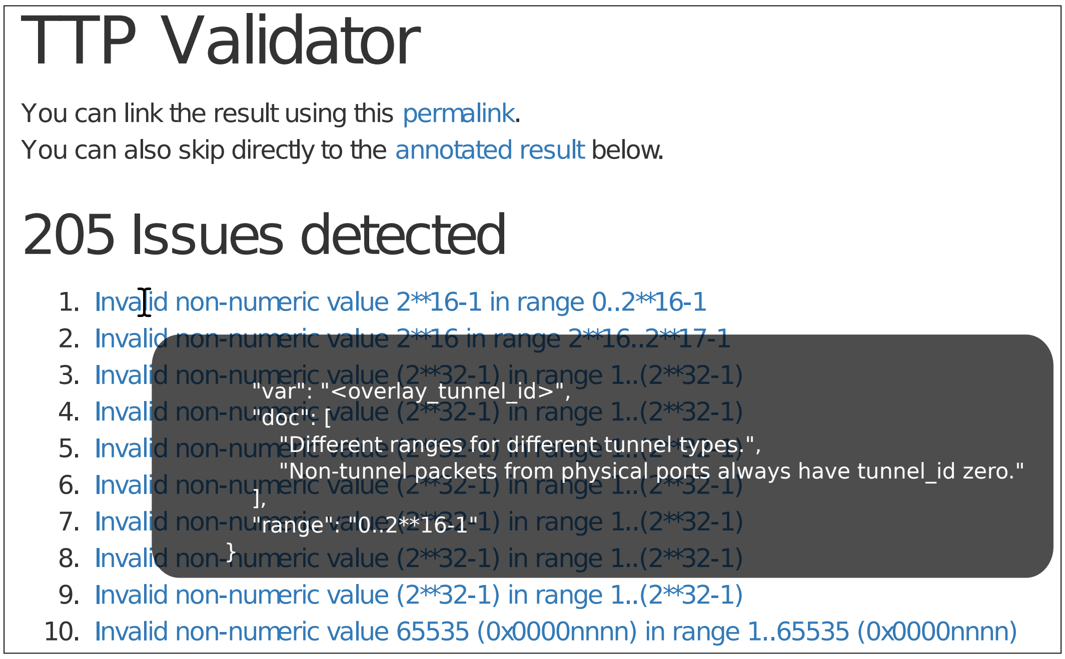
3.2.1 Loading and Validating a Table Type Pattern

Loading a Table Type Pattern from JSON uses Python's built-in JSON library. The JSON library loads the TTP into standard Python types including lists, dictionaries (maps), strings and integers. Once loaded from JSON, the library converts all the resulting standard Python types into our TTP object types. This process starts from the bottom of the TTP and walks through the hierarchy converting all objects and lists encountered into our TTPObject and TTPList subclasses. As part of the conversion, the library sanity checks the TTP, including type checking values, normalising values and replacing named references within the TTP with that object. The library normalises values such as Ethernet, IPv4 and IPv6 address, to integer values from their original string format.

Key to guiding the development of the TTP library and loading a TTP was the ability to load existing TTPs. In particular, we ensured the library could load all the example TTPs released by FAWG and Broadcom's OF-DPA TTP. Broadcom's OF-DPA pipeline describes their entire fixed-function pipeline; it is large and complex, it is more than 12,000 lines long and has numerous OpenFlow extensions for custom header fields and actions. As toolsets for working with TTP are non-existent, mistakes in TTPs are common, and we wanted to be able to automatically resolve common mistakes or remove issues so that parsing could continue.

The library logs all errors and issues encountered loading a TTP. Whenever possible, the library removes invalid values encountered in the TTP and continues parsing. Due to the size of a TTP and the lack of prior tools for working with TTPs we have found that errors in TTP are common and being more accepting and continuing after an issue allows most TTPs to be machine-read. Additionally, continuing parsing after encountering an error allows all issues to be found and fixed in one go, rather than having to fix each new issue encountered incrementally.

When loading a TTP, the library compares the names of matches, instructions, and actions with those in the OpenFlow specification to ensure they are valid. The library checks that values associated with a header field in a match or set field action TTP description are within the range of that field and will truncate the value and log a warning for any violations. All built-in rules are checked to ensure that they have exact values specified for the matches, instructions and actions applied in the rule. After loading all objects from a TTP, the library checks all references within the TTP are valid and refer to an object. Again, logging warnings for any missing references during this step.


TTP Validator

You can link the result using this permalink.
You can also skip directly to the annotated result below.

205 Issues detected

1. Invalid non-numeric value 2**16-1 in range 0..2**16-1
2. Invalid non-numeric value 2**16 in range 2**16..2**17-1
3. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
4. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
5. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
6. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
7. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
8. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
9. Invalid non-numeric value (2**32-1) in range 1..(2**32-1)
10. Invalid non-numeric value 65535 (0x0000nnnn) in range 1..65535 (0x0000nnnn)

```
    "var": "<overlay_tunnel_id>",
    "doc": [
        "Different ranges for different tunnel types.",
        "Non-tunnel packets from physical ports always have tunnel_id zero."
    ],
    "range": "0..2**16-1"
}
```

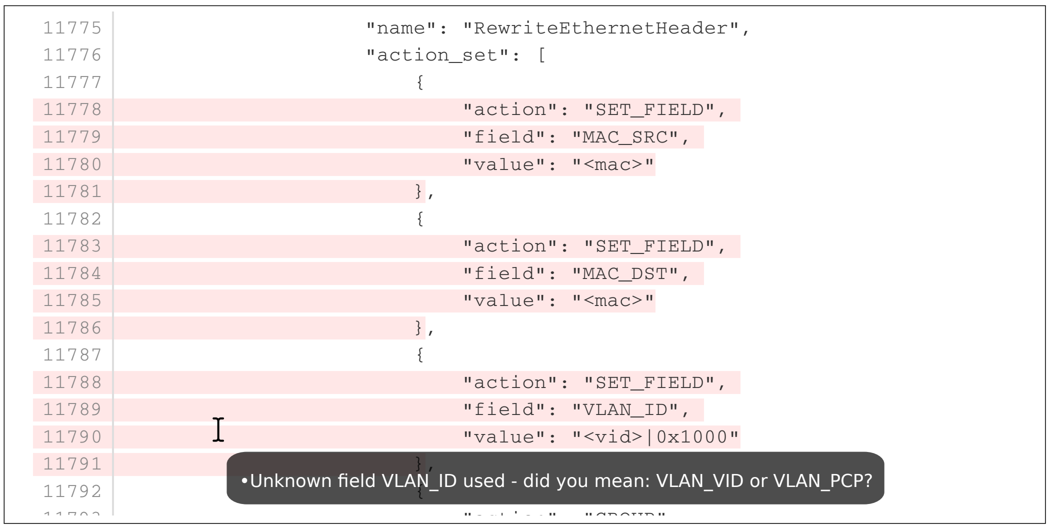
Figure 3.3: The top of the TTP validator's HTML visualisation which shows a list of issues found. When a user clicks an issue the page scrolls to the object highlighted within the original JSON. Hovering over an issue displays a tooltip with the related object as it appeared in JSON.

3.2.2 Viewing Issues with a Table Type Pattern

As mistakes in TTPs are prevalent, we developed a visualisation for displaying the errors detected by our library during the loading process. The visualisation makes it easy to find and correct any issues. We opted to create a webpage to visualise errors found in a TTP as this was accessible and allowed for interactivity such as jumping to an issue and providing additional information on the issue encountered in a tooltip.

We call this tool a TTP validator, however, most of the validation code is in the base TTP library which does the validation when is loads a TTP. The TTP validator tool is primarily responsible for presenting the issues found. The validator is written in Python and uses the Flask [34] web framework to construct and serve the results as a webpage. A user can run the validator as a local standalone server. Additionally, we have published a version online here: `https://wand.nz/ttp-validator/`.

Figure 3.3 shows the beginning of the HTML webpage returned for the OF-DPA TTP pipeline. All issues found are enumerated at the top of the page,



```
                    "name": "RewriteEthernetHeader",
                    "action_set": [
                        {
                            "action": "SET_FIELD",
                            "field": "MAC_SRC",
                            "value": "<mac>"
                        },
                        {
                            "action": "SET_FIELD",
                            "field": "MAC_DST",
                            "value": "<mac>"
                        },
                        {
                            "action": "SET_FIELD",
                            "field": "VLAN_ID",
                            "value": "<vid>|0x1000"
                        },
                        {
```

•Unknown field VLAN_ID used - did you mean: VLAN_VID or VLAN_PCP?

Figure 3.4: Visualisation of issues found by our TTP validator in the OF-DPA TTP version 1.2.2 (Revision 2). The visualisation highlights the original JSON object that caused an issue during loading. When the user hovers the mouse over an issue, a tooltip appears with information about the issue and suggested remediation. All three objects highlighted have used an incorrect field name, MAC_SRC and MAC_DST should be ETH_SRC and ETH_DST. VLAN_ID should be VLAN_VID. The shown tooltip suggests both VLAN_VID and VLAN_PCP as corrections. The lefthand side of the page tracks the line number.

followed by the original JSON TTP annotated with the issues found. The webpage is interactive. Hovering over an issue will display the JSON object in a tooltip, this provides context for the issue selected. Figure 3.3 shows the tooltip for the first issue which appears as text in a black box. Clicking an issue scrolls to the related object in the annotated JSON, this provides the full context of that object with relation to the TTP hierarchy.

Figure 3.4 shows a section of the annotated JSON TTP for the OF-DPA pipeline. If the validator finds an issue, it highlights the object in the original JSON input. The user can then easily find and edit the object in the original JSON using the associated line number. If a user hovers over a highlighted JSON object a tooltip displays the information about the issue, this includes details of the issue which will include the name of the member and the issue with the value found and, where possible, the suggested remediation.

The primary challenge encountered with this visualisation is mapping an issue back to the original location in the JSON. Typically a JSON library loads a JSON file from a textual form into the appropriate types (list, hash map, int, string etc.) for the programming language and all further processing takes place on these types. However, the visualisation requires a mapping from these types back to their original location in the textual JSON representation. To accomplish this, we customised the JSON loading process to attach the original location to all JSON objects. Python loads a JSON object into a hash map. Our validator adds a callback hook to the JSON object loader which stores the object's start and end character offset in the original input as member entries in the resulting hash map.

When the library loads a TTP, it logs any issues it finds. Additionally, this logging attaches the start and end character of the corresponding object to the log message. In addition to logging, all issues encountered are also maintained in a list against the Table Type Pattern. The message logged depends on the issue encountered. The TTP library's loading process type checks and converts most values within the TTP to their expected type. For example, if the library expected an integer but found a string, and no conversion is possible, it logs the issue and removes the value so it can continue. In cases where OpenFlow header field, action, or instruction identifiers are expected the library verifies these against both the OpenFlow standard and any additional identifiers defined by the TTP. If the library cannot find an identifier of that name, it logs the issue along with suggested corrections as remediation which it finds from closely matching valid field names. Figure 3.4 shows an example of this process with VLAN_ID, an invalid header field in OpenFlow, which correctly suggests VLAN_VID as a replacement. Minor capitalisation and whitespace issues will generate a warning but are automatically corrected.

3.2.3 Viewing a Table Type Pattern

As Table Type Patterns can be thousands of lines long, in the case of the OF-DPA TTP over 12,000 lines of JSON, they are often hard to read. In many cases, an entire rule's matches or actions will not fit on the screen. So we developed a command line tool to traverse and view the hierarchy of a TTP in sections compactly. This tool dramatically enhances the human readability of a TTP.

The viewing tool loads a TTP and presents a list of options to choose from at the top of the TTP hierarchy. Here is the top level menu:

```
1) TTP Info
2) Security
3) Variables and Extension Identifiers
4) Tables
5) Groups
q) quit
Which one?
```

When the user selects an option, the tool presents either another list of options for that level of the hierarchy or a compact representation of the TTP object. The security item in the top level of the TTP hierarchy is a documentation string containing security guidance, while the other items relate to OpenFlow directly and were covered in Section 3.1. The tool is particularly useful for exploring the rules available in each table and printing these in a condensed format. Here is the *IPv4 Multicast MAC* rule description from OF-DPA in the Termination MAC table which allows multicast traffic to be directed to a routing table rather than L2 switching:

```
IPv4 Multicast MAC
    Doc: Enables IPv4 multicast routing.
    Priority: 2
    Matches: all(ETH_TYPE!=0x800,ETH_DST=0
       ↪ x1005e000000/0xffffff800000)
    Instructions:
        GOTO_TABLE: Multicast Routing
        APPLY_ACTIONS: zero_or_one(OUTPUT=CONTROLLER)
```

In the original JSON formatting, this description takes up 38 lines, much more than the 7 above. The TTP viewer prints condensed version of the rule match descriptions on a single line in the following format:

```
FIELD_NAME[!][@][*]=[value][/mask]
```

The symbol suffixes have the following meanings: '!' denotes an exact match, '@' denotes a prefix match, and '*' denotes that including the match field is optional. The value and mask are optional in a TTP match description and mean that the rule's match must too have this exact value or mask. Here we can see the switch requires the rule to have specific values for its matches, ETH_TYPE must match 0x800 (i.e. match only IPv4 packets) and ETH_DST must match 01:00:5E:00:00:00 with a mask of ff:ff:ff:80:00:00 corresponding to RFC 1112 multicast addresses. A rule must send this multicast IPv4 traffic to the Multicast Routing table and optionally to the controller. Both the packets matched by this rule and the action taken cannot be modified. However, the SDN application can still choose not to install the rule.

Next, we show the *IPv4 VLAN* rule description from the ACL table in OF-DPA which allows for flexible filtering of packets:

```
IPv4 VLAN
    Doc: IPv4 - switched/bridged packet including
       ↪ encapsulated MPLS
    Priority: 2
    Matches: all(IN_PORT!*,ETH_TYPE!*,ETH_SRC*,
       ↪ ETH_DST*,VLAN_VID,VLAN_PCP!*,$VLAN_DEI!*,
       ↪ $VRF!*,IPv4_SRC*,IPv4_DST*,IP_PROTO!*,
       ↪ IP_DSCP!*,IP_ECN!*,TCP_SRC!*,UDP_SRC!*,
       ↪ SCTP_SRC!*,ICMPV4_TYPE!*,ICMPV4_CODE!*,
       ↪ TCP_DST!*,UDP_DST!*,SCTP_DST!*,ARP_SPA*)
    Instructions:
        zero_or_one(METER)
        zero_or_one(GOTO_TABLE: Color Based Actions)
        zero_or_one(CLEAR_ACTIONS)
        zero_or_one(APPLY_ACTIONS: zero_or_more(
           ↪ $COLOR_ACTIONS_INDEX=<
           ↪ color_actions_index>,$COLOR=<color>,
           ↪ $TRAFFIC_CLASS=<traffic-class>,OUTPUT=
           ↪ CONTROLLER))
        zero_or_one(WRITE_ACTIONS: all(zero_or_one(
           ↪ GROUP=<L2 Interface>,GROUP=<L2
           ↪ Unfiltered_Interface>,GROUP=<L2 Rewrite
           ↪ >,GROUP=<L2 Multicast>,GROUP=<L3 Unicast
           ↪ >,GROUP=<L3 Multicast>,GROUP=<L3 ECMP>),
           ↪ zero_or_more(IP_DSCP=<ip_dscp>,IP_ECN=<
           ↪ ip_ecn>,VLAN_PCP=<pcp>)))
```

It is one of the most extensive rules as it supports almost every type of match field, instruction, and action. In the original TTP JSON, it takes up 234 lines, much more than can comfortably fit on a screen at a time. Match fields, instructions or actions prefixed with a dollar sign ($) are non-standard OpenFlow extensions that OF-DPA has defined. In the *IPv4 VLAN* rule description all match fields, except VLAN_VID, are optional, and all instructions are optional.

This tool presents most machine-readable elements of TTP descriptions compactly. It has been beneficial for quickly determining if a table will accept a rule. This tool has been invaluable in debugging our ruleset transformation algorithm and debugging mistakes in TTPs.

3.2.4 Fitting a Rule into a Table Type Pattern

For an SDN developer, it is essential to be able to check where they can install a rule into a switch's pipeline. It is also a requirement of transforming rulesets to new pipelines as we will discuss in Chapter 5. We created the TTPSatisfies library to fulfil this purpose. Loading the TTPSatisfies library adds methods for checking if an OpenFlow rule satisfies the requirements of a TTP description to the objects created by the base TTP library. To add methods to existing objects, we created an *@extend_class* decorator which functions like an Extension Method in C# [35]. By separating these libraries, the rule-fitting logic and additional objects representations such as rules are kept separate from the TTP object representation and validation logic. With this pattern, it is simple to add functionality and mix and match such libraries.

Checking if a rule satisfies the requirements of a TTP description requires recursing the TTP hierarchy in all its complexity, most notability meta-members. The TTPSatisifies library adds a method named SATISFIES to all TTPObjects, which checks if the corresponding component of an OpenFlow rule satisfies the requirements of the TTPObject description. An OpenFlow rule corresponds to a TTPFlow description, an OpenFlow match to a TTP-

Algorithm 3.1 The SATISFIES method of a TTPMatch class

```
Input: this The TTPMatch description to satisfy - this.field: is the header-
    field.
Input: unplaced The unplaced matches; a map, header-field to value+mask.
Input: placed The placed matches; a map, header-field to value+mask.
Input: final A boolean, if true returns only fully placed matches.
Output: placements A list of (unplaced, placed) pairs.
function TTPMATCH.SATISFIES(this, unplaced, placed, final)
    placements ← list()
    if IS_OPTIONAL(this) then                ▷ Can be excluded from the rule
        placements.append(PAIR(unplaced, placed))
    end if
    if this.field in unplaced then
        match_value ← unplaced[this.field]
        if match_value satisfies this's requirements then
            unplaced ← COPY(unplaced)
            unplaced.remove(this.field)
            placed ← COPY(placed)
            placed[this.field] ← match_value
            placements.append(PAIR(unplaced, placed))
        end if
    end if
    if final then
        for item in placements do
            unplaced, placed ← item
            if !IS_EMPTY(unplaced) then
                placements.remove(item)
            end if
        end for
    end if
    return placements
end function
```

Match description and so on. The SATISFIES method of a TTPFlow calculates its result recursively, by calling SATISFIES on the rule's matches and instructions.

SATISFIES takes three arguments: 1) the portion of the input rule remaining to place (initially the entire input rule), 2) the portion of the rule placed so far (initially empty), and 3) a *final* flag set true only on the first call. SATISFIES returns a list of placements, which contain both the placed portion of the input rule and its corresponding unplaced portion. The SATISFIES method of each TTPObject moves the portion satisfied from the input rule to the placed rule, if a required condition is not satisfied then the return is an empty list. However, this does not necessarily mean that placement has failed, as a TTPMatch may return zero valid placements but itself can be within an optional branch of a meta-member which is valid to exclude. The *final* flag is set true for the first call to SATISFIES and filters the resulting placements to include only fully placed rules, i.e. when the unplaced portion of the rule is empty. If any component of the original rule remains unplaced, be it a match, instruction or action then the rule as a whole cannot be installed.

To better illustrate this process Algorithm 3.1 shows the pseudo code for how a TTPMatch processes a set of matches to return all conditions which satisfy it. A TTPMatch ends the recursion down the TTP hierarchy as it is the deepest object in the hierarchy. A TTPMatch describes the pipeline's restrictions on a header field such as its maskability and constraints on the values of the match's value and mask. TTPMATCH.SATISFIES returns between zero and two placements depending on whether its conditions are satisfied. Each placement contains an unplaced and placed portion of the match set. Lines 3-5 create the first placement if the match field is optional in which case it is satisfied without inclusion. Lines 6-15 create the second placement if this match field exists in the unplaced rule and it meets all the constraints of the TTPMatch. This placement is created by moving the match from unplaced to placed. Lines 16-23 are run if the *final* flag is true, and filter the placement returned as to only return fully placed rules.

Within our TTP library the TTPList base class stores a list of TTP objects including matches, instructions and actions. A TTPList also stores the meta-type requirement which describes the valid combinations of the items contained. We use a common SATISFIES implementation for objects subclassing a TTPList which handles the complexities of the meta-member restrictions. Algorithm 3.2 shows the pseudo code for the TTPList SATISFIES method. Algorithm 3.2 highlights both how calls to SATISFIES are chained together and how to compute meta-member constraints.

Lines 3-15 handle the default 'all' case for a TTPList, where all items in

Algorithm 3.2 The SATISFIES method of a TTPList class

```
Input: this The TTPList object to satisfy - this.meta_type: is the
    meta_type constraint.
Input: unplaced The unplaced portions of the OpenFlow object
Input: placed The placed portions of the OpenFlow object
Input: final A boolean, if true returns only fully placed rules.
Output: placements A list of (unplaced, placed) pairs.
 1: function TTPLIST.SATISFIES(this, unplaced, placed, final)
 2:     placements ← list()
 3:     if this.meta_type = "all" then
 4:         placements.append(PAIR(unplaced, placed))
 5:         for item ∈ this do
 6:             accumulated ← list()
 7:             for up, p ∈ placements do
 8:                 accumulated ← concatenate(accumulated,
                                        item.SATISFIES(up, p, False))
 9:             end for
10:             if EMPTY(accumulated) then
11:                 return accumulated        ▷ A condition cannot be satisfied
12:             end if
13:             placements ← accumulated
14:         end for
15:     else if this.meta_type = "zero_or_one" or "exactly_one" then
16:         initial ← list()
17:         initial.append(PAIR(unplaced, placed))
18:         if this.meta_type = "zero_or_one" then
19:             placements ← COPY(initial)
20:         else
21:             placements ← list()
22:         end if
23:         for item ∈ this do
24:             placements ← concatenate(placements,
                                item.SATISFIES(unplaced, placed, False))
25:         end for
26:     else if this.meta_type ="zero_or_more"or "one_or_more " then
27:         ...
28:     end if
29:     if final then
30:         Remove all pairs with unplaced portions from placements
31:     end if
32:     return placements
33: end function
```

the list must be satisfied. Lines 5-14 accumulate all valid placements that satisfy an item, and then check if those placements also satisfy the next item, until all items in the list have been considered. Line 4 seeds the process with the original input placed and unplaced pair, and then lines 7-9 find all new placements which satisfy the current item. The *accumulated* placements satisfy all items up to and including the current item. The extra loop for lines 7-9 is required as multiple valid placements are possible; recall that a TTPMatch can return two placements. Additionally, because TTPLists can be nested themselves, an item in the list may be another TTPList, which can return multiple placements. Lines 10-12 check if any item has returned an empty list as this fails the 'all' condition of the TTPList and therefore TTPList.satisfies must return an empty list too. If the conditions of all items are satisfied, then line 32 returns the valid placements.

Lines 15-26 check if the TTPList is satisfied when it has either the *zero-or-one* and *exactly-one* meta-type. *Zero-or-one* shares the *exactly-one* logic but additionally is more permissive allowing the zero case where a rule satisfies no items in the list. Lines 18-20 add this zero case to the returned placements, which is the input placement unmodified. Lines 23-25 create all valid placements for the *exactly-one* case. Lines 23-25 take the initial placement and find the placements that satisfy each item one-by-one and collects all resulting valid placements.

We have omitted the code for the *zero-or-more* and *one-or-more* meta-types, but these follow a similar process to determine the valid placements. Lastly, lines 29-31 will filter the result if this is the final satisfies method so that only fully placed items are returned. We have omitted the code, as it has already been shown in Algorithm 3.1 lines 16-23.

3.2.4.1 Rule Fitting for Ruleset Transformation

TTP rule fitting functionality is a fundamental component of our ruleset transformation algorithm discussed later in Chapter 5. To find more placements which have or are likely to have equivalent behaviour we allow some deviation in the placement of the input rule's actions.

If a TTP action list description includes group actions, SATISFIES recurses into that group and continues attempting to place the actions from the original rule. Recursing into groups allows a rule with an output action to be installed in a table which only supports output via indirect groups. SATISFIES can also fit to groups of the *'all'* type, and will check nested groups.

Although apply-actions and write-actions have different behaviour; in many cases, including when the rule is in the last table, the behaviour is equivalent. Because write-actions and apply-actions can be equivalent, SATISFIES treats all actions equally and attempts placement into both write-actions and apply-actions. Ignoring the difference between write and apply-actions increases the chance of finding a successful placement, particularly in a pipeline such as OF-DPA which requires write-actions rather than apply-actions in most tables.

Similarly, the clear-actions instruction does nothing if the packet's action set is already empty. SATISFIES returns both versions of rules with and without clear-action instructions as allowed by the TTP description.

Due to splitting actions between groups and apply and write-action instructions the action ordering may change, which may result in a non-equivalent action set. As such a caller should verify the return rules have equivalent behaviour.

3.2.4.2 Optimisation

We added an optimisation to SATISFIES to end processing early if a rule cannot possibly be placed to avoid traversing unnecessary parts of the hierarchy.

To fully place a rule all of its matches, instructions and actions must be fully placed. If any component fails to be fully placed, then the entire rule cannot be placed. As such SATISFIES first checks if the rule's matches can be fully placed, if not processing ends early. We chose matches because they are

Algorithm 3.3 Creating match list requirements bitmasks

```
Input: self: the TTPMatchList
Output: ($required, optional$): bitmasks of the header-fields that a rule requires or can optionally include to satisfy this. Each bit in *required* or *optional* corresponds to a header-field.
1: function TTPMatchList.generate_masks(self)
2:     $opt\_masks \leftarrow$ list()
3:     $req\_masks \leftarrow$ list()
4:     for $match \in self$ do
5:         $opt\_masks.append(match.get\_opt\_mask())$
6:         $req\_masks.append(match.req\_opt\_mask())$
7:     end for
8:     if $self.meta\_type =$"all" then
9:         required $\leftarrow$ reduce($|, req\_masks, 0$)
10:        optional $\leftarrow$ reduce($|, opt\_masks, 0$)
11:    else if $self.meta\_type$ in ("one_or_more" or "exactly_one") then
12:        required $\leftarrow$ reduce($\&, req\_masks, 0$)
13:        optional $\leftarrow$ reduce($|, req\_masks, 0$) | reduce($|, opt\_masks, 0$)
14:    else if $self.meta\_type$ in ("zero_or_more", "zero_or_one") then
15:        required $\leftarrow 0$
16:        optional $\leftarrow$ reduce($|, req\_masks, 0$) | reduce($|, opt\_masks, 0$)
17:    end if
18:    return ($required, optional$)
19: end function
20: function Reduce($op, items, default$)
21:    if empty($items$) then return default
22:    end if
23:    return $items_0$ $op$ $items_1$ $op$ ... $items_n$
24: end function
```

simple and therefore, faster to check than instructions and actions.

Additionally, we optimised the process of satisfying matches to detect and avoid branches of the TTP description which cannot possibly be satisfied. TTPMatchList descriptions store two bitmaps one of the header-fields required in a rule and another of the header-fields optional in a rule.

Algorithm 3.3 shows how SATISFIES calculates required and optional match bitmasks for a TTPMatchList. The calculation must factor in meta-member constraints to ensure both:

1. The required bitmask has bits set for header-fields that must be present in a rule's match to satisfy the rule, i.e. if (*match_bitmask* & *required*) $\neq$ *required* then the rule cannot possibly be fully satisfied.

2. The optional bitmask has bits set for header-fields that are possible to match, i.e. if $(match\_bitmask \,\&\, (required|optional)) \neq match\_bitmask$ then the rule cannot possibly be fully satisfied.

Both the optional and required checks allow SATISFIES to break early, but if both tests pass then a full check is required as these bitmasks do not represent the full complexity of requirements. Lines 20-24, REDUCE combines a list of bitmasks together using the selected bitwise operation (op).

Lines 4-7 loop and collect the required masks and optional masks from each match. A match will either return a bit set in the required or optional bitmask, however, recall that a TTPMatchList can contain another TTPMatchList which itself returns the bitmasks from GENERATE_MASKS. Meta-type *all* is only satisfied when all items in the list are satisfied, so the match list's required and optional masks are the bitwise OR (|) combination of each item's mask (Lines 8-11). To satisfy meta-type *one-or-more* and *exactly-one* one item must be satisfied and because each rule can choose a different item to satisfy and leave out the others all required items become optional. Hence the match list's optional bitmask is the bitwise OR combination of every item's required and optional bitmasks (Line 13). However, if every item requires a specific match field then it must always be picked, hence bitwise AND (&) is used to combine the required bitmasks (Line 12). The meta-types *zero-or-more* and *zero-or-one* allow zero items to be satisfied. Therefore all items are optional, so the required bitmask for the list is zero (Line 15). As any number of items in the list can be satisfied the match list's optional bitmask is the bitwise OR combination of required and optional bitmasks.

Chapter 4

Ruleset Equivalence Checking

In this chapter, we present our method to compare the forwarding equivalence of two OpenFlow rulesets; however, our technique applies to all match-action pipelines. Checking ruleset equivalence has a direct application for our work in validating the success of a ruleset transformation, and we use it as part of our solver discussed further in Chapter 5. It also has a broader application for the research community as equivalence checking can be used to verify that an optimisation, code rewrite or alternative application maintains equivalent behaviour. This chapter is based on work we have published during the completion of this thesis [17]. We have released our implementation to the research community [36].

4.1 Problem Overview and Terminology

We present our forwarding equivalence checking in the context of OpenFlow 1.3 [12], a popular Software-Defined Networking (SDN) standard. OpenFlow 1.3 exposes a programmable multi-table match-action pipeline as we described in Section 2.2. The set of rules installed in these tables define the forwarding behaviour of an OpenFlow switch.

We use the term packet-space to refer to any set of packets, in particular, the values set in matchable header fields. An empty packet-space contains no packets, and a full packet-space contains all possible values of packet headers.

Table 1

	Priority	Match	Write Action	Apply Action
A	10	**01	Output:1	GotoTable:2
B	9	*010	Output:2	GotoTable:2
C	0	****		

Table 2

	Priority	Match	Write Action	Apply Action
D	100	1***	Clear	
E	0	****		

Figure 4.1: A multi-table pipeline which makes forwarding decisions in Table 1 and performs firewall filtering in Table 2.

Table 1

	Priority	Match	Write Action	Apply Action
A	100	1***		
B	0	****		GotoTable:2

Table 2

	Priority	Match	Write Action	Apply Action
C	10	*010		Output:2
D	10	*001		Output:1
E	10	*101		Output:1
F	0	****		

Figure 4.2: A multi-table pipeline which performs firewall filtering in Table 1 and makes forwarding decisions in Table 2.

An OpenFlow rule matches a packet-space to an action. An OpenFlow table defines actions for the full packet-space because any unmatched packets will have the default action applied. These actions determine the forwarding of a packet. We define forwarding behaviour for a packet to be the ports (if any) it egresses and all modifications made to the packet, which can vary per port. A ruleset is equivalent if the forwarding behaviour is equivalent for the full packet-space, i.e. every possible value of packet header. We do not consider non-forwarding behaviour such as packet and byte counters which are attached to rules but do not affect forwarding.

Figure 4.1 and 4.2 both represent a simplified two table pipeline which forwards based on the lower three bits of the match and firewalls based on the highest bit. Both pipelines have equivalent forwarding behaviour. In Fig. 4.1 rules A and B apply forwarding by adding the corresponding output action to the packet's *action set*. In the second table, rule D applies firewalling by clearing the *action set* for packets with a 1 high bit match thus removing any output actions and dropping the packet. The remaining packets will match E which terminates processing and applies the forwarding behaviour stored in the *action set*. Fig. 4.2 performs firewalling in the first table, rule A drops all packets with a 1 high bit match and B sends the remaining packets to the second table where rules C, D and E apply forwarding.

We emphasise that even though both rulesets have equivalent forwarding behaviour the matches used between these rulesets are different, in Fig. 4.1 rule A forwards to port 1, whereas this is split into rules D and E in Fig. 4.2. The rulesets also use different actions, Fig. 4.1 uses *write actions* to set and then later clear forwarding, while Fig. 4.2 applies forwarding using *apply actions*. Thus, our aim is to find a canonical way to represent both mapping priority-ordered matches to actions and finding a canonical form for actions to represent forwarding behaviour.

4.2 Ruleset Conversion to a Canonical Form

We convert a ruleset into a Multi-Terminal Binary Decision Diagram (MTBDD) based form which is canonical for the same ruleset forwarding behaviour. This conversion process has 3 key steps:

1. Flattening multi-table pipelines to an equivalent single-table representation using cross product merging of rules (§4.2.1).

2. Converting the actions applied by the flattened rules to a canonical representation of forwarding behaviour (§4.2.2).

3. Building a canonical representation of packet-space mapped to the canonical forwarding behaviour using an MTBDD (§4.2.3).

This final MTBDD representation is trivially comparable to check equivalence of two rulesets' forwarding behaviour. Additionally, we show how it

Algorithm 4.1 Flatten OpenFlow tables to a single-table equivalence

```
Input: $Tables$ Lists of original rules per table
Output: $T_s$ The resulting single table equivalence
 1: function FLATTEN_TABLES($first$, $TableIndex$)
 2:     $ST \leftarrow$ Empty Table/List
 3:     for all $second \in Tables[TableIndex]$ do
 4:         $merged \leftarrow$ MERGE($first$, $second$)    ▷ MERGE as per description in §4.2.1
 5:         if $merged$ != NULL then
 6:             if $merged.GotoTable$ then
 7:                 $ST$.add(FLATTEN_TABLES($merged$, $merged.GotoTable$))
 8:             else
 9:                 $ST$.add($merged$)
10:             end if
11:         end if
12:     end for
13:     return $ST$
14: end function
15: $EmptyRule \leftarrow$ An Empty Rule
16: $T_s \leftarrow$ FLATTENTABLES($EmptyRule$, 0)
17: return $T_s$
```

is possible to perform other operations on this representation such as finding the packet-space with a differing forwarding behaviour (§4.2.3.1). Section 2.5 discusses the characteristics of the MTBDD and Header Space match representations which we use in this conversion process.

4.2.1 Conversion to a Single-Table Equivalence

The first step in equivalence checking is to convert a multi-table pipeline to an equivalent single-table, thus simplifying the problem. Algorithm 4.1 describes the recursive approach we take to flatten a multi-table pipeline to a single-table equivalence. FLATTENTABLES recursively computes the cross product of all rules; all rules in a table are merged with all rules in the next table they *goto* recursively until only one table remains. The recursive process is started in the first table and with an empty rule which acts as an identity element in the MERGE operation. Cross product conversion has been used in prior work [37, 38], however, they do not detail how to merge rules where fields which have been previously set by *apply actions* and how to merge *write* *actions.*

Below we outline how to merge the individual components of a rule with another, (i.e. the MERGE operation). We say the first rule is merged with the second rule in the next table. The result is a single rule with equivalent forwarding behaviour for only the packet-space which is matched by both rules:

Matches: The merged rule must only match the packet-space matched by both rules, therefore, the intersection of the matches. If the intersection is empty, then packets cannot possibly hit both rules, so we do not create a merged rule. We must consider a special case if the first rule modifies (using *apply actions* or *write metadata*) a field that the second rule matches. Algorithm 4.2 shows the bitwise operation to calculate a merged match for one ternary bit (t-bit) (0, 1 or both *). Where M_l and M_r are one t-bit of the left and second matches and W_l is the t-bit written by the first *apply actions* (* if not modified). Lines 2-6 simulate the value of the t-bit reaching the second rule, 7-9 check that the intersection[1] is not empty (packets can hit both rules). Lines 10-14 determine the packet-space that will be accepted by both rules. If the t-bit was set by the first rule any t-bit matching the first rule will be accepted, otherwise, the intersection of the left and second rules' match is accepted.

Write actions: *Write actions* are merged as if they were *action sets* following the order of OpenFlow pipeline processing; first applying *clear action* instructions and then overwriting existing values with any applied later in the pipeline.

Apply Actions: *Apply actions* are simply concatenated in pipeline processing order, see [12, pg. 16].

Priority: When flattening tables relative priorities must be maintained, in pipeline processing priority, e.g. in Fig. 4.1 the priority order of merged rules from highest to lowest is: A+D, A+E, B+D, B+E, C+D, and C+E. The relative priority order of the first rule takes precedence over the second

[1]Intersection ($\cap$) as defined in Header Space [39]

Algorithm 4.2 Merging OpenFlow matches (bitwise)

```
Input: $W_f$ A t-bit written by the first rule
Input: $M_f$ A t-bit of the matches of the first rule
Input: $M_s$ A t-bit of the matches of the second rule
Output: $M_n$ The new merged match as a t-bit
 1: function MERGE_BITWISE($W_f$, $M_f$, $M_s$)
 2:     if $W_f = *$ then
 3:         $ps \leftarrow M_f$
 4:     else
 5:         $ps \leftarrow W_f$
 6:     end if
 7:     if $ps \cap M_s = \varnothing$ then
 8:         return NULL                          ▷ No overlapping packet-space
 9:     end if
10:     if $W_f = *$ then
11:         $M_n \leftarrow M_f \cap M_s$
12:     else
13:         $M_n \leftarrow M_f$
14:     end if
15:     return $M_n$
16: end function
```

rule. We achieve this by scaling all priorities based on the table that they are installed into to allow enough space between two adjacent priorities to fit all priorities of subsequent tables, using this formula:

$$new\_priority = priority \times (MaxPriority^{|tables|-1-table_{index}}) \tag{4.1}$$

These scaled priorities are merged using addition. Because all components of the merge operation are associative, the operation as a whole is also. This means the order in which tables are combined is irrelevant to the result. However, we recommend working from the first table, to avoid unnecessarily computing unreachable paths, as shown in Algorithm 4.1.

An OpenFlow rule cannot match an inner tag field (such as VLAN QinQ and MPLS) in a single-table, as only the outermost tag can be matched [38]. However, matching inner tags is possible in a multi-table pipeline by first popping the outer tag and matching the inner tag, which is now the outer, in the next table. In order to represent matching an inner tag in our single-

Original	Minimal	Notes	Resolved By
group([output:1])	output:1	groups add indirection when outputting packets	Step (2)
output:1, set vlan-vid:1	output:1	changes made to a dropped packet are irrelevant	Step (2)
set vlan-vid:1, set vlan-vid:2, output:1	set vlan-vid:2, output:1	an overwritten set-field is redundant	Step (3)
set vlan-vid:1, pop vlan, output:1	pop vlan, output:1	fields set on popped headers are redundant	Step (3)
push vlan, pop vlan, output:1	output:1	push+pop pairs are redundant	Step (3)

Table 4.1: Base-cases of equivalent operations in OpenFlow 1.3 we identified, showing an example of the original action set, compared to a minimal representation and the step in our process which resolves them.

table equivalence, we create new match fields to express matching the n^{th} tag, allowing all multi-table pipelines to be expressed in a single table. Our paper [17, alg. 3.3 and 3.4] provides further detail as how we calculate the merge operation when rules use these tag fields.

4.2.2 Identifying Equivalent Actions

Identifying equivalent actions is difficult because OpenFlow has many ways to represent the same behaviour. Base cases of equivalent action sequences are listed in Table 4.1, which we resolve by converting to a minimal form.

We represent actions per output port minimised and ordered to create a canonical form. We are careful to perform this canonicalisation in a dependency-aware manner to ensure actions are not removed or reordered in a way which changes forwarding behaviour. We define a dependency between two actions if performing them in reverse order will result in different forwarding behaviour. For example, every set-field is dependent on itself, setting VLAN-ID is additionally dependent on push and pop VLAN operations, and all actions share a dependency with output actions. Our process of converting actions to this canonical format is detailed below and is shown by example in Figure 4.3.

Apply Actions	VID:1, Out:1, group([*push VLAN, MAC:A, VID:2, Out:2*])	
Write Actions	Out:3, VID:3	
(1) Combined Action List	VID:1, Out:1, group([*push VLAN, MAC:A, VID:2, Out:2*]),VID:3, Out:3	
	Output	Actions
(2) Flatten Groups	1	VID:1
	2	VID:1, push VLAN, MAC:A, VID:2
	3	VID:1, VID:3
(3) Remove Redundant Operations	1	VID:1
	2	VID:1, push VLAN, MAC:A, VID:2
	3	VID:3
(4) Topographical Sort	1	VID:1
	2	MAC:A, VID:1, push VLAN, VID:2
	3	VID:3

VID*=set VLAN ID,* ***Out****=output port,* ***group*** *is indirect*

Figure 4.3: Canonicalisation of a complex action list.

1. We combine a rule's *write actions* and *apply actions* into a single apply actions list. The *write actions* are appended to the end of *apply actions* in the processing order detailed by OpenFlow [12, pg. 27]. An example output of this step is shown in Figure 4.3 Step 1.

2. We flatten all groups and output actions by creating a mapping of the output port to actions (Figure 4.3 Step 2). The process walks from the start of the combined action list collecting the actions applied to the packet. When an output action is found, instead of collecting it we map the output port to the actions collected thus far. When a group is encountered we make a copy of the actions collected so far for each group bucket, then we walk each bucket; removing the group while preserving its behaviour.

3. We walk through the list finding and removing redundant actions. These include duplicate set-fields, and sequences such as push VLAN, set VLAN, pop VLAN. Figure 4.3 Step 3 removes the redundant VID:1 action on output 3. These cannot be removed if another dependency of a different type is found in-between, as seen with output 2; the push VLAN

action between the VID stops VID:1 being removed as VID:1 refers to a different VLAN header to VID:2.

4. We perform a topographical sort on the list to normalise ordering while maintaining dependency ordering. Figure 4.3 Step 4 shows how **M**AC:A is sorted alphabetically before **V**ID:1. But, a dependency exists between **V**ID:1 and **P**ush VLAN which prohibits reordering.

The process normalises OpenFlow groups to flat representations of output actions, minimises the result by removing redundant actions, and finally sorts the result while maintaining dependencies. The result of this process is a minimal canonical representation of forwarding behaviour through an OpenFlow pipeline.

4.2.3 Equivalent Ruleset Behaviour

Now we have a canonical way to represent forwarding behaviour, we need to be able to check that the complete packet-space mapping to forwarding behaviour is the same. We need to find a canonical representation of partial packet-space to action mapping, for the complete packet-space. The difficulty lies in finding an efficient method of representing sections of packet-space. Naively representing packet-space per packet requires 2^k packets, where k is the number of matchable bits in the packet header, which is infeasibly large. OpenFlow match-style TCAM representations such as Header Space [39], which provides bitwise logic to perform set operations on packet-spaces, are not a canonical representation. In Section 2.5, we found the alternative method of checking equivalence, using set difference, results in a huge expansion quickly exhausting memory for TCAM representations.

Instead, we use a MTBDD [4, 26] to represent a full packet-space to forwarding behaviour mapping. While Binary Decision Diagrams (BDDs) traditionally return either True or False at their terminal nodes, a MTBDDs is a generalisation which allows any finite set of terminal nodes, allowing our

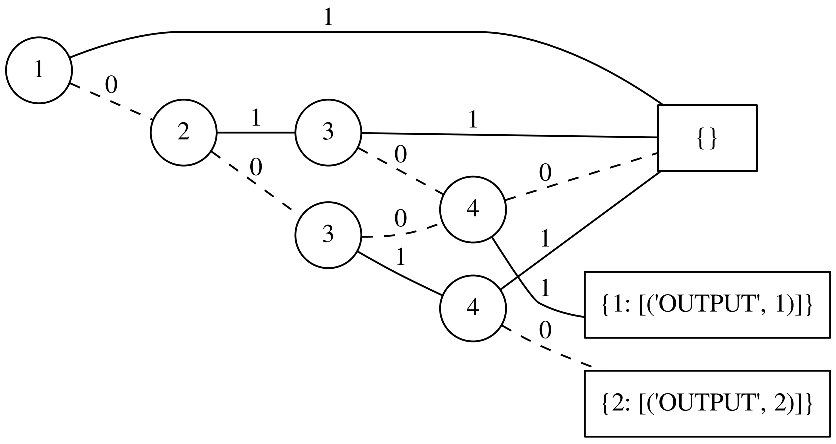


Figure 4.4: The canonical MTBDD representation of the equivalent rulesets shown in Figures 4.1 and 4.2.

Algorithm 4.3 Convert a rule to a BDD

```
Func: GETNODE($num, zero, one$) Creates/retrieves an existing BDD node
Input: $terminals$ The terminal cache, indexed by [action]
Input: $bits$ Matches as t-bits                         ▷ 0, 1 or *(do not care)
Input: $action$ A canonical form of the rule's actions
Output: $BDDRoot$ The resulting BDD representation
  $BDDRoot \leftarrow terminals[action]$
  $numBits \leftarrow |bits|$
  for $i \leftarrow numBits - 1$ to 0 do                ▷ MSB to LSB ordering
      if $bits[i] = 0$ then
          $BDDRoot \leftarrow$ GETNODE($i, BDDRoot, \varnothing$)
      else if $bits[i] = 1$ then
          $BDDRoot \leftarrow$ GETNODE($i, \varnothing, BDDRoot$)
      end if                                 ▷ $bits[i] = *$ does not add a node
  end for
```

canonical representation of forwarding behaviour to be used as terminal nodes.

Figure 4.4 shows a complete canonical MTBDD representation of the ruleset in Figure 4.1. Each node is numbered corresponding to a bit in the packet header with 1 being the most significant bit used for firewalling. The branches from each node represent the forwarding decision made if that bit is 0 or 1 for any given packet, the terminal (leaf) node holds the forwarding decision. A special terminal node ∅ is used to represent empty packet-space, allowing a BDD to represent a partial packet-space. BDD equivalence is trivial to check. Equivalent BDDs have identical root nodes because a reduced BDD maintains only one instance of every subgraph.

Converting an OpenFlow Rule to a Partial BDD: OpenFlow matches are first represented as a series of t-bits (0, 1 or *); all matchable fields are concatenated together in a consistent order and fields not included in the match are filled with *. The 40 matchable fields in OpenFlow 1.3.5 are represented in 1261 t-bits. Algorithm 4.3 shows the conversion to a BDD; the result is a BDD representing a partial packet-space defined for packets matching the rule with the remaining packet-space empty (∅). We build the BDD in reverse from the terminal node up to the root for efficiency. Only 0 and 1 t-bits add nodes to the BDD, * does not require a node. The node ordering chosen within the BDD will change the BDD size, however, finding the best node ordering is NP-complete [40]. We have found using a node ordering so that the most significant bit of a field is stored in the lowest numbered node (i.e. top of the graph) more efficiently stores prefix matches[2].

Converting a Priority Ordered Table to a BDD: Representing individual rules as partial BDDs is not sufficient, as this ignores the priority order in OpenFlow tables. Algorithm 4.4 details the process of converting a priority ordered list of partial BDDs to a full BDD for the complete packet-space. The intuition is that lower priority rules can only match the packet-space not already represented in higher priority rules. As such, lower priority rules can only fill empty (∅) space in the BDD. Lines 1-4 add each rule to an empty BDD from high to low priority; PriorityAdd takes the highest priority as its first argument. The PriorityAdd function recursively walks all nodes in both graphs in unison until leaves are found (lines 12-21), this is the common basis of all BDD operations, called the apply operation. Lines 6-11 define the PriorityAdd operation that ends the recursion. If at anytime the left (higher priority) BDD becomes empty the right BDD (lower priority) will be returned, otherwise reaching a terminal node on the left or an empty right side will return the left.

For better performance we have found building the BDD using the Divide-

[2]We have not explored heuristics to optimise field ordering

Algorithm 4.4 Convert a flow table to a BDD (Naive)

```
Input: nodes The node cache, indexed by [num, zero, one]
Input: rules A priority ordered list of rules represented as BDDs Algorithm 4.3
Output: BDD A full representation of the forwarding behaviour
 1: $BDD \leftarrow \varnothing$
 2: for all $f \in rules$ do
 3:     $BDD \leftarrow$ PriorityAdd($BDD$, $f$)
 4: end for
 5: function PriorityAdd($l$, $r$)
 6:     if $l = \varnothing$ then
 7:         return $r$
 8:     end if
 9:     if IsTerminal($l$) or $r = \varnothing$ then
10:         return $l$
11:     end if
12:     if $l$.num = $r$.num then
13:         return nodes[$l$.num, PriorityAdd($l$.zero, $r$.zero),
14:                         PriorityAdd($l$.one, $r$.one)]
15:     else if $l$.num < $r$.num then
16:         return nodes[$l$.num, PriorityAdd($l$.zero, $r$),
17:                         PriorityAdd($l$.one, $r$)]
18:     else if $l$.num > $r$.num then
19:         return nodes[$r$.num, PriorityAdd($l$, $r$.zero),
20:                         PriorityAdd($l$, $r$.one)]
21:     end if
22: end function
```

and-Conquer (D&C) approach, shown in Algorithm 4.5, is much faster than the naive approach. Algorithm 4.5 works in a similar fashion to merge sort, even and odd numbered rules in the list are combined pairwise (lines 3-5) resulting in a list half the size repeatedly until one final BDD remains as checked by line 1. Lines 6-8 check if the list of BDDs is uneven and will add the remaining BDD, which has no pair, to the new list. Compared to the naive approach, the same number of PriorityAdds are performed however most are working with smaller BDDs, while the naive approach adds a small BDD to an ever-growing BDD.

4.2.3.1 Finding Different Forwarding Behaviour

In order to identify the packet-space with different forwarding behaviour between two rulesets, we define the BDD difference operation. We can easily map this

Algorithm 4.5 Convert a flow table to a BDD (D&C)

```
1: while $|rules| > 1$ do
2:     $newRules \leftarrow$ Empty List
3:     for all $r1 \in even(rules); r2 \in odd(rules)$ do
4:         $newRules.append($PRIORITYADD(r1,r2)$)$
5:     end for
6:     if $mod(|rules|, 2) = 1$ then
7:         $newRules.append(rules[-1])$
8:     end if
9:     $rules \leftarrow newRules$
10: end while
11: $BDD \leftarrow rules[0]$
```

packet-space representing the difference back to the OpenFlow rules involved for further analysis. The difference operation is logically similar to a set difference, BDDs naturally support such operations. The difference operation of two BDDs, $l - r$, will return a tuple (l, r) for packet-space where the actions of l and r differ, otherwise $\varnothing$. Algorithm 4.6 shows the termination check which is applied recursively using the BDD APPLY operation; therefore it replaces Algorithm 4.4 lines 6-11. The difference operation returns a partial packet-space which represents packet-headers observing different forwarding behaviour between l and r.

Algorithm 4.6 BDD difference operation termination check

```
Input: $l$ The left BDD node
Input: $r$ The right BDD node
1: if $l = r$ then
2:     return $\varnothing$
3: end if
4: if $l = \varnothing$ or $r = \varnothing$ then
5:     return $(l, r)$
6: end if
7: if IsTerminal$(l)$ and IsTerminal$(r)$ then
8:     return $(l, r)$
9: end if
```

4.3 Evaluation

4.3.1 Completeness

We have shown how to convert actions to a canonical form, and how to map packet-space to these actions to create a canonical representation of a ruleset's forwarding behaviour. Our solution will not return false positives, but, in rare cases can return false negatives, i.e. equivalent rulesets may incorrectly be deemed nonequivalent. There are two causes for such false negatives 1) a set-field is redundant if the packet already contains that value, and 2) equivalences in actions we have not considered, such as those depending on the switches state. Both cases are subtle edge-cases, and pragmatically unlikely.

4.3.1.1 The Canonical Action Set Depends on the Packet

In our method described in Section 4.2 false negatives arise in rare cases because our method of converting actions to a canonical form assumes independence to the packet header. In most cases this is a correct assumption; however, a subtle edge case exists where an action sets the value of a field back to its original value in the packet header as this is equivalent to not modifying the field. As we use a minimal representation to form our canonical action representation, we should exclude a set-field action when the packet already has that field set to the same value because it is redundant.

Figure 4.5 shows three equivalent rulesets which highlight the flaw with this assumption. The equivalence of these rulesets is simple to reason about, as we only need to consider two input packet values, 1.1.1.0 and 1.1.1.1, which are both output as 1.1.1.1. If we compare actions, Figures 4.5b and 4.5c take no action on 1.1.1.1, whereas Figure 4.5a rewrites it to 1.1.1.1 and as such would incorrectly be found nonequivalent.

We next detail both an eager and lazy solution to this problem. We have implemented a prototype of both solutions and found they correctly resolve equivalences.

	Match IP Dst.	Write IP Dst.
A_1	1.1.1.0/31	1.1.1.1

(a) Always set the field to 1.1.1.1

	Match IP Dst.	Write IP Dst.
B_1	1.1.1.1/32	
B_2	1.1.1.0/31	1.1.1.1

(b) First ignore packets which are 1.1.1.1

	Match IP Dst.	Write IP Dst.
C_1	1.1.1.0/32	1.1.1.1
C_2	1.1.1.0/31	

(c) First set-fields which are not 1.1.1.1

Figure 4.5: Three equivalent rulesets which demonstrate that canonical actions can be dependent on the match, because setting a field is redundant if the field already has the same value. Rules are ordered from highest priority to lowest, and the rules will output the rewritten packets.

Eager: This approach ensures the resulting MTBDD is canonical. As we use a minimal representation to form our canonical action representation, we should exclude a set-field action when the packet already has that field set to the same value because it is redundant. In this solution, we replace rules that contain set-field actions with a sequence of rules without the set-field for packets already set to that value before building the MTBDD. The logic is to create a copy of every rule containing set-field actions with 1) a specific match on the written value set, and 2) the set-field instruction removed. This new rule is placed at a higher priority and provides a minimal representation for the actions. This process must be applied to all combinations of set-fields, making it scale with the number of unique set-fields. Figure 4.6 shows how to convert the rule in Figure 4.6a into the four priority-ordered rules in Figure 4.6b with a canonical action set for all combinations of set-field values.

A set-field action in OpenFlow 1.3 sets the entirety of that field to the value supplied, so our eager solution scales exponentially with the number of set-field actions, $2^{|setfields|}$. Anecdotally we have found this exponential scaling remains manageable as a typical ruleset applies fewer than four set-field actions to any one set of packets. However, OpenFlow 1.5 allows a programmer to set partial fields (i.e. individual bits), if we apply the same solution, then combinations need to be made per bit, which becomes infeasibly large.

Match	Set-Fields
—	EthDst:A, EthSrc:B

(a) Original rule

Match	Set-Fields
EthSrc:A, EthSrc:B	—
EthSrc:B	EthDst:A
EthDst:A	EthSrc:B
—	EthDst:A, EthSrc:B

(b) A priority-ordered ruleset, using minimal action sets for cases where the packet already contains the value set by a set-field action.

Figure 4.6: The conversion from a single rule containing multiple set-field actions into all possible combinations where those set-fields can be eliminated as the packet header already contains the value, filtered by adding it as a match. This minimises the action set for those cases allowing detection of equivalent rulesets where are actions are dependant on the matched value as shown in Figure 4.5.

Lazy: in this approach, the MTBDD is built as usual, and the comparison is modified to include additional checks lazily. The comparison first checks if the two MTBDDs are already equivalent and can break early. Otherwise, the equivalence check performs additional checks on the differing portions. This additional checking can break early at the first non-equivalent portion it finds as this means the forwarding behaviour as a whole is not equivalent. Therefore, the next portion only needs to be evaluated when redundant set-fields cause the difference.

To perform this additional check, we leverage the structure of the MTBDD by APPLYING the difference operation (§4.2.3.1) which upon finding a difference encodes a terminal that includes both actions. A single path through this MTBDD from root to terminal encodes a portion of the mismatched packet-space to the two different actions. Any action which omits a redundant set-field will always become a separate path in the MTBDD. The path is different because a ruleset must include a set-field in the actions for all other values of the field apart from the redundant value. Therefore the actions are different and will be stored in different terminal nodes. The additional check converts each path to the match it represents, and then each matched field is added as a set-field to the beginning of both rulesets' apply actions and then the actions are canonicalised as per Section 4.2.2 and compared again. Rerunning canonicalisation removes any duplicate set-fields we introduced. This comparison would be equally valid if it removed rather than added set-fields.

Compared to the eager approach the lazy approach will add near zero overhead unless rulesets differ due to redundant set-fields. This lazy approach can be applied per bit (rather than field) to deal with cases like OpenFlow 1.5 which supports bitwise set-fields, without additional overhead. Further, it avoids the expansion issue of the eager approach. However, unlike the eager approach, the lazy approach does not result in a canonical MTBDD, so other MTBDD operations may need to consider this case.

4.3.1.2 Actions can be Equivalent Depending on the Switch State

Equivalent actions can arise with the processing of special OpenFlow ports, such as a FLOOD output which is equivalent to an output action for every port on the switch. Alternatively, an OpenFlow meter or queue which a rate-limit exceeding the port bandwidth has no effect. The correctness of resolving such situations are very situational as it depends on switch state such as the number of connected ports on a switch. The validity of considering such cases equivalent is unclear and ultimately depends on the use case.

None of these issues are fundamentally unsolvable, but rather are not transformations we expect to encounter. We highlight these issues as depending on the use case they may be important.

4.3.2 Implementation

We have implemented a prototype of our ruleset equivalence checking described in §4.2.3 in Python. Internally we represent OpenFlow rules as mappings of match field to value, and lists of actions applied using the built-in dict and list types. We also convert matches to a ternary representation, Header Space [39], to allow quick intersection operations when building a single-table. While most of the code is implemented in Python, some hot spots have been converted

	Original		To Single-Table		To MTBDD		
Ruleset	No. Rules	No. Tables	No. Rules	Time	Naive Alg. Time	D&C Alg. Time	No. Nodes
Faucet Router	582	8	11,447	9.71s	1.97s	1.13s	130,287
Faucet Access	1937	8	7,216	6.77s	0.82s	0.64s	74,148
RouteViews FIB	740,332	1	740,332	0.6s	23.82s	22.66s	279,985
FIB Reversed	740,332	1	740,332	0.6s	15.09hr	425s	10,009,281

Table 4.2: Details of the rulesets evaluated, and the MTBDD build time for each. The time taken to convert the ruleset to a canonical MTBDD format is divided into the time to convert to a single-table, and time to build the MTBDD using both the Naive and Divide-and-Conquer approach.

to C. The MTBDD is implemented entirely in C with Python bindings for better performance. We used the CUDD BDD library [41] as a base for our MTBDD and added our custom rule conversion, PRIORITYADD, and difference operation logic.

4.3.3 Performance

In order to evaluate our solution, we present its performance with three different real-world rulesets. We do not evaluate the time to check equivalence of any two rulesets as by using reduced and ordered MTBDDs equivalent rulesets will have the same memory address for their root nodes [41, 26]. Instead, we measure the time to build the MTBDD representation. We perform our tests with an i7-4790 @3.6Ghz and 8GB of RAM, our implementation is single-threaded.

We evaluate the rulesets shown in Table 4.2. These consist of two captures, Faucet Router and Faucet Access, from two OpenFlow switches in a real-world enterprise deployment [42] which were programmed by the Faucet [7] controller. The Faucet controller was configured to perform VLAN switching, IPv4/6 routing, and stateless firewalling. Faucet Router has more complexity than Faucet Access as it was connected to the upstream and carries routes. Faucet Access does not carry routes, but had a larger ruleset due to having more ports, each with a stateless firewall policy applied. RouteViews FIB is based on a RouteViews [43] RIB[3], which we converted to a FIB. FIB reversed is the same as RouteViews FIB but with the bit order reversed so that the least

[3]RouteViews RIB available: `http://archive.routeviews.org/oix-route-views/2018.02/oix-full-snapshot-2018-02-13-0000.bz2`

significant bit is the lowest numbered node in the BDD. This demonstrates performance in the case where a poor node ordering is chosen, as is evidenced by the increase in the final size of BDD from 280 thousand to 10 million unique nodes.

Table 4.2 shows the time in seconds it takes to convert each OpenFlow ruleset into an MTBDD for equivalence checking and the final size of the MTBDD by counting the unique nodes. We report the conversion to an MTBDD in two parts, first converting a multi-table pipeline to an equivalent single-table (§4.2.1) and second the time to build this table into a canonical MTBDD (§4.2.2, §4.2.3). We compare results for both the Naive Algorithm 4.4 and the Divide-and-Conquer (D&C) Algorithm 4.5.

For the Faucet multi-table pipelines the conversion to a single-table is the most expensive operation, this is not surprising as this operation is implemented primarily in Python. While the conversion to a BDD is primarily performed in C. The D&C approach outperforms the Naive approach in all cases. We believe this better performance is because while both the Naive and D&C approaches perform the same number of PriorityAdd operations, the size of the BDDs added are on average smaller for D&C. The performance of our implementation is good with all rulesets except for the intentionally poorly ordered FIB reversed. Comparing FIB reversed to RouteViews FIB, the difference in build time between the Naive approach of 15 hours vs. D&C 7 minutes is very significant and shows that this technique will finish in a reasonable time even if a poor BDD node ordering is selected. This is important as it reduces the need to pick the most optimal node ordering, which is an NP-complete problem [40].

4.4 Related Work

Yang et al. [44] presented two ideas on how to compare OpenFlow ruleset equivalence as theoretical algorithms. The first, *match-field oriented approach*, considers a rule at a time from the first ruleset and successively eliminates matching rules from the second ruleset. If all rules were eliminated, then the rulesets are equivalent. However, it does not account for overlaps in rules, which when encountered will only remove the highest priority match, the remaining shadowed rule is not eliminated causing rulesets to incorrectly be deemed nonequivalent. Their second approach, *action oriented approach*, creates a canonical mapping of action to matches. Yang et al. did not consider equivalence in actions and did not consider how to represent matches as their work is theoretical. From our experience, a BDD representation would be suitable. Our implementation using an MTBDD provides a practical canonical match to action mapping, and further shows how equivalences can exist in actions and resolves these.

Chapter 5

The Rule-Fitting Problem

This chapter introduces the background and our solution to the key problem this research addresses: the rule-fitting problem. This chapter provides a high-level overview of Chapters 6 and 7, which provide in-depth detail of our solution.

The goal of the rule-fitting problem is to rewrite an existing OpenFlow ruleset to fit a constrained fixed-function OpenFlow hardware pipeline. In doing this, we improve the interoperability between OpenFlow switches for a network operator and ease the transition to new OpenFlow switches, or from software to hardware OpenFlow switches.

This chapter motivates the rule-fitting problem, defines the scope of our research, and defines the approach we took to a solution. Then, the chapter gives a high-level overview of the rule-fitting solver which we have implemented. Finally, this chapter lists related research and alternative approaches to OpenFlow device interoperability.

Our rule-fitting solver builds on the research presented thus far, including, the Table Type Pattern tools presented in Chapter 3 and the equivalence checking in Chapter 4. Our rule-fitting solver uses Table Type Patterns to describe the capabilities of a fixed-function pipeline, and equivalence checking to verify each candidate solution.

Our rule-fitting solver has two main stages: 1) finding possible transformations for rules in the input ruleset to fit the target pipeline, and 2) finding a valid combination of these rules which maintains equivalent forwarding. Chapter 6 details the first part; how the rule-fitting solver finds possible transformations for rules and the preprocessing the solver performs on the input ruleset. Chapter 7 details the second part; how the rule-fitting solver finds a valid combination of these transformations. Finally, Chapter 8 evaluates the rule-fitting solver.

5.1 Motivation

The features supported by the underlying network hardware often limit the interoperability of Software-Defined Networking (SDN) and in particular OpenFlow with different network hardware. Rather than SDN applications being write-once and deploy-anywhere, the reality is that different switch pipelines impose different limitations on the rules an application can install. These pipeline limitations arise from both hardware and software limitations of the switch. This lack of interoperability makes it hard to transition from a prototype written for an unconstrained software OpenFlow switch to a production deployment with constrained hardware switches, or transitioning between different hardware switches.

A motivating hardware pipeline for this research is the OpenFlow Data Plane Abstraction (OF-DPA) OpenFlow 1.3 pipeline released by Broadcom [45], which has strict limitations on the types of matches and actions available to rules in different tables. OF-DPA provides the OpenFlow interface to program the underlying Broadcom fixed-function switching Application-Specific Integrated Circuit (ASIC). The tables that OF-DPA exposes are specialised in the fixed-function ASIC to efficiently perform fundamental networking functions such as routing, switching, and tunnelling. This fixed-function design is cost and power-effective as it uses specialised data-structures and hardware design for each network function. The obvious downside is the loss of flexibility, meaning that an OpenFlow application cannot install an arbitrary rule anywhere in this pipeline. Instead, the developer must tailor the application's ruleset for the pipeline. Section 2.3.4 describes the OF-DPA pipeline in more detail. In the future, we expect similar fixed-function pipelines to continue to have a place in networking as they are cost and power-efficient.

One solution to this interoperability problem has been to manually write device drivers to convert rules or a higher level abstraction created by applications to new devices [46, 47]. This process is manual and therefore error-prone, and requires tools, skills, and knowledge of pipeline [46] that is often confidential to the device vendor. An alternative approach is that of converting rulesets algorithmically to target new pipelines [38, 37].

Our research explores a new algorithmic approach. While this algorithmic approach is stand-alone, it is also complementary to the device driver approach, because a developer of a device driver can use such automated fitting to suggest and verify an initial placement and then convert this into a device driver with manual optimisation.

5.2 Problem Statement

The interoperability of SDN and in particular OpenFlow is often limited by the features supported by the underlying hardware. Therefore, OpenFlow applications are developed to target particular devices, limiting their deployability in new networks with different OpenFlow hardware.

Our goal is to improve OpenFlow device interoperability by developing a general algorithmic approach to the rule-fitting problem for constrained fixed-function pipelines.

5.2.1 Rule-Fitting Solver Design Scope

Figure 5.1 shows where our rule-fitting solver integrates into an existing OpenFlow scenario. The rule-fitting solver provides a method for an existing Open-

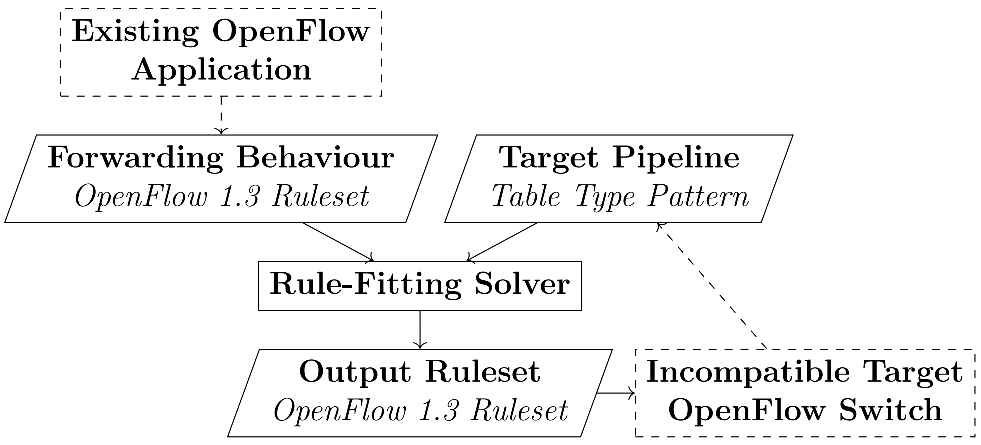


Figure 5.1: Shows how our rule-fitting solver integrates into an existing OpenFlow scenario to improve device interoperability by enabling an existing OpenFlow application to program an incompatible switch. The OpenFlow ruleset programmed by the existing application is input to the rule-fitting solver. This input ruleset provides the rule-fitting solver with a description of forwarding behaviour. The rule-fitting solver additionally takes a pipeline description of the incompatible switch in the format of a Table Table Pattern which, ideally, the device vendor provides. The output of the rule-fitting solver is an equivalent ruleset compatible with the previously incompatible switch.

Flow application to install a ruleset on an incompatible OpenFlow switch.

To describe the desired forwarding behaviour, the rule-fitting solver accepts an OpenFlow 1.3 ruleset, called the input ruleset. Using the ruleset directly as a forwarding description is practical as it saves interpreting a second forwarding description given that the output already must be an OpenFlow 1.3 ruleset. Additionally, a network operator can trivially collect the ruleset from switches on a running network without requiring the applications source code.

To describe the target pipeline, the rule-fitting solver accepts a Table Type Pattern (TTP) description. TTPs were a natural choice for describing hardware as they can express the complexity of fixed-function OpenFlow pipelines. Unfortunately, other than the OF-DPA TTP, very few vendors have released machine-readable pipeline descriptions.

The output from the rule-fitting solver is an OpenFlow 1.3 ruleset. This ruleset must be both compatible with the target pipeline and have forwarding which is equivalent to the input ruleset. The rule-fitting solver uses the research presented in the previous chapters to verify both constraints. Chapter 3 detailed how the rule-fitting solver ensures rules are compatible with the target pipeline. Chapter 4 detailed how the rule-fitting solver checks ruleset equivalence.

We chose to explore an algorithmic approach because it is more accessible than a device driver approach as it does not require cooperation from a device vendor or access to the source code of the SDN application. In addition, FlowAdapter has demonstrated that this approach could integrate seamlessly with an existing architecture as a middle layer between the OpenFlow application and switch [38].

We chose to specifically target fixed-function pipelines, as their strict constraints pose a unique challenge. Additionally, this problem has a practical application, as many OpenFlow vendors ship products which use fixed-function merchant silicon, including Edge-Core, Quanta, Allied Telesis, Pica8, Dell, and HPE [15].

We chose to develop the rule-fitting algorithm to be as general as possible, for the broadest applicability. General means the rule-fitting solver cannot rely on pipeline support for any particular OpenFlow feature, including metadata. Section 5.2.1.1 details the full list of assumptions the rule-fitting solver avoids.

5.2.1.1 A General Solver

What we mean by a general rule-fitting solver, is that the solver should make as few assumptions as possible, thus resulting in a technique applicable in most situations. Next, are the key assumptions we wanted to avoid, with the reasoning behind each.

Do not assume that OpenFlow 1.3 metadata is available in the target pipeline.

Metadata is convenient for a rule-fitting solver as it allows the solver to ignore overlaps between rules with different priorities. The solver can set and match metadata to ensure a packet only matches the intended rules, as seen in FlowAdapter and FlowConvertor [38, 37]. However, metadata is not always available in a fixed-function pipeline. For example, metadata is not available in the OF-DPA pipeline [45], which was the main motivating fixed-function pipeline for this research.

Do not assume an application will use a header-field in a traditional manner.

It is common for researchers to develop new network functions by re-purposing existing fields due to it being immediately deployable. As a concrete example, it is a mistake to assume that an application will use the Ethernet header for traditional Layer 2 forwarding. Umbrella uses the Ethernet address to encode ports along a source-routed path [48]. Portland creates positional Ethernet addresses which is much more akin to Layer 3 routing [49]. That is not to say that our solver does not need to handle traditional networking protocols efficiently. Traditional protocols such as Layer 2 forwarding and Layer 3 routing remain fundamental to networks today and in the future. So the solver must be able to find a solution to traditional networking functions, regardless of the header-fields used.

Do not hard-code the match fields and actions available to rules.

To be as general as possible, our solver should avoid hard-coding to a limited set of matches and actions. Avoiding hard-coding fields is useful as OpenFlow allows non-standard experimenter extensions with which a switch vendor can define custom header-fields and actions. The OF-DPA pipeline uses vendor extensions to support some additional fields including non-standard metadata shared between tables some tables [45].

5.3 Design Methodology

We took an incremental approach to design and implement the rule-fitting solver, starting by implementing small independent pieces and combining these to find more complex solutions. We refined the rule-fitting solver through trial and error of new ideas, typically based on observing failures to find a solution.

The solver development started as code to find a valid placement of a single unmodified rule into a target pipeline.

From here, we incrementally worked towards the goal of transforming an entire ruleset where the ruleset requires significant changes. We grew the algorithm in two main areas: 1) finding more sophisticated ways to transform a rule or combinations of rules, and 2) fitting those transformed rules together into a valid OpenFlow ruleset.

We used the OF-DPA pipeline to guide the types of transformations of rules the solver needs to support. The OF-DPA pipeline was ideal as it provided a sophisticated example of fixed-function pipeline restrictions. Therefore, we focused our time on the concrete transformations required to transform rulesets to real-world pipelines. For each type of transformation required, we created a minimal example input ruleset and table type pattern as part of the solver's test-suite. These examples allowed us to detect regressions early and are small enough to debug easily, unlike the complexity of a complete ruleset and the entire OF-DPA pipeline. Often our development was test-driven from these examples.

An early observation we made about the rule-fitting problem is that it has an intractable problem space because of the many alternative representations in OpenFlow. Consider a solver splitting a single rule between n tables in an unconstrained pipeline; the solver can split the rule between 2^n unique combinations of tables. Moreover, 2^n is before accounting for the number of unique ways the solver can split a rule's matches and actions between these tables. Additionally, if one solution is available, then practically infinite solutions are available. To illustrate, within a table only the highest-priority rule is matched so the solver could add any number of lower-priority rules with the same match without affecting forwarding as they are unreachable. The assumption of a fixed-function target pipeline helps reduce the problem size, as pipeline constraints significantly reduce the number of valid combinations. However, the problem still remains large, so we developed techniques to limit

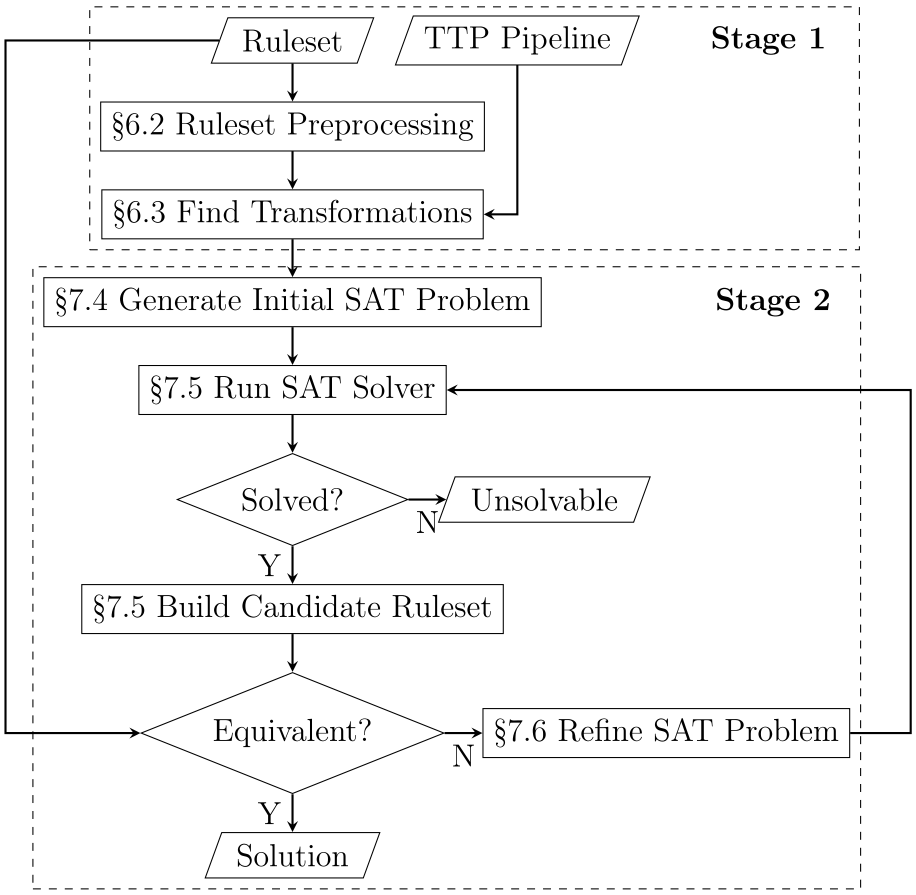


Figure 5.2: The design of the rule-fitting solver, showing the key steps involved in fitting an OpenFlow ruleset to a new pipeline. The first stage deals with preprocessing the ruleset (to simplify the problem) and transforming the rules from the input ruleset to valid placements in the TTP on a rule by rule basis. This first stage outputs a list of transformations for each rule. The task of the second stage is to pick a combination of these transformations that generate a valid solution. The second stage expresses this problem as a Boolean Satisfiability (SAT) problem. Due to the size of the problem, we found it was not possible to constrain the SAT problem to return only valid solutions. So instead, the second stage iteratively runs the SAT solver and refines the SAT problem, based on each invalid solution, until it finds a valid solution or no solution is possible.

the problem space to where reasonable solutions occur.

With such a large problem space, exploring it is often polynomial or exponential in runtime and memory usage. We used profiling to find the problematic cases, and developed heuristics to remove cases where valid solutions are unlikely. Heuristics are particularly important when building a valid ruleset to remove combinations of transformed rules which are incompatible with each other.

5.4 Overview of the Rule-Fitting Solver Design

Figure 5.2 shows an overview of the design of the rule-fitting solver. We divide the solver into two separate stages, where each stage is its own problem and poses unique challenges. The first stage generates transformations of the input rules placed in the target pipeline. The second stage finds a combination of these transformations that have equivalent forwarding to the original ruleset. Chapter 6 and Chapter 7 detail the first and second stage, respectively.

The rule-fitting solver requires two inputs: 1) a description of forwarding behaviour as an OpenFlow 1.3 ruleset [12], and 2) a description of the target pipeline as a Table Type Pattern [3].

Ruleset preprocessing is responsible for making the ruleset easier for the rest of the solver to process. Preprocessing includes removing unreachable rules and reducing the complexity of the ruleset. Removing unreachable rules ensures that every rule in the input ruleset has a purpose. The most notable preprocessing process is ruleset compression, a technique we developed to reduce the size of a ruleset dramatically. Section 6.2 details the preprocessing methods the rule-fitting uses.

From this preprocessed ruleset, the rule-fitting solver generates all possible transformations of a rule that express the same isolated forwarding behaviour in the target pipeline. A transformation maps one or more rules from the same path in the original ruleset to one or more rules in the target pipeline. We more commonly refer to a rule in the target pipeline as a placement. The rule-fitting solver uses the Table Type Pattern library described in Chapter 3 to generate these placements for the target pipeline.

A principle which guides the rule-fitting problem is that every rule in the input ruleset has a purpose. Therefore, for a solution to be valid, it must represent every rule from the input ruleset in some form. Thus the first stage of the solver generates transformations in the knowledge that the second stage will pick one for each rule in the original ruleset.

For simplicity, now consider a transformation that maps one rule to one placement. Because the next stage picks one transformation to represent each input rule, to be representative, the placement of that transformation must have the same forwarding as the original rule when considered in isolation. To check if a transformation has the same isolated forwarding, the rule-fitting solver assumes that all packets reach a placement and checks that placement applies the same forwarding as the input rule for the packet-space the input rule matches. This check carefully excludes packet-space not matched by the input rule.

The first solver stage generates a variety of different transformations, including splitting an input rule in placements spread across multiple tables and merging multiple input rules into one single placement. Section 6.3 details the transformations the solver generates and how the solver generates them.

From these transformations, the second stage must find a combination of these transformations which have the correct overall forwarding behaviour. This second stage uses the ruleset equivalence work presented in Chapter 4 to check if the candidate ruleset created from these transformations is equivalent to the input ruleset. Finding a valid combination of transformations is non-trivial as their placements often conflict with each other. Placements can conflict by shadowing each other, including conflicting actions at the same priority and directing packets to the wrong table such that other placements are not reached.

The basic constraint to guide searching combinations of these transformations is to pick exactly one transformation for each input rule. We quickly realised that naively checking all possible combinations of transformations was infeasible for any non-trivial ruleset. In our initial attempts to solve this problem, it was clear that some combinations of placements always led to invalid solutions and did warrant further consideration. So instead, we looked for a solution that would allow us to search combinations of transformations with constraints to filter out invalid combinations. We found that we could express this problem as a Boolean Satisfiability (SAT) problem, and thus could use one of many off-the-shelf SAT solvers to generate combinations of these transformations. Later, Chapter 7 gives a comprehensive introduction to the Boolean Satisfiability (SAT) problem.

The difficulty in this second stage does not lie in generating and checking these solutions, but instead in finding constraints to filter the search-space to a tractable size. Ideally, we would fully constrain the SAT problem so that it only returned equivalent solutions. However, our attempts to fully constrain the SAT problem essentially amounted to naively checking all possible combinations. So instead, we use a partially constrained SAT problem where solutions were likely to have the correct forwarding.

Based on heuristics, the rule-fitting solver generates the initial constraints for the SAT problem for known conflicting combinations of transformations and to rule out problem space where solutions are improbable. Section 7.4 details the constraints the rule-fitting solver adds to the initial SAT problem. The rule-fitting solver uses MiniSat 2 [50], an off-the-shelf SAT solver, to solve the SAT problem and return combinations of transformations. From these transformations, the rule-fitting solver generates the ruleset and checks its equivalence against the input ruleset (Section 7.5). If the ruleset is equivalent, then the SAT solver has found a valid solution to the rule-fitting problem; otherwise, another iteration is required. After each iteration, the rule-fitting solver then adds additional constraints to the SAT problem based on analysis as to why the solution failed. Section 7.6 describes these constraints the rule-fitting solver adds to refine the problem after each iteration.

5.5 Related Work

This section introduces related research in building and running SDN applications on multiple switches with different hardware pipelines. We focus on how to solve this problem for fixed-function pipelines, rather than flexible pipelines. Technologies including POF [11] and P4 [10] allow a network operator to program these flexible pipelines to support their SDN applications.

There are two main approaches taken by prior work towards solving the rule-fitting problem and OpenFlow device interoperability on fixed-function pipelines: 1) adding an abstraction layer and programming hardware-specific device drivers, and 2) algorithmically rewriting the ruleset.

5.5.1 Switch Abstraction Layers

Yu et al. [46] proposed NOSIX, a lightweight portability layer between the OpenFlow application and OpenFlow switch. In the NOSIX model, a pipeline of virtual flow tables act as an intermediary between the OpenFlow application and switch hardware. The OpenFlow application developer designs virtual flow tables suited to their application and writes their application to install flows directly into these virtual tables. The OpenFlow device vendor takes these virtual flow tables and, with their knowledge of their switching hardware, creates an efficient driver to map the virtual pipeline to their physical pipeline. In NOSIX, either a switch or the controller could run the device driver.

The application developer predefines NOSIX's virtual tables. The virtual tables cannot be changed at runtime and do not need to match the hardware pipeline. The application developer will annotate the virtual tables with additional information as *requirements* or *promises*. Requirement annotations indicate the features a virtual table uses, such as the match, action and consistency requirements. A promise annotation provides information to aid device driver development. For example, a promise annotation can indicate a rule will not exceed an amount of traffic, and as such a switch could process these packets in software.

Yu et al. [46] demonstrated the advantages of annotations providing additional information about the application with an elephant and mouse flow scenario. In which the controller has knowledge of which flows are elephants and mice. An elephant flow is a long-lived, large transfer such as a file transfer, whereas a mouse is a short transfer such as loading a web-page. Typically mice flows are more frequent than elephants and are short-lived with a high churn rate. A naive approach that does not distinguish between mouse and elephant flows will install rules for both types of flows into the hardware. Once the hardware table fills up, a switch must process both elephants and mice in software, and the elephant flows saturate the software path resulting in poor forwarding performance. However, by using the NOSIX model, Yu et al. installed elephant and mouse flows in different virtual tables, and annotated mouse flows as having a low bandwidth promise. This annotation enabled the NOSIX switch to choose to install only the elephant flows in hardware and process the low bandwidth mice in software. This flow arrangement prevented the hardware table filling up with mice, and ensured space to install elephant flows in hardware, so they did not saturate the software path.

The NOSIX model creates a very high barrier to entry, as it requires a OpenFlow switch vendor to write the driver. The switch vendor has no financial incentive or otherwise to create such a driver for all except their largest customers.

During our research, there was an ongoing project named Atrium [51] from OpenSourceSDN (OSSDN) an open-source SDN community supported by the Open Networking Foundation (ONF). Atrium created a complete turn-key solution to demonstrate SDN concepts in practice, in particular, emphasising SDN interoperability between both different network switches and different control software. Atrium created two solutions, a BGP router and a leaf-spine fabric, built as applications for both the Open Network Operating System (ONOS) and OpenDaylight (ODL) network controller platforms. As part of the Atrium effort, both ONOS and ODL implemented a high-level abstraction called flow objectives which specify forwarding through a network switch without requiring knowledge of its pipeline. There are three types of flow objectives [47]: 1) a filter objective allows traffic to be accepted or dropped, 2) a forwarding objective describes the type of traffic to forward, and 3) a next objective allows a set of actions to be applied to the forwarded traffic. Beneath flow objectives sit a series of device drivers, much like in the NOSIX model, which convert the flow objectives into rules for the hardware's pipeline. The controller developer or device vendor must write a device driver for each switch model they wish to support.

The Atrium project was deemed successful and has now completed [52]. The ONF have incorporated the ideas and code from Atrium into CORD (Central Office Re-architected as a Datacenter) [53], which continues to leverage the flow objectives built into ONOS.

Parniewicz et al. [54] built a Hardware Abstraction Layer (HAL) to address the problem of device interoperability between OpenFlow networks and non-OpenFlow devices. The HAL allows an OpenFlow application to control non-OpenFlow network devices. The HAL has a modular design, allowing a HAL developer to change or add individual modules without having to modify others. The HAL consists of two layers: 1) the Cross-Hardware Platform Layer which maintains northbound connections with the control applications, and speaks with, 2) the Hardware Specific Layer which implements device-specific logic to translate to and from hardware.

The Cross-Hardware Platform Layer maintains endpoints for different protocol versions of OpenFlow and enables network management with NETCONF and an interface for virtualisation. This layer also maintains an abstract OpenFlow pipeline which processes packets that are not supported by the hardware and packet in and out messages. The Hardware Specific Layer contains logic to convert and install flow rules into hardware as well as handling device discovery and orchestration. This layer is the device driver, and a programmer needs to write this conversion layer. Parniewicz et al. implemented HAL for devices such as NetFPGA, EZchip NP-3 NPUs, traditional CPUs, Data Over Cable Service Interface Specification devices, and, with extensions to OpenFlow, Reconfigurable Optical Add/Drop Multiplexers.

5.5.2 Rewriting Rulesets Algorithmically

Pan et al. [38] took a different approach to this same problem. Pan et al. [38] developed an algorithm, FlowAdapter, to take an existing OpenFlow ruleset and fit it to a switch pipeline in which tables can match a limited subset of header-fields, which is a constraint of fixed-pipeline commodity hardware.

First, FlowAdapter converts a multi-table ruleset to a single table with the use of a specialised tree the authors named an N-tree. Then FlowAdapter converts the resulting single table to the target OpenFlow switch's N-stage pipeline; named the OTN (One-stage to N-stage) conversion. The OTN conversion assigns each match field of a rule into a flow table supporting that match field, and then links these rules together using goto and metadata instructions; thus forming an equivalent path through the target pipeline which contains all original match fields.

Pan et al. [38] measured the performance of FlowAdapter to take 110 microseconds to fit a set of 1000 rules. This performance showed this approach suitable to run in real-time. The authors identified areas for further research, including the optimisation of the N-tree. Their implementation would completely rebuild the entire tree for each rule update, rather than computing an incremental difference. Support for the offloading of rules that a pipeline cannot match to a software path, and optimisation so that a switch could run this conversion.

During our research, Pan et al. [37] presented a continuation of FlowAdapter named FlowConvertor. Compared to FlowAdapter, FlowConvertor considers pipelines with constraints on actions, in addition to match constraints. FlowConvertor makes use of incremental algorithms to maintain and update the input pipeline representation by intercepting OpenFlow flow add, update and delete messages in real-time. FlowConvertor maintains the cross-product of all flow tables in an efficient filtered Directed Acyclic Graph (DAG) structure in which nodes represent rules and edges represent paths through the pipeline. FlowConvertor maps each path through the DAG to the target pipeline by filling the target pipeline with matches from left to right. If the original path still contains matches the mapping has failed. Next, FlowConvertor fills in values for any remaining empty match fields that the target pipeline requires with the last value set or matched in that field. FlowConvertor generally places actions in the final rule. Critical action-match pairs, which require a specific ordering (e.g. set field, match field), are an exception; FlowConverter places the critical action on the rule before the associated match. Before installing these rules to the switch, FlowAdapter searches for existing equivalent rules (which it uses instead) and backtracks through the path to link rules together using either metadata or redundant operation, such as a push tag, set tag, match tag, and pop tag sequence.

In their evaluation, Pan et al. found FlowAdapter performed well. Pan et al. evaluated three synthetic rulesets which they constructed to fit three different processing models through the OF-DPA pipeline. They evaluated FlowAdapter fitting these rulesets into three different target pipelines, one commodity hardware switch and two synthetic software switch pipelines. The total latency added to rule installation was typically between 1 and 2ms, with computation taking up a small portion of that, in the order of 10μs, with the remaining time being communication overhead. FlowAdapter is sensitive to the original ruleset as it tries to fit flows from the origin pipeline to the target with as little rewriting as possible. It also relies heavily on metadata to map together flows in the final pipeline, which is not available on all switches.

Sun et al. [55] conducted similar work in the context of a multi-tenant virtualised network environment. In a multi-tenant virtualised network environment, any tenant can configure their network in isolation from others while sharing the same physical hardware. Their research targeted modern network hardware which supported multiple unconstrained tables, which included the ability to arbitrarily forward packets to earlier tables and the same table. Their hardware target is contradictory to our target of supporting constrained fixed-function hardware, but their technique introduced some interesting techniques.

Sun et al. [55] presented a technique to convert a large virtual flow table pipeline into a hardware pipeline. Each tenant's virtual flow table pipeline is unconstrained, independent and can exceed the number of hardware tables. Their technique maps these virtual software tables to hardware tables. The mapping assigns each virtual flow table a segment ID and installs these segments into hardware tables. The mapping stores the segment ID in metadata and adds this match to each rule, thus allowing a hardware table to contain multiple segments. To traverse between virtual flow tables, the rules installed in hardware write the next segment ID to metadata and forward the packet to the associated hardware table. If a virtual table, i.e. segment, is too large to fit in one hardware table, it is split into more segments and spread across multiple hardware flow tables by sending the unmatched packets of the first segment to the second segment. All rules in the first segment must be of a higher priority than the second, and both segments retain the same ID. By allowing arbitrary traversal between tables, combined with splitting segments, this technique can map any number of virtual table pipelines into hardware tables until it has filled all hardware tables.

Sun et al. highlighted a performance issue stemming from the fact that a single table in the hardware pipeline could include rules related to any virtualised table, in the worst case a rule from every virtual table. As such a hardware table could be attempting to simultaneously match a packet from each virtual table, which could exhaust the bandwidth of the hardware table. The authors suggested that once the full pipeline matches a network flow, the switch could install a single more specific rule for that network flow in the first table; thus all further packets for the flow would only hit the first table.

Jose et al. [56] investigated compiling a logical pipeline to a reconfigurable hardware pipeline. While they do not tackle the problem we are attempting to solve, there are similarities; recall that we are translating rulesets to new pipelines. A logical pipeline is defined using a language such as P4 [10] or POF [11] and describes a match-action pipeline. It is the compiler's job to take these logical tables and map them to a reconfigurable pipeline. While a reconfigurable hardware pipeline supports arbitrary matching, it has limitations including the total number of tables, the number of rules per table and the width of matching available in a single table.

Jose et al. [56] calculated dependencies within the logical pipeline and built a table dependency graph, which they used to identify tables which a switch can execute in parallel. They compared an ILP and greedy algorithm's ability to find an optimal solution given the constraints imposed by table dependencies and the limitations of the hardware pipeline. The optimisations considered were to: minimise the number of pipeline stages, the latency of the pipeline, and the power consumption. Jose et al. found that Integer Linear Programming (ILP) gave more optimal solutions but had a longer runtime compared to the greedy algorithms. ILP typically had a runtime in the hundreds of seconds, whereas greedy algorithms took fractions of a second, but, ILP can outperform the optimality of a greedy approach by up to 25%.

5.5.3 Summary

We have outlined research including NOSIX, Atrium, FlowAdapter and FlowConvertor, which show a variety of different approaches to the problem of running an SDN application on a different pipeline. NOSIX used a pipeline of Virtual Flow Tables defined by the application and required the device vendor to implement a device driver translation layer. Whereas, Atrium abstracted away from OpenFlow by using Flow Objectives. Both NOSIX and Atrium use a device driver to convert the input to OpenFlow rules that fit the switch's pipeline. This approach requires a programmer to write a device driver for every new type of device. FlowAdapter and FlowConvertor showed it is possible to develop an algorithm to take OpenFlow input ruleset and convert it to fit an arbitrary fixed-function hardware pipeline.

Chapter 6

Transforming Rules and Preprocessing Rulesets

This chapter details the first stage of the rule-fitting solver, transforming OpenFlow rules. Transforming rules includes 1) preprocessing the input ruleset to reduce the size of the problem, and 2) transforming rules in isolation into equivalent placements (rules) in the target pipeline. The placements found in this stage of the solver are input to the second stage, which finds a valid combination of these placements that do not conflict with each other. Chapter 7 details the second stage of the solver.

Before discussing ruleset preprocessing and rule transformations, first, Section 6.1 introduces the idea of dependencies between rules in a ruleset and provides algorithms to calculate these dependencies. A dependency exists between two rules if removing one rule would potentially change the set of packets that reach the other rule, and likely the overall forwarding of the ruleset. These dependencies provide an intuitive method of reasoning about forwarding, which we visualise in figures. Dependencies also play a vital role in our ruleset compression technique, presented in Section 6.2.3.

6.1 Dependencies Between Rules and Paths

A path is a sequence of rules through an OpenFlow pipeline, each connected by a goto table instruction. The path a packet takes through a pipeline determines its forwarding. For a given packet, the rules in its path are the highest-priority matching rule in each table it visits, starting from the first table and following goto instructions. Within a ruleset some paths will be unreachable by any packet, we call this an invalid path. A path is valid if at least one packet can follow the the path in its entirety. Unless stated otherwise, we generally refer to valid paths simply as paths.

A rule is dependent on another rule if removing the other rule would potentially change the packets reaching this rule. Removing or adding rules to a ruleset will change these dependencies and therefore, in most cases, the valid paths through the ruleset and its overall forwarding. There are two types of dependencies:

Shadow dependencies: Occur between rules with different priorities in the same table with overlapping matches. The higher priority rule shadows the lower priority rule, stopping the overlapping portion of packets from reaching the lower priority rule.

Inter-table dependencies: Occur between rules in different tables where one sends to the other, using the goto instruction. Inter-table dependencies form paths through the ruleset which result in the actions of both rules being applied to packets on this path.

Further, we split these dependencies into direct and indirect variations of each. Direct dependencies are more succinct and natural to visualise and are sufficient for most calculations, such as calculating valid transformations (§6.3). Section 6.2.3 uses indirect dependencies when compressing a ruleset, to ensure calculations do not inadvertently introduce or remove dependencies between rules.

We base calculating these dependencies on the foundations laid by Katta et al. in CacheFlow [57]. CacheFlow computes direct shadow dependencies to

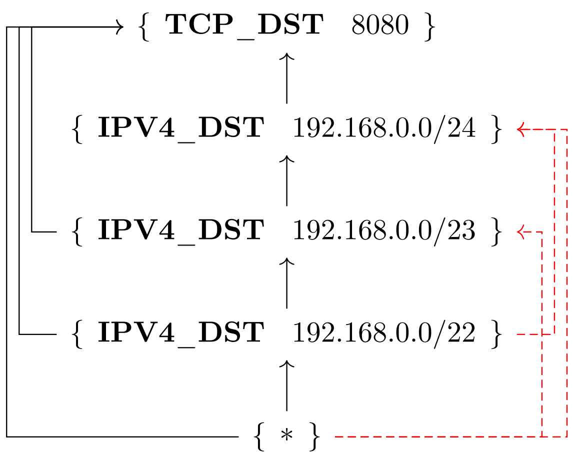

{ TCP_DST 8080 }
{ IPV4_DST 192.168.0.0/24 }
{ IPV4_DST 192.168.0.0/23 }
{ IPV4_DST 192.168.0.0/22 }
{ * }

Figure 6.1: An example showing the shadow dependencies between a set of rules in the same table. Rules are listed from highest to lowest priority top to bottom, { * } is a table-miss rule and matches all packets. An arrow points from a rule to the rule it depends on to filter (shadow) the packets that reach it. The solid black arrows show direct shadow dependencies. The dashed red lines show indirect shadow dependencies. The instructions and actions of a rule has no influence on shadow dependencies.

create a directed acyclic dependency graph to find dependency chains that can be moved in their entirety to a faster cache switch while maintaining the correct forwarding. We extend Katta et al.'s work with inter-table dependencies and indirect dependencies.

Consider dependencies between shadowed rules. A lower-priority rule is dependent on a higher-priority rule when that rule stops packets from reaching the lower-priority rule. Therefore, when the intersection of both rules' matches is not empty. Figure 6.1 gives an example of shadow dependencies between rules in the same table. Direct dependencies are shown as black solid lines and indirect dependencies as red dashed lines.

To illustrate the difference between direct and indirect dependencies consider the /22, /23 and /24 rules. The /22 does not hold a direct dependency with the /24 because all packets matched by the /24 are also matched by the /23 and therefore removing the /24 will not change the packets which reach the /22. However, the /22 does hold an indirect dependency with the /24, as if the /23 was removed then the /24 would stop packets reaching the /22.

Algorithm 6.1 Calculate a rule's direct and indirect shadow dependencies

```
Input: rule The rule to compute the shadow dependencies of
Input: higher_priority A list of higher priority rules from the same table in
    priority ascending order.
function DIRECT_SHADOW(rule, higher_priority)
    remaining_match ← rule.match
    direct ← a set
    for all $r \in$ higher_priority do
        if $r.match \cap remaining\_match \neq \emptyset$ then
            direct.add(r)
            remaining_match ← remaining_match − r.match
        end if
    end for
    return direct
end function
function INDIRECT_SHADOW(rule, higher_priority)
    indirect ← a set
    for all $r \in$ higher_priority do
        if $r.match \cap rule.match \neq \emptyset$ then
            indirect.add(r)
        end if
    end for
    return indirect
end function
```

All rules are directly dependant on TCP_DST 8080 because it stops a portion of packets from reaching them all. Consider the /22, the /23 directly stops packets with IPV4_DST 192.168.0.0/23 from reaching the /22. TCP_DST 8080 directly stops the remaining packets (not already stopped by the /23) with the header values IPV4_DST 192.168.**2**.0/23 and TCP_DST 8080 from reaching the /23.

Algorithm 6.1 shows the calculation of for direct and indirect shadow dependencies for a given rule. The direct shadow dependencies for a rule are calculated by taking the packet set representing the rule's match and subtracting the intersection of each higher priority rule's match in ascending priority order. A dependency exists for any rule with a non-empty intersection so it is added to the set of dependencies *direct*.

Whereas, indirect dependencies are calculated as all higher priority rules with a non-empty match intersection, without subtracting this intersection between rules. By this definition indirect dependencies include all direct de-

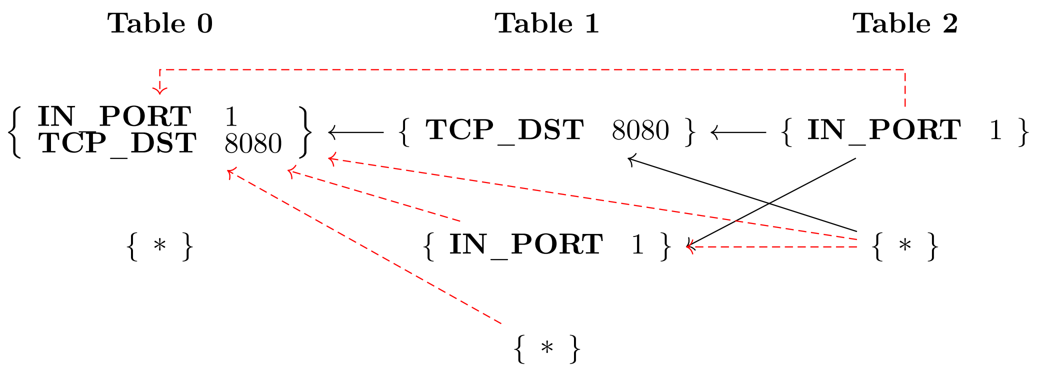


Figure 6.2: An example showing the inter-table dependencies between a set of rules. Each rule is shown by its match, and all rules send packets to the next table. Except for the table-miss ({ * }) rules which drop packets. An arrow points from a rule to the rule it depends on to receive packets. The solid black arrows show direct inter-table dependencies. The dashed red lines show indirect inter-table dependencies. Note that the second rule in table 1 is unreachable as it has no direct dependencies to earlier tables.

pendencies which is required by calculations. However, in figures, for clarity, we have only drawn the additional indirect dependencies.

Consider dependencies between rules in different tables. A rule in a later table is dependent on a rule in an earlier table if it directs packets to that later rule. The earlier rule uses a goto table instruction to direct packets, and the later rule must match the packets sent to it. Therefore, the packets sent to the later rule (after any modifications applied) must have a non-empty intersection with the later rule's match. Figure 6.2 gives an example of inter-table dependencies between rules in different table. Direct dependencies are shown as black solid lines and indirect dependencies as red dashed lines.

Algorithm 6.2 shows how to calculate direct dependencies. Line 6 considers each rule and if the rule includes a goto table instruction, then rules in that next table will be dependent on it, otherwise no rules are dependent on it. Line 5 calculates the *egress_packets* of the rule, this is the packet-space of the rule's match combined with the rule's apply actions using Algorithm 4.2 from our equivalence checking work. Lines 9-14, find the highest priority rules in the next table which will match these egress packets, subtracting the intersection each time in the same manner as DIRECT_SHADOW in Algorithm 6.1.

Algorithm 6.2 Calculate a ruleset's direct inter-table dependencies

```
Input: $ruleset[table]$ The ruleset, indexable by table e.g. table[0] returns rules
    in the first table.
Output: $dependencies$ A set of direct dependencies, each dependency is a
    pair of rules
 1: dependencies ← An empty set
 2: packets ← A packet-space containing all packets
 3: for all $rule \in ruleset$ do
 4:     next_table ← rule.instructions.goto_table
 5:     egress_packets ← rule.calculate_egress($packets$)
 6:     if next_table is NULL then
 7:         continue
 8:     end if
 9:     for all $nrule \in ruleset[next\_table]$ do    ▷ in decreasing priority order
10:         if $nrule.match \cap egress\_packets \neq \emptyset$ then
11:             dependencies.add((nrule, rule))
12:             $egress\_packets \leftarrow egress\_packets - nrule.match$
13:         end if
14:     end for
15: end for
```

This calculation does not consider the ingress to the rule, a rule might be unreachable yet still hold a dependency with a rule in the next table.

For example, in Figure 6.2, consider the in-port rule in Table 1. The rule holds no direct inter-table dependencies with earlier tables, so no rules send it traffic and it is unreachable. However, if packets did reach the rule, it would send packets to the in-port rule in Table 2 so that rule is a dependent. And because the higher in-port rule in Table 2 matches the entire egress of the same rule in Table 1, these packets cannot reach the Table 2 table-miss rule so there is no dependency. The table-miss rule holds a direct shadow dependency (not shown) with the in port rule in Table 2, therefore encapsulating the information that if the in-port rule in Table 2 were removed then the table-miss would receive its traffic.

Algorithm 6.3 shows how to calculate indirect inter-table dependencies. Indirect dependencies include shadowed rules that would receive packets if the shadowing rules were removed and rules that send packets to a rule indirectly via one or more other rules and therefore tables.

The main algorithm INTERTABLE_INDIRECT_REC is recursive and walks

Algorithm 6.3 Calculate a ruleset's indirect inter-table dependencies

```
Input: ruleset[table] The ruleset, indexable by table e.g. table[0] returns rules
    in the first table.
Output: dependencies A set of indirect dependencies, each dependency is a
    pair of rules
 1: dependencies ← An empty set
 2: packets ← A packet-space containing all packets
 3: path ← An empty list
 4: for all table ∈ ruleset do
 5:     INTERTABLE_INDIRECT_REC(table, packets, path)
 6: end for
 7: function INTERTABLE_INDIRECT_REC(table, packets, path)
 8:     for all rule ∈ ruleset[table] do
 9:         next_table ← rule.instruction.goto_table
10:         egress_pkts ← rule.calculate_egress(packets)
11:         if egress_pkts = ∅ then
12:             continue
13:         end if
14:         for all path_rule ∈ path do
15:             dependencies.add((rule, path_rule))
16:         end for
17:         if next_table then
18:             new_path ← path + [rule]
19:             INTERTABLE_INDIRECT_REC(next_table, egress_pkts,
    new_path)
20:         end if
21:     end for
22: end function
```

all valid paths through the ruleset from a given table, and adds dependencies between the rules in these paths. Lines 4-6, start this recursive algorithm from each table, with all packets as input to that table and an empty path. Starting from each table ensures that unreachable rules are considered, so that calculations do not introduce or remove incorrect dependencies with these rules.

INTERTABLE_INDIRECT_REC checks if each rule in the table matches the packets following the path so far. Line 10, calculates the egress from a rule given *packets* as input, *calculate_egress* considers the intersection of the rule's match and the ingress packets and then applies any modifications from the rule's apply actions to return the packets which egress the rule. *Calculate_egress* will return $\emptyset$ if the rule matches none of the input packets, indicating this rule is not a valid extension to the path. For any rule on the path, lines 14-16 adds a dependency with all preceding rules. Then lines 17-19 extend the path by recursively calling INTERTABLE_INDIRECT_REC with the next table the current rule gotos.

Note that inter-table dependencies are closely related to the single-table conversion described in Section 4.2.1, the valid paths that create these dependencies are the same paths which get merged into a single rule in the single-table conversion.

6.2 Ruleset Preprocessing

Prior to finding transformations for rules, the solver can optionally preprocess the ruleset to simplify the problem. This section details techniques that we explored, including converting the ruleset to a single table and compressing the ruleset to remove the redundancy. We evaluate the performance of these preprocessing techniques in relation to the overall solver later in Chapter 8.

6.2.1 Conversion to a Single-Table

If the solver first converts the ruleset to a single-table; i.e. by MERGING all rules as Section 4.2.1 describes, then merge transformations are unneeded. This simplifies the problem for the solver as it need only calculate split transformations. As equivalence checking already requires conversion to a single-table, there is no additional performance hit other for this conversion.

Converting to a single-table can result in transformations that the solver otherwise would not explore as it does not try a combination of split and merge transformations. For a given network function, we expect a developer to split rules between tables in their application efficiently in a similar manner to the designer of a fixed-function pipeline. The solver loses the information about sensible places to split rules in the conversion to a single-table. Additionally, another downside is that a single-table ruleset has more rules than the ori-

Table	**ID**	**Pri.**	**Match**	**Apply Act.**	**Goto**
0	R_A	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 1\\ \textbf{IN\_PORT} & 1\end{array}\right\}$	[]	1
	R_B	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 1\\ \textbf{IN\_PORT} & 2\end{array}\right\}$	[]	1
	R_C	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 2\\ \textbf{IN\_PORT} & 3\end{array}\right\}$	[]	1
	R_D	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 3\\ \textbf{IN\_PORT} & 4\end{array}\right\}$	[]	1
	R_E	0	{}	[]	
1	R_F	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 1\\ \textbf{ETH\_DST} & AA:...\end{array}\right\}$	[**OUTPUT** 1]	
	R_G	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 1\\ \textbf{ETH\_DST} & AB:...\end{array}\right\}$	[**OUTPUT** 2]	
	R_H	10	$\left\{\begin{array}{ll}\textbf{VLAN} & 2\\ \textbf{ETH\_DST} & AC:...\end{array}\right\}$	[**OUTPUT** 3]	
	R_I	0	{}	[**OUTPUT** *Cntr*]	

Figure 6.3: A simplified ruleset which performs per VLAN forwarding. For brevity we exclude the Ethernet source table. VLAN 1 has three learnt hosts, VLAN 2 one host, and VLAN 3 has not learnt any hosts.

ginal multi-table ruleset. However, in practice, we find ruleset compression significantly reduces number of rules and alleviates this issue.

6.2.2 Removing Unreachable Rules

Our rule-fitting solver makes the assumption that every rule in the input ruleset needs to be represented somewhere in the solution ruleset. However, this is an incorrect assumption if a rule is unreachable, as unreachable rules do not influence forwarding. The conversion to a single-table can result in unreachable rules with actions with bogus action combinations which the solver cannot place.

To avoid this situation the solver removes unreachable rules from the input ruleset. The solver finds and removes all fully shadowed rules, through analysis of a ruleset's shadow dependencies.

6.2.3 Ruleset Compression

Consider the ruleset shown in Figure 6.3. There are clear groupings of rules which all serve the same purpose. The rules R_A through to R_D filter ports based on the VLAN tag and R_F through to R_H apply forwarding per VLAN.

Intuitively, one would expect that if the rule-fitting solver found a valid transformation for R_A, it would also be a valid transformation for R_B, R_C, and R_D. Throughout this section, we refer to rules which serve the same purpose as similar rules. Section 6.2.3.1 refines the definition of similar rules in the context of our rule-fitting solver.

From this intuition, it appears the solver is doing more work than it needs to when solving the rule-fitting problem for the entire ruleset. To minimise the problem size and speed up rule fitting, we investigated compressing copies of similar rules from a ruleset. We base this compression on the observation that any non-trivial SDN application will have code paths which are called multiple times to generate similar rules with the same purpose. A controller executes such code paths in response to events such as a learning a host, a port coming up, or learning a route.

All rules that a given code path generates will be similar. In the majority of cases, a code path will generate a rule which is installed in the same table at the same priority with the same matches and actions; however, the rule will vary on the specific value matched and the value of an action. For example, a code path responsible for learning a host will install a rule which varies only by the specific host matched and the port of the output action. These similar rules conform closely to the typical restrictions of hardware pipelines. Almost all hardware pipeline limitations place restrictions only on the matches and actions available, rather than the specific values of the matches or actions. So it is reasonable to expect that a rule-fitting solver can transform similar rules — those with similar matches, actions, table and priority — in the same way.

Beyond reducing the ruleset size and improving performance, compressing the ruleset also results in a natural solution more akin to a handcrafted solution. The solution is more natural because similar rules from the same code path are likely to become one rule in the compressed ruleset and therefore when applied back to the original ruleset, placed in the same location. Without this compression, there is nothing to stop the solver from installing similar rules in arbitrary locations, which is confusing and unreadable. This more natural solution is much easier for a network operator to understand and for a software developer to implement in code for a new or updated SDN application.

6.2.3.1 What is a Similar Rule?

Our compression algorithm groups similar rules together and then, carefully selects one representative rule from each group to create a compressed ruleset. This compressed ruleset is input to the rule-fitting solver and must be representative of the complete ruleset in terms of forwarding complexity and the types of rules required in the pipeline. Additionally, the solver must be able to map a compressed ruleset back to the complete ruleset.

We developed heuristics to group similar rules, such that the compressed ruleset maintains the representative information the rule-fitting solver requires. The requirements of a group of similar rules are as follows:

1. **The transformation a solver finds to place any rule in a group must be able to be applied to all other rules in the group**: If any rule within a group cannot have the same transformation applied, then any solution found is unusable. For example, consider a rule which the solver splits between table 1 and table 2. If another rule in the same group matched an additional header field, the solver does not know where to place this match field and therefore, fails to generate a solution. Arbitrarily selecting a table does not work, as the original transformation did not consider how this additional field effects overall forwarding or whether this field is allowed by the pipeline.

2. **Once transformed, a group of rules must conform to the requirements of the target pipeline**: If a transformed rule does not meet the pipeline's requirements, the solution is invalid. It is infeasible to use the actual pipeline requirements, as these vary depending on where the solver places a rule. For example, it might be possible to place two

different rules in the same table, and yet only one of these rules meets the requirements for placement in another table.

3. **Once transformed, rules within the same group must represent the same forwarding in relation to other rules**: The rule-fitting solver will apply the transformation it selects for one rule in the compressed ruleset to all rules from the same group in the original ruleset. The resulting ruleset must have the same forwarding as the original ruleset.

   Dependencies between rules like a rule shadowed by a higher priority rule and a rule directing traffic to another rule (using goto) are fundamental to determining the forwarding behaviour of a ruleset. The compressed ruleset must include all dependencies between rules; otherwise, it would represent fundamentally different forwarding behaviour. Because compression selects one rule from each group, these groups need to have similar dependencies to ensure the compressed ruleset retains the same dependencies.

4. **Fewest groups possible**: As each group becomes one rule in the compressed ruleset, minimising the number of groups minimises the final ruleset's size.

To find sensible groups, the heuristics utilise existing information in the ruleset. It is typical for a controller to generate rules to perform a given network function from the same code path and install these similar rules at the same priority and in the same table. For any target pipeline which supports this network function, the solver should install all similar rules in the same location.

To group rules by the requirements listed above, under the assumption that the original ruleset has rules placed in reasonable locations, the compression algorithm groups rules which have all of the following:

1. **The same table and priority:** Rules at the same priority in the same table likely came from the same code path and handled the same network function.

2. **The same match mask:** Typically, a hardware pipeline will limit the match fields available and their maskability but not their values within a table. As such, requiring the same match mask (i.e. the same bits matched in the packet header) within a group almost always ensures all rules within that group can be placed in the same location.

3. **The same action types:** Following the same reasoning as the match mask above, a hardware pipeline typically limits the actions available within a table, but not their values. So rules in the same group must have the same action types. For example, a rule with the action output:1 is grouped with a rule with the action output:2, because both actions are of the output type. However, neither can be grouped with a rule that both sets the Ethernet source and outputs the packet.

4. **The same dependencies**: Each rule within the same group needs to have the same inter-group dependencies. If a rule in G_1 has a dependency with one or more rules in G_2, then every other rule in G_1 must also have a dependency on at least one rule (which can be different) in G_2. In this way, selecting one rule from each group can represent all of the original dependencies.

In this section, we will continue to use the word 'similar' in the context of a rule or rule dependencies as listed above. Otherwise, in general usage, to mean that in most cases we expect to be able to generalise successfully fitting one instance to all other similar instances with the correct behaviour. This compression technique remains a heuristic and counter-examples exist where generalising back to the original ruleset fails.

6.2.3.2 Compression Algorithm

This section outlines the compression algorithm, which is shown by example in Section 6.2.3.3.

The algorithm first uses heuristics to create a coarse set of groups, which are refined by considering dependencies between rules. Last, from each group, a representative rule is selected to create a compressed ruleset that is suitable input for the solver.

The algorithm follows:

1. Create the initial coarse groups of rules from those with the same priority, table, match mask, and action types.
2. Precompute the shadow and indirect inter-table dependencies between all rules, using the technique described in Section 6.1.
3. Pick a group
   (a) For each rule in that group, calculate its inter-group dependencies. An inter-group dependency exists between a rule and another group if the rule holds a dependency with one or more of that group's rules.
   (b) If the inter-group dependencies for all rules within that group are identical, select the next group and repeat Step 3a until all groups have been considered. If all groups have been considered without modifications, move onto Step 4. Otherwise, if the dependencies differ, split the existing group into new groups of rules containing the same inter-group dependencies and restart the process at Step 3.
4. Walk the groups from the lowest priority group in the last table to the highest priority group in the first table. All rules within a group have the same priority and table as per Step 1.
   (a) Select a rule from the first group; this should have only one rule, the default table-miss.

(b) Then for all subsequent groups, pick a rule from the group such that all of its inter-group dependencies with groups already considered are met by the rules selected so far. For example, if a rule matching VLAN 1 is chosen from one group, then all other groups in the same dependency chain with a VLAN match will have their VLAN 1 variant selected to maintain the dependency between these groups unless the VLAN is rewritten between rules.

The resulting compressed ruleset is suitable for input into the solver as it maintains dependencies between rules and therefore represents all paths that packets can take through the original ruleset. Any solution to the compressed ruleset can also be trivially generalised back to the original ruleset.

We can be confident that this process of splitting groups, Step 3, will complete as each loop either splits a group into smaller groups or completes once the inter-group dependencies within a group are the same. Once all groups contain only one rule they cannot be split further, and the process completes; in this case, compression was not possible.

In Step 4, the order that the compression algorithm considers groups in is important, because the choice of rule from one group will force the choice of a rule in a dependent group. Consider groups with the inter-group dependencies $G_A \leftarrow G_B \leftarrow G_C$ where all rules within each group include an exact VLAN tag match. For these dependencies to exist, G_B must be either in a later table than G_A or in the same table, but at a lower priority. In the same way, G_C must come after G_B. Consider processing these groups in the order G_C, G_A, G_B and selecting a rule matching VLAN 1 from G_C and VLAN 2 from G_A. Then it is impossible to select a rule from G_B that both matches VLAN 1 and VLAN 2. However, when processing in the correct order (G_C, G_B, then finally G_A), the choice from G_C forces a rule with the same VLAN to be picked from G_B and subsequently G_A.

Table	ID	Pri.	Match	Apply Act.	Goto
0	R_J	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 1 \\ \textbf{IN\_PORT} & 1 \end{array} \right\}$	[]	1
	R_K	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 1 \\ \textbf{IN\_PORT} & 2 \end{array} \right\}$	[]	1
	R_L	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 2 \\ \textbf{IN\_PORT} & 3 \end{array} \right\}$	[]	1
	R_M	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 3 \\ \textbf{IN\_PORT} & 4 \end{array} \right\}$	[]	1
	R_N	0	{}	[]	
1	R_O	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 1 \\ \textbf{ETH\_DST} & AA:... \end{array} \right\}$	[**OUTPUT** 1]	
	R_P	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 1 \\ \textbf{ETH\_DST} & AB:... \end{array} \right\}$	[**OUTPUT** 2]	
	R_Q	10	$\left\{ \begin{array}{ll} \textbf{VLAN} & 2 \\ \textbf{ETH\_DST} & AC:... \end{array} \right\}$	[**OUTPUT** 3]	
	R_R	0	{}	[**OUTPUT** *Cntr*]	

Figure 6.3 *(Repeated)*: A simplified ruleset which performs per VLAN forwarding. For brevity we exclude the Ethernet source table. VLAN 1 has three learnt hosts, VLAN 2 one host, and VLAN 3 has not learnt any hosts.

6.2.3.3 Compression by Example

This section better illustrates the compression process by showing how to compress the ruleset listed in Figure 6.3. The first step of compression is to create an initial coarse set of groups before considering indirect dependencies. This grouping aims to ensure that the solver can place all rules together in the target pipeline. The first step is to group rules together with identical table, priority, match mask, and action types. For the ruleset in Figure 6.3 this results in the following four groups:

- $\mathbf{G_1} = \{R_A, R_B, R_C, R_D\}$ — Per port VLAN filtering
- $\mathbf{G_2} = \{R_E\}$ — Table 0 default
- $\mathbf{G_3} = \{R_F, R_G, R_H\}$ — Learnt VLAN hosts
- $\mathbf{G_4} = \{R_I\}$ — Table 1 default

The next step is to calculate the indirect dependencies between rules. Figure 6.4 shows this initial coarse grouping of rules and additionally, the dependencies between rules. The compression process does not distinguish between

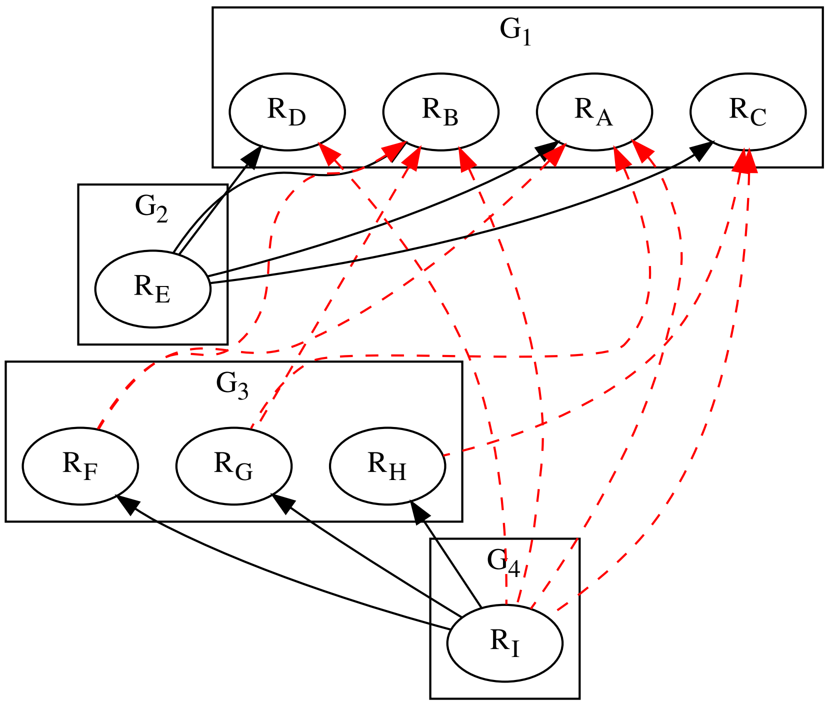


Figure 6.4: A dependency graph of the ruleset in Figure 6.3 after the coarse initial grouping of rules by table, priority, action, and match mask. Solid black edges represent shadow dependencies, while dashed red edges represent inter-table dependencies.

shadow and inter-table dependencies, but this separation is a helpful aid to visualise the process.

In the next step, these groups are split up to ensure that the inter-group dependencies are the same for all rules within the same group. First, let us consider the dependencies of rules within G_1. If considered naively, per-rule the dependencies are as follows:

- $\mathbf{R_{A_{deps}}} = \mathbf{R_{B_{deps}}} = \{R_E, R_F, R_G, R_I\}$
- $\mathbf{R_{C_{deps}}} = \{R_E, R_H, R_I\}$
- $\mathbf{R_{D_{deps}}} = \{R_E, R_I\}$

R_A and R_B both have the same dependencies, yet R_C and R_D both have different dependencies and would thus be put in separate groups. Separating R_C from R_A and R_B is overzealous because the only difference in dependencies comes from dependencies held with a different combination of rules within G_3.

As all groups contain similar rules (same match mask, action types, table, and priority), all dependencies between a rule in G_1 and G_3 are similar enough that fitting one dependency pair should generalise to all other rules. In the case of R_D, it does not hold any dependency with any rule in G_3, so does not share similar dependencies.

So instead of considering dependencies between rules, we consider the inter-group dependencies of a rule. For each rule within G_1, we find the other groups it holds a dependency with, that is to say, where a rule has a dependency with one or more rules in a group.

- $\mathbf{R_{A_{deps}}} = \mathbf{R_{B_{deps}}} = \mathbf{R_{C_{deps}}} = \{G_2, G_3, G_4\}$
- $\mathbf{R_{D_{deps}}} = \{G_2, G_4\}$

As R_D does not hold a dependency with G_3 it is moved into a separate group to R_A, R_B, and R_C. Notice that to be considered similar the number of dependencies between a rule and a group is irrelevant; R_A and R_B both hold dependencies with two rules in G_3 and R_C with only one rule in G_3.

After splitting G_1 we end up with the following five groups $\mathbf{G_1} = \{R_A, R_B, R_C\}$, $\mathbf{G_2} = \{R_D\}$, $\mathbf{G_3} = \{R_E\}$, $\mathbf{G_4} = \{R_G, R_F, R_H\}$, and $\mathbf{G_5} = \{R_I\}$ as shown in Figure 6.5.

Because splitting a group will change the inter-groups dependencies of rules within other groups, we must restart the process and recheck the dependencies of all groups. On rechecking all groups, we find that all rules in each group have the same inter-group dependencies so we can stop. This is easy to verify as we have just ensured the inter-group dependencies are the same for G_1 and G_2, this must also be true of G_3 and G_5 as they contain only a single rule, and each rule in G_4 holds a dependency with both G_1 and G_5.

Now we move on to picking one rule from each group such that all dependencies are maintained to create the final compressed ruleset. Figure 6.5 shows, in blue, the rules selected for the final compressed ruleset.

We start from the lowest priority group in the last table. This is $\mathbf{G_5} =$

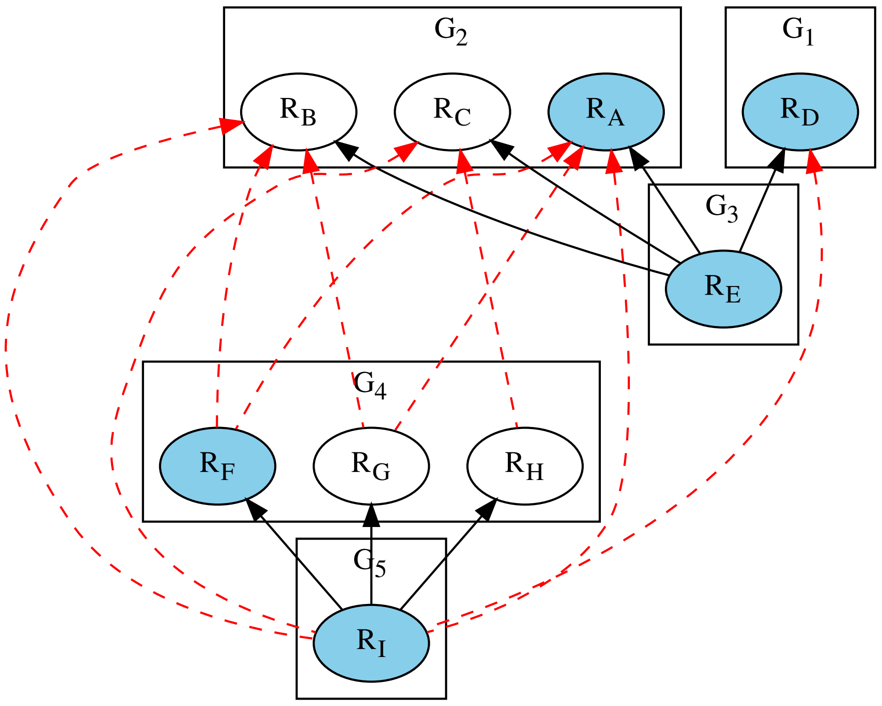


Figure 6.5: Dependency graph of the ruleset shown in Figure 6.3, showing final groups. The blue rules are those selected for the compressed ruleset. Compared to Figure 6.4, R_D has been placed into its own group because it did not share a dependency with G_4 as rules in G_2 do.

$\{R_I\}$, and we only have one choice so must pick R_I. Next we consider selecting a rule from $\mathbf{G_4} = \{R_F, R_G, R_H\}$, which must maintain the dependency with the rule R_I selected from G_5. As all rules hold a dependency with R_I we can chose any, let's pick R_F. From $\mathbf{G_3} = \{R_E\}$ we must pick R_E and from $\mathbf{G_2} = \{R_D\}$ we must pick R_D. Finally, we consider $\mathbf{G_1} = \{R_A, R_B, R_C\}$, and now to maintain the group dependency we must pick either R_A or R_B to maintain the dependency with R_F from G_4, whereas, R_C does not. If we had picked R_H from G_4 we would have been forced to chose R_C from G_1.

As a result, we have removed four rules from the original ruleset: R_B, R_C, R_G, and R_H. The compressed ruleset contains five rules instead of the original nine, simplifying the problem. Other than the performance benefit, fitting a compressed ruleset has an advantage over the original ruleset as the solver will place similar rules in the same manner resulting in a more natural and human readable solution. Additionally, this compressed solution is easier to code into an SDN application, as similar rules will all follow the same code path.

As the compressed ruleset is treated as a model to generalise fitting the original ruleset, it could also be incrementally updated when rules are added and removed without the need to rerun the solver. For example, consider what happens when a controller learns a new host in the example ruleset (fig. 6.3). If a new host was learnt on port 1, the controller would add a rule to table 1 at priority 0xF with the match $\left\{ \begin{array}{ll} \textbf{VLAN} & 1 \\ \textbf{ETH\_DST} & AD: \end{array} \right\}$ and the action $[\ \textbf{OUTPUT}\ \ 1\ ]$. This new rule is similar to the other rules in G_4 in the final groupings (fig. 6.5) and also holds the same inter-group dependencies so can be added to G_4 and fitted in the same manner as all other G_4 rules. Incrementally updating a ruleset in this manner remains future work and is not explored in this research.

By the same logic, the number of hosts learnt by the network is irrelevant to the problem size after compression. The example ruleset with 1000 hosts learnt, would still only require solving the problem for five rules.

6.2.3.4 What if a Single Rule From a Group Cannot be Selected While Maintaining all Dependencies?

It is possible to construct a ruleset in which a single rule from each group cannot be picked while maintaining all inter-group dependencies. We have seen this arise in real rulesets. Figure 6.6 shows an example ruleset where this occurs. In this example, table 0 applies routing, table 1 applies a whitelist ingress policy, and table 2 applies a whitelist egress policy. The ruleset is simplified to show only the match portion of rules to illustrate this issue. Assume that all rules direct packets to the next table, are at the same priority, and their actions do not modify any header fields.

Consider the example ruleset after creating the final groups, i.e. after Step 3 in Section 6.2.3.2. Those final four groups are:

1. $\mathbf{G_1} = \{R_A, R_B\}$ — Control routing

2. $\mathbf{G_2} = \{R_C, R_D\}$ — Filter ingress traffic reaching a host

Table 0	Table 1	Table 2
R_A { **IPV4_DST** 192.168.2.0/24 }	R_C { **IPV4_DST** 192.168.2.1, **TCP_DST** 22 }	R_G { **IPV4_SRC** 192.168.2.0/24, **TCP_DST** 443 }
R_B { **IPV4_DST** 192.168.3.0/24 }	R_D { **IPV4_DST** 192.168.3.1, **TCP_DST** 443 }	R_H { **IPV4_SRC** 192.168.3.0/24, **TCP_DST** 22 }
	R_E { **IPV4_DST** 192.168.2.2, **IPV4_SRC** 192.168.2.0/24 }	
	R_F { **IPV4_DST** 192.168.3.2, **IPV4_SRC** 192.168.3.0/24 }	

Figure 6.6: A ruleset where it is not possible to pick only one rule from each group and maintain all the original dependencies during compression.

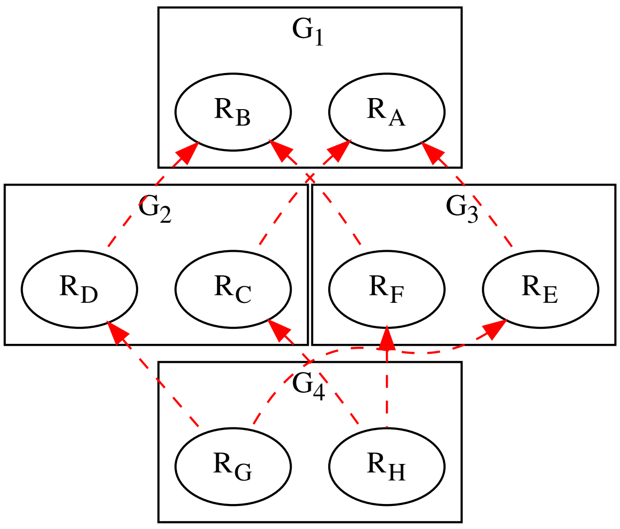


Figure 6.7: The final grouping of similar rules and the dependencies of the ruleset in Figure 6.6 as calculated by the compression process. It is impossible to select one rule from each group while maintaining all inter-group dependencies.

3. $\mathbf{G_3} = \{R_E, R_F\}$ — Filter ingress traffic reaching the subnet

4. $\mathbf{G_4} = \{R_G, R_H\}$ — Filter egress traffic from the subnets

Figure 6.7 shows the dependencies between each rule in the final groups.

Now if we consider picking a minimal ruleset, we start from the last table with the lowest priority and work back through the groups from G_4 to G_1. If we pick R_G from G_4 then we must pick R_E from G_3 to maintain the dependency between G_3 and G_4. Similarly, we must pick R_D from G_2 to maintain the dependency between G_2 and G_4. Now consider selecting a rule from G_1, it is impossible to pick a single rule which directs traffic to both R_D and R_E.

The best way to deal with this situation still requires further research. Currently, we fall back to using the original ruleset. However, a better approach could be to include both rules R_A and R_B in the minimal ruleset and ask the

Original Number of Routes	Compressed
100	41
1,000	144
5,000	539
10,000	929
50,000	3,417
740,332	48,389

Table 6.1: The resulting ruleset size after compressing routing tables of different sizes. The full routing table contains 740,322 rules, and we generated smaller tables by taking the first rules from this full table.

solver to place these in the same place.

6.2.3.5 Compressing a Routing Table

The typical way for a controller to install routes in OpenFlow is to place each prefix length at a different priority to maintain the longest-prefix-match behaviour. All /24's are placed at a higher priority than /23's and so on. As compression uses priority to classify similar rules, compression places each prefix into its own initial coarse group. Then these groups are split further to ensure the same inter-group dependencies. This further splitting can result in many groups which maybe unnecessary as we would expect all routes to be installed in the same table.

This inter-group dependency splitting happens, for example, when some /24 prefix rules shadow a /22 directly, while other /24 prefix rules shadow a /23 which then shadows the /22. Consider the initial /22 group, some of its rules will have dependencies with only the /24 rules and others only the /23, so it will be split into two groups. This split then also propagates to all other prefix groups with an indirect dependency, both those of higher and lower priority (shorter and longer prefix length).

Table 6.1 shows the effectiveness of compression on partial routing tables and the full routing table.

While the compression massively reduces the number of rules, by 93% of the full table, in absolute terms 48,410 rules is still too many for our solver.

Due to the complex nature of dependencies between rules in a routing table, we do not think that these numbers generalise to other routing tables of the same size. In particular, we generated the smaller rulesets by taking the first n rules from the full routing table, which is unlikely to match the characteristics of a real-world routing table of the same size.

Reducing the size of routing and similar rule patterns remains a challenge for future research. Ideally, the entirety of such matches can be compressed down to just a few representative rules. Two problems need to be solved 1) how to best detect prefix matching or similar masked matching in the general case, and 2) how many and which rules are required to create a representative compressed ruleset.

6.2.3.6 Related Work

Beckett et al. use a technique similar to the compression that we presented in this section to compress the control plane behaviour of large networks [58]. Beckett et al. had observed that many analysis and verification tools do not scale well with large networks. However, there is also much symmetry in networks; for example, a network spine might have similar clusters of leaf and edge routers attached. So they developed a tool named Bonsai to compress away this symmetry, thus reducing the input size given to existing analysis tools without affecting the results.

More concretely Bonsai models how routing information is shared between routers for a particular destination in a network and compresses routers sharing similar routing information into a single router which analysis tools can process faster. Beckett et al. have proved many properties are preserved in this representation, including reachability, path length, black holes, and routing loops, while necessarily losing the number of paths between nodes due to the compression. Beckett et al. found that Bonsai can compress the number of routers in real networks by over a factor of 5 and speed up analysis by orders of magnitude.

In comparison to our compression technique, Bonsai creates similar groups based on the topology and the configuration of routers, whereas we create similar groups based on dependencies between rules. Bonsai creates its initial set of coarse groups by putting the destination router in its own group and all other routers in a separate group, whereas, we create coarse groups heuristically based on rules we expect to be able to place the same within a hardware pipeline. Other than the inputs to the process, we use the same abstraction refinement method to create groups as Bonsai's forall-exists abstraction.

6.3 Finding Rule Transformations

Given an input ruleset and a TTP, the solver first finds the available rule transformations in the new OpenFlow pipeline. This section covers the transformations which we have implemented in the solver. Other useful transformations are listed in Section 6.3.8 which future work could consider.

A transformation is a mapping from an original rule, or rules along the same path, to a placement, or placements. Placement is the name that we give to a rule placed into the output ruleset. A transformation's placements must have equivalent forwarding to the original rules when considered in isolation. The next stage of the solver is responsible for picking a combination of transformations where placements are reachable and do not conflict.

6.3.1 Placing a Rule in a Target Pipeline

Vital to understanding the basis of all the transformations is understanding how our Table Type Pattern (TTP) library determines where a rule can be placed using the SATISFIES method, as originally introduced in Section 3.2.4.

SATISFIES is called on the Table Type Pattern with a rule to fit into the pipeline; it returns a list of pairs. Each pair contains two rules, 1) the placement for a rule in the target pipeline, and 2) a rule containing the unplaced matches and actions. In this way, SATISFIES can return both fully and partially placed rules. Where a fully placed rule has an empty flow rule as its unplaced pair, therefore all matches and actions from the original rule are in the placed rule. A partially placed rule has one or more matches or actions unplaced. If a match is unplaced, the rule will match more packet-space than the original. If an action is unplaced, then the rule almost certainly has different forwarding behaviour, and this missing action will need to be applied elsewhere in the pipeline.

SATISFIES does not only try to fit all fields directly as is, but it also tries variations that may prove to be equivalent. The variations of placements that SATISFIES attempts are:

- Rules both with and without the clear-actions instruction. Clear-actions are used to 'undo' actions set earlier in the pipeline. Whether or not clear-actions is required depends on the order of actions with the target pipeline.
- Rules both with and without a goto-table instruction. Thus creating and exploring all possible paths through the target pipeline.
- Rules with actions both placed in the apply-actions and write-actions instructions regardless of the original instruction containing the action. Allowing both increases compatibility because both are equivalent in most circumstances.
- Rules with actions moved both in and out of groups. Moving actions between groups will be equivalent in most circumstances and is particularly important for OF-DPA, as OF-DPA only allows output actions in groups.

6.3.2 A Direct Transformation

A direct transformation takes one rule from the input ruleset and places it as a single rule with the same forwarding in the target pipeline. The placed rule

Table 0		
Pri	VID	Actions
10	1	PopVlan, Goto:1
ⓐ*10*	*2*	*PopVlan, Goto:1*

Table 1		
Pri	DstIP	Actions
5	1.0.0.0/8	Out:1
5	2.0.0.0/8	Out:2
ⓑ*1*	*0.0.0.0/0*	*Out:10*

Merge $\downarrow$ $\uparrow$ **Split**

Single Table			
Priority	VID	DstIP	Actions
$10 \cdot 2^{16} + 5$	1	1.0.0.0/8	PopVlan, Out:1
$10 \cdot 2^{16} + 5$	2	1.0.0.0/8	PopVlan, Out:1
$10 \cdot 2^{16} + 5$	1	2.0.0.0/8	PopVlan, Out:2
$10 \cdot 2^{16} + 5$	2	2.0.0.0/8	PopVlan, Out:2
$10 \cdot 2^{16} + 1$	1	0.0.0.0/0	PopVlan, Out:10
ⓒ$\mathit{10 \cdot 2^{16} + 1}$	*2*	*0.0.0.0/0*	*PopVlan, Out:10*

Figure 6.8: A demonstration of fully merging two tables into one. Taking an individual flow example ⓐ and ⓑ are merged to form ⓒ, the reverse direction is the split operation.

does not necessarily have to be identical to the original, because it can have any combination of the variations applied by SATISFIES. Direct transformations are the full placements found by SATISFIES; there can be more than one due to the variations SATISFIES attempts.

In this stage, the solver generates direct transformations for all tables in the target pipeline. Direct placements are useful to find placements for filtering rules, such as dropping all of a traffic class, so they can be placed in an ACL table without worrying about how traffic is directed to that table. It is the responsibility of the next stage of the solver to pick a combination of transformations that direct traffic to reach this placement.

6.3.3 A Merge Transformation

The solver uses the MERGE operation from Section 4.2.1 to combine two rules from different tables in the original ruleset into a single rule placed in the target pipeline. The solver attempts to merge all rules in separate tables which are connected by a goto-table instruction. For two selected rules, MERGE combines matches by taking the intersection, if empty, no packet can hit both rules,

Placed		Unplaced Input		
VID	Actions	VID	DstIP	Actions
2	PopVlan, Goto:1	—	0.0.0.0/0	Out:10
2	Goto:1	—	0.0.0.0/0	PopVlan, Out:10
2	PopVlan	—	0.0.0.0/0	Out:10
2	—	—	0.0.0.0/0	PopVlan, Out:10
—	PopVlan, Goto:1	2	0.0.0.0/0	Out:10
—	Goto:1	2	0.0.0.0/0	PopVlan, Out:10
—	PopVlan	2	0.0.0.0/0	Out:10
—	—	2	0.0.0.0/0	PopVlan, Out:10

Table 6.2: Partial placements of rule ⓒ into Fig. 6.8 *Table 0*. Rule ⓒ matches VID:2, DstIP:0.0.0.0/0 and applies the actions PopVlan, Out:10. *Table 0* can match VID, PopVlan, and Goto:1. Partial placements include those with Goto:1 instructions added.

so MERGE generates no rule. MERGE combines flow instructions following the processing OpenFlow standard. MERGE concatenates apply-actions, and combines write-action by taking into account clear-actions instructions and the set behaviour. Once the solver merges two rules, it treats them like a direct placement and calculates placements of the merged rule using SATISFIES.

A merged placement will typically match a smaller packet-space than the original rules due to matching only the intersection of the original rules. One must replace a rule with the full Cartesian product (MERGE) of rules in the next table or alternatively the previous tables, to retain the original forwarding.[1] A rule is *fully merged* if it meets this requirement. Fully merged rules are essential in the next stage to find a valid final ruleset.

Figure 6.8 demonstrates why the full Cartesian product is required to preserve the original forwarding. If you consider the result of merging the two rules highlighted ⓐ with ⓑ to form ⓒ; the resulting rule, ⓒ, misses the VLAN:1 traffic from *Table 0* which would have hit the ⓑ rule. In order for the merged ruleset to process all packets that hit rule ⓑ, all rules in *Table 0* merged with ⓑ must be present in the merged ruleset.

[1]Multiple tables can goto another, but one rule can only goto one table

6.3.4 A Split Transformation

A split transformation takes a single rule and splits it across multiple tables, the opposite of a merge. Figure 6.8 shows how rule ⓒ can be split into rules ⓐ and ⓑ. The individual rules in a split placement will often match more packets than the original rule and apply only a portion of the original actions. Consider the split transformation shown in Figure 6.8, ⓒ is split into ⓐ and ⓑ both of which apply actions to a broader set of packets than the original rule. ⓑ applies the an output action from ⓒ to all IP packets, however, ⓒ only matched to packets which were both IP packets and had a VLAN ID of 2. It is the responsibility of solver's next stage to pick a combination of transformations that avoids the more broad rules in such split placements from conflicting with other placements.

The solver finds split transformations by taking a rule and using SATISFIES to find all partial placements in the target pipeline. Table 6.2 shows an example of partial placements of ⓒ into *Table 0*. The solver then finds all valid paths through these partial placements by following the placement's goto instructions. Then the solver filters these resulting paths to include only those resulting in the same forwarding as the original rule for the corresponding packet-space.

6.3.5 Filtering Split Transformations

The number of paths that the solver checks when generating a split transformations is the product of the partial placements it finds for each table. This number quickly becomes very large in a multi-table pipeline, as do the number of valid split transformations.

Consider the partial placements shown in Table 6.2. Because the VID match is optional in the target pipeline, each placement has a variation with and without the VID match. The variations without the VID match are more general and match all packets and are therefore more likely to conflict with other placements.

Input ruleset

	IPv4 Dst	TCP Dst	Actions
ⓐ	192.168.1.0/24	22	Drop
ⓑ	192.168.1.0/24	—	Out:1

Target Pipeline Requirements

Routing Decision (Table 0): Must match an IPv4 subnet. Actions **can** add an output to the action-set, then **must** goto Table 1.

Access Control List (Table 1): May include any arbitrary match. Actions **can** clear the action-set.

Split Transformations per rule

	Table 0		**Table 1**		
	IPv4 Dst	Apply-Actions	EthDst	TCP Dst	Actions
ⓐ	192.168.1.0/24	Goto:1	192.168.1.0/24	22	Clear-Actions
ⓑ	192.168.1.0/24	Out:1, Goto:1	192.168.1.0/24	—	—

Figure 6.9: An example where ⓐ's split transformation needs an additional 'wrong' action to avoid conflicting with ⓑ's placement in Table 0. Consider the two descending-priority ordered rules in the input ruleset, given the pipeline constraints this figure shows one possible split transformation for each. If the solver was to install both transformations, ⓐ's placement in Table 0 takes priority over ⓑ's placement. Therefore the default forwarding decision in ⓑ is lost, and forwarding is incorrect. However, if ⓐ's placement in Table 0 was replaced with ⓑ's placement then the forwarding is correct.

Instead of considering all possible partial placements, the solver only considers the partial placement with the most specific match for each unique set of actions. With the example shown in Table 6.2 this means that only the first four partial placements, which include the VLAN match, are considered.

More precisely the solver filters partial placements before generating split placement paths. The solver filters out any placements which have the same action as another placement and where the match is a superset of the other placement's match. Consider the case the original rule matches three fields. If there are no partial placement matching all three fields and only placements matching two fields, then all these two field combinations would remain, as none are supersets of each other.

6.3.6 Adding Additional ‘Wrong’ Actions

Some fixed-function pipelines use the packet’s action-set to store the default forwarding decision for a broad set of packets but allow a rule in a later table to override the forwarding. This works because the pipeline executes the action-set at the end of the pipeline, and rules can overwrite or clear the action-set. As a result, transformed rules may need to be given a ‘wrong’ action so packets can traverse a table and correct it with a rule in a later table. OF-DPA follows this design: almost all tables, including L2 forwarding and L3 routing, are before the policy ACL table where a controller installs firewall policy to drop unwanted packets.

Figure 6.9 shows a simplified version of the OF-DPA’s pipeline which demonstrates the need for rules to have the ‘wrong’ action added. Recall that the next stage of the rule-fitting solver selects one transformation for each rule. If the solver selects both split placements shown in Figure 6.9, the forwarding is incorrect forwarding because the placements in Table 0 conflict. ⓐ’s placement in Table 0 takes priority over ⓑ’s and results in the incorrect forwarding for packets originally processed by ⓑ. However, ⓑ’s placement in Table 0 is compatible ⓐ’s originally forwarding, as it clears the output action, thus correctly dropping packets in Table 1.

In order to avoid such conflicts, the rule-fitting solver generates new split transformations by replacing that transformation’s placements with placements from other input rules’ transformations. For each split transformation, the solver tries to replace each placement with another placement in the same table with the same match but different actions. The solver considers all placements from other rule’s possible transformations as candidates with which to replace. If a transformation with a replacement maintains its original forwarding behaviour, it is an acceptable replacement, and the solver adds this as a new transformation.

When generating these new transformations, the solver retains the priority of the other rule’s placement. So this process also generates placements with different priorities. Switching the relative priority between two placements, changes which shadows the other and ultimately whether the overall forwarding is correct. While it is up the next stage of the solver to pick the correct priority order, this stage must generate all options.

Our solver only considers deviations of one placement changed from the original split transformation. Doing otherwise would increase the space the solver searches for possible solutions, but unfortunately, results in a substantial expansion in compute time. Future research is required to find more efficient ways of generating and representing these equivalent variations to split transformations.

6.3.7 Placement Priorities

The solver gives all transformations' placements priorities based on the original rules' pipeline priority. Therefore the highest priority placements are derived from transformations of rules in the first table with the highest priority.

The solver scales the input ruleset using the same equation as used in conversion to a single table described in Section 4.2.1.

$$new\_priority = priority \times (MaxPriority^{|tables|-1-table_{index}}) \tag{6.1}$$

Equation (6.1) shows how to calculate the scaled priority of a rule. The new priority is the product of the original priority and the maximum priority of an OpenFlow rule, 2^{16}, raised to the power of that rule's table's distance from the end of the pipeline. This scaling leaves enough space between priority adjacent rules in the first table to fit all possible priorities of the second table, and so on. A merge placement derives its priority from the sum of the original rule's scaled priority. Whereas, direct and split placements inherit their priority directly from the scaled priority of the original flow. Scaling priorities like this attempts to put rules at the correct priority relative to each other; where a more specific merged placement takes precedence over a directly placed or split rule. Additionally, by using this priority assignment fully merging a ruleset down to a single table will result in the correct priority order.

Our technique to include ‘wrong’ actions in split transformations (§6.3.6) also introduces priority variation into split placements. For example, as part of a split transformation, it is common to generate a pass-through placement, which matches all packets and passes them through a table without applying any actions. This pass-through placement retains the priority of the original rule, so for a high-priority rule, it is very likely to shadow other placements in the same table incorrectly. This priority variation technique allows a low-priority pass-through placement to be selected instead, such as an equivalent pass-through placement of priority 0 generated for a table-miss rule in the original ruleset.

These new priorities are no longer within the range of valid OpenFlow priorities. So, once a solution is found, the solver scales these priorities back to valid OpenFlow priorities.

6.3.8 Transformations: Future Work

This section outlines other practical transformations we considered, but which were not implemented in the rule-fitting solver. These were not implemented due to their complexity compared to the number of cases in practice that they solved. This is not intended to be a comprehensive list.

6.3.8.1 Using Metadata To Link Split Transformations

Individual split transformation placements cannot always include the entire original match, and thus rules must match more packets than the original did. Split transformations for two different input rules with different actions can result in two placements in the same table with the same, yet more general, match but different actions. If two rules in a table have the same match, packets only hit the highest priority rule as it shadows the other rule. Thus, forwarding for one of the split transformations will be incorrect.

Input Rules

IPv4 Dst	TCP Dst	Actions
192.168.1.0/24	22	Drop
198.51.100.0/24	22	Out:1

Split Transformation

Table 0			Table 1	
IPv4 Dst	Actions		TCP Dst	Actions
192.168.1.0/24	Goto:1	$\rightarrow$	22	Drop
198.51.100.0/24	Goto:1	$\nearrow$	22	Out:1

Linking Split Transformation With Metadata

Table 0			Table 1		
IPv4 Dst	Actions		MD	TCP Dst	Actions
192.168.1.0/24	Set MD:10, Goto:1	$\rightarrow$	10	22	Drop
198.51.100.0/24	Set MD:20, Goto:1	$\rightarrow$	20	22	Out:1

Figure 6.10: An example using Metadata (MD) to link the placements in a split transformation together. In the scenario shown, two rules from a single table are placed using a split transformation into the target pipeline. In the target pipeline, only Table 0 can match the IPv4 Dst and Table 1 the TCP Dst. The two split transformations conflict when combined because Table 1 can only apply one action to packets with TCP Dst 22, not two. Table 1 needs to apply two different actions depending on the rule matched in Table 0. To solve this, Table 0 can set metadata to share the rule hit with Table 1. Table 1 then matches this metadata in addition to the TCP Dst to apply distinct forwarding depending on the rule hit in Table 0.

The OpenFlow 1.3 metadata header field is a specialised field that only exists within the OpenFlow pipeline for sharing information between tables [12]. Figure 6.10 shows an example of using metadata to link the placements of split transformations into a path. Without metadata to link the paths, the split transformations create conflicting placements in Table 1 where one placement has a drop action and the other an output action.

The FlowAdapter and FlowConvertor rule-fitting solvers extensively use metadata to ensure packets take the required path [38, 37]. Our rule-fitting solver avoids reliance on metadata because fixed-function pipelines often do not support metadata. If supported by the pipeline, metadata can be used, alternatively, any other reversible packet operation can encode this information. For example, the solver can achieve this same behaviour by installing rules to push and set a Virtual LAN (VLAN), Provider Backbone Bridging (PBB), or

Aggregated	
IPv4 Dst	Actions
192.168.1.40/30	Out:1

$$\begin{array}{c}\longleftarrow \\ \longrightarrow\end{array}$$

Deaggregated	
IPv4 Dst	Actions
192.168.1.40	Out:1
192.168.1.41	Out:1
192.168.1.42	Out:1
192.168.1.43	Out:1

Figure 6.11: An example transforming between rules with a masked match into multiple rules with an exact match. Deaggregation facilitates placing a rule with a wildcard match into a pipeline which only supports an exact match.

Multiprotocol Label Switching (MPLS) header, then subsequently match and finally pop that header.

6.3.8.2 Transforming Between Masked Matches and Exact Matches

A rule with a masked match cannot be placed directly into a pipeline which only supports an exact match. Figure 6.11 shows that a masked match can be split into an equivalent set of exact matches. Deaggregating facilitates the rule-fitting solver placing a rule with a masked match into a target pipeline that only supports an exact match.

Deaggregating a masked match into exact matches can result in many rules and is impractical for rules with many bits masked. However, in the case of only a few masked bits, it could be the difference between the solver successfully fitting the ruleset and the solver failing. Splitting a wildcard match would require a threshold to ensure it remains useful. For example, allowing only up to 4 wildcard bits would result in at most 16 rules, yet allowing 32 wildcard bits can result in over 4 billion rules.

Conversely, aggregating exact matches together into a masked match reduces the rules required of the target pipeline supports masked matches.

6.3.8.3 Field-Centric Tables

Some pipelines have field-centric tables, in such tables a particular optional field must present on all packets, and rules within the table must match an exact value of the field. Field-centric tables logically separate, for each unique

Input rule

Port	EthDst	Actions
6	02:...:01	Out:1

Target Pipeline Requirements

Untagged VLAN Assignment (Table 0): Must match an exact ingress port and packets without a VLAN tag. Actions **must** push and set the VLAN, then goto Table 1.

Switching (Table 1): Must match an exact VLAN and Ethernet Destination. Actions **can** include popping the VLAN and selecting output.

Field-centric Split Transformation

Table 0		
Port	Vlan	Actions
6	No VLAN	PushVlan:1, Goto:1

Table 1		
Vlan	EthDst	Actions
1	02:...:01	PopVlan, Out:1

Figure 6.12: An example of transforming a non-VLAN-aware rule into a simplified VLAN-centric pipeline, loosely based on OF-DPA. VLAN-centric pipelines force VLAN isolation and require all packets to have a VLAN assigned before forwarding decisions are made. In this example, Table 0 forces assigning a VLAN to all packets without a VLAN, otherwise they will be dropped. Thus all packets entering Table 1 include a VLAN tag, and Table 1 requires an exact VLAN is matched. Note, the input rule will also forward packets with VLAN tags, where as the transformed rule ignores this case so is not strictly equivalent.

value of that field, one table into multiple virtual isolated tables.

For example, fixed-functions pipelines like OF-DPA are VLAN-centric and enforce all packets include a VLAN tag in most tables. Requiring a VLAN tag is a common design of fixed-function pipelines because VLAN isolation is a fundamental part of Layer 2 VLAN switching and is built into the hardware's design. This also occurs with other tags like MPLS, and other situations like virtual routing tables, where a virtual routing ID must be assigned to all packets.

Figure 6.12 gives a simplified version of the OF-DPA showing how it enforces all packets to have a VLAN and shows how a non-VLAN-aware rule can be transformed. Table 0 demonstrates how a table early in the OF-DPA pipeline enforces that only VLAN tagged packets go to the next table. Table 0 drops all untagged packets unless the controller adds a rule matching untagged packets and assigns them a VLAN. Table 1 demonstrates how switching in OF-DPA is separated into logically isolated virtual tables, one for each VLAN, by requiring all rules match an exact VLAN.

Figure 6.12 shows how to transform a non-VLAN-aware rule into this pipeline. The rule is split between the tables with modification in Table 0 to push a default VLAN tag and modification in Table 1 to pop the VLAN tag before outputting the packet. This is not strictly equivalent to the original, as the the original rule will forward VLAN tagged packets, however, in the transformation Table 0 will drop packets with VLANs. It is possible in OF-DPA to install another rule in Table 0 to correctly handle packets with VLANs, the example ignores this case for simplicity.

Detecting field-centric tables in the general case is non-trivial as simply looking for tables which require an exact match field is insufficient. For example, consider that the rules in Table 1 require an exact match on Ethernet destination. Additionally, the original ruleset may be written with assumptions on the types of packets on the network, such as no packets have a VLAN tag but does not explicitly include that in its match. A transformation without these assumptions is difficult, and may be impossible, to maintain strict forwarding equivalence. Future work is required to detect and create transformations for field-centric tables in the general case.

Chapter 7

SAT Solver: Finding a Valid Combination of Transformations

The first stage of our rule-fitting solver outputs a list of possible transformations. Each transformation maps from rules in the input ruleset to placements in the target pipeline. In isolation, each transformation applies the correct forwarding. However, when transformations are combined, their placements can conflict with each other. For example, a placement can shadow packets from reaching another rule, or a placement is installed in an unreachable table.

This chapter discusses the final stage of the rule-fitting solver which aims to find a valid combination of these transformations. In our initial attempts to solve this problem, it was clear that some combinations of transformations would always lead to invalid solutions and did not warrant further consideration. Thus our algorithm needed to skip over these conflicting transformations quickly. It became apparent that writing our own algorithm to do this was difficult. So we searched for alternatives and found that we could express the problem as a boolean satisfiability (SAT) problem. Which meant we could use an existing SAT solver to generate combinations of transformations for the rule-fitting solver to consider.

In this design, the rule-fitting solver creates an initial set of constraints as a boolean expression to remove solutions that are very unlikely to result in the correct forwarding. The rule-fitting solver gives this boolean expression to the SAT solver and then checks the equivalence of the candidate solution the SAT solver returned. If a candidate solution is invalid, the solver finds the particular conflicting placements and adds these as a constraint to the SAT expression, then reruns the SAT solver.

The remainder of this chapter is laid out in the following order. The chapter starts by introducing fundamentals for later understanding the SAT problem the rule-fitting solver creates. Section 7.1 introduces the basics of the boolean satisfiability problem and SAT solvers. Section 7.2 introduces the notation this chapter uses to represent boolean expressions. Section 7.3 describes considerations and pitfalls to avoid when constructing boolean expressions.

Building on these fundamentals, Section 7.4 provides an in-depth description of the initial SAT expression the rule-fitting solver supplies to the SAT solver. Section 7.5 describes how the rule-fitting solver converts the output from the SAT solver into a candidate solution which checks for equivalence with the original ruleset. If this candidate solution is invalid, then Section 7.6 describes how the rule-fitting solver finds conflicting transformations and incorporates them in the SAT expression to prevent unnecessarily considering invalid solutions.

7.1 The Boolean Satisfiability Problem

Consider a boolean expression formed from boolean variables and boolean operations, for example, $(A \land B) \lor C$. Changing the values of the boolean variables A, B, or C will change the overall truth of the expression. A boolean expression is said to be satisfied if it evaluates to true.

The boolean satisfiability problem asks if it is possible to satisfy a given boolean expression; therefore, to determine if an assignment of variables exists where the expression evaluates to true. Literature commonly refers to the boolean satisfiability problem simply as the SAT problem.

```
c A comment
p cnf 4 3
1 -3 0
2 3 0
-4 0
```

Figure 7.1: A sample conjunctive normal form DIMACS file representing the equation $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (\neg x_4)$. Lines beginning with 'c' are **c**omments. One line begins with 'p' and describes the **p**roblem: p <format> <number variables> <number of clauses>. Following the problem are clauses, which are combined using conjunction (logical AND). Each clause is a disjunction (logical OR) of the variables or negated variables (when preceded by -) listed. Variable numbering starts at 1. Each clause is terminated by the 0 rather than the newline character.

The SAT problem was the first NP-complete problem found as per the Cook-Levin theorem [59, 60]. Being NP-complete means that all other NP problems can be converted to the SAT problem in polynomial time. Finding a polynomial-time solution to SAT would answer the P versus NP problem. Solving the SAT problem remains a challenge with the brute-force approach, checking all combinations of every variable, the only known general solution. Checking these combinations thus scales exponentially 2^n, where n is the number of variables.

Due to the ubiquity of problems that the SAT problem can express it is a well-researched field. SAT solvers are heavily researched, with an annual competition [61] presenting a challenge to the research community to better the state of the art [62]. Today, SAT solvers use many different techniques and heuristics to reduce the problem size and to find a solution quickly, without needing to explore the entire problem space.

The Conjunctive Normal Form (CNF) DIMACS [63] format is the de-facto input and output format of most SAT solvers. The DIMACS format was created for the DIMACS SAT solver competition and has remained the standard format used in competitions. Figure 7.1 shows an example of the DIMACS file format. Almost all SAT solvers and libraries support the DIMACS format.

Boolean Operators

Name	**Symbol**
Negation	$\neg$
And (Conjunction)	$\wedge$
Or (Disjunction)	$\vee$
Exclusive or (Xor)	$\oplus$
Implies	$\rightarrow$
Equivalence	$\leftrightarrow$

Boolean Set Flattening

Name	**Symbol**	**Description**
Big And	$\bigwedge\{...\}$	$\bigwedge\{a, b, c\}$ is equivalent to $a \wedge b \wedge c$
Big Or	$\bigvee\{...\}$	$\bigvee\{a, b, c\}$ is equivalent to $a \vee b \vee c$
At Most One	$\textsc{amo}(\{...\})$	Satisfied if only 0 or 1 expressions are true
One Hot	$\textsc{onehot}(\{...\})$	Satisfied if only one expression is true

Set Operations

Name	**Symbol**	**Description**
Union	$\cup$	
Intersection	$\cap$	
Element In	$\in$	
Subset	$\subseteq$	
For All	$\forall$	We use $\forall$ to iterate a set e.g. $\forall x \in \{...\}$. Logically equivalent to $\bigwedge$.

Figure 7.2: Boolean and set notation.

For problems that can be expressed as SAT, the DIMACS format makes it easy to test different SAT solvers with minimal code changes. Thus we designed the rule-fitting solver to construct the SAT problem in the DIMACS format to decouple itself from any particular SAT solver.

7.2 Boolean Notation

$$(x_1 \vee x_2 \vee x_3 \vee ...) \wedge ... \wedge (\neg x_1 \vee x_2 \vee x_3 \vee ...) \quad (7.1)$$

Equation (7.1) shows the composition of a boolean expression in CNF. By definition, CNF is a combination of clauses using logical AND. Where each clause is a combination of literals, e.g. x_1, or negated literals, e.g. $\neg x_1$, using logical OR.

Our rule-fitting solver expresses the boolean expression to the SAT solver in CNF form using the DIMACS format. However, building a boolean expression directly in CNF form is unwieldy and difficult to understand. Instead, this chapter defines the rule-fitting problem using a complete set of boolean operations, as listed in Figure 7.2. It is then easy for the solver to convert arbitrary expressions into CNF form.

The final SAT expression is built by combining together all smaller partial expressions using logical AND. Each partial expression is a piece of logic which constrains the problem. Breaking the problem up like this makes it more understandable, and makes it trivial to enable or disable constraints to test their effectiveness. Additionally, if the solver builds each partial expression in CNF form, the combining these to form the complete expression amounts to the concatenation of these expressions, which is inexpensive.

In this chapter, we have opted to use a verbose description in our equations, for example, using 'transformations' rather than 'T' as the set of transformations. Additionally, we use both 'for all' ($\forall$) and 'big and' ($\bigwedge$) to make our expressions more readable, despite both having the same meaning; to combine all items together with logical AND. We make a distinction by using $\bigwedge$ to combine a set as part of a partial expression for one constraint. While we use $\forall$ to iterate sets of items where, for each item, we generate a partial expression for the constraint.

Additionally, $\bigwedge$ only applies to a set of boolean expressions which includes single boolean variables. Whereas, we use $\forall$ to iterate sets which are not of boolean expressions, and do not have a corresponding boolean variable in the boolean expression. For example, a rule from the input ruleset does not have a corresponding boolean variable in the SAT expression, so will only ever be iterated with $\forall$.

For example, consider adding a constraint for a set of rules: for each rule select all merge transformations or one split transformation. We can express this constraint mapped verbatim as:

$$\forall r \in \text{rules}: \\ \bigwedge \Big[\text{merge-transformations}(r)\Big] \vee \textsc{onehot}(\text{split-transformations}(r)) \quad (7.2)$$

Equation (7.2) uses $\forall$ to iterate over all rules, and then for each rule, generates a partial expression which limits the transformations. The left half of the partial expression uses $\bigwedge$ so it is only satisfied when all merge transformations are selected. Where merge-transformations(r) returns the set of variables representing all merge transformations of r. The right half of the partial expression uses one-hot (ONEHOT) to ensure that it is only satisfied when one split transformation is selected. Notice that the partial expression does not directly contain a rule (r) as a variable.

Ultimately, because the final expression is the logical AND of all these partial expressions created for each rule, $\forall$ maintains its logical equivalence to $\bigwedge$.

7.3 Considerations when Developing Partial Expressions

A partial expression must only be satisfied when it meets the underlying constraint. It is straightforward to create an expression that is satisfied when a constraint is met, but also unintentionally when it is not. Logical OR and XOR are easy to confuse due to both mapping back to ‘or’ in the English language. Logical fallacies are easy to make with operations such as implication. Additionally, a written constraint can imply additional assumptions, often apparent to the reader, but which are not explicitly stated.

To demonstrate this, let us reconsider Equation (7.2) from the previous section. The equation is obviously satisfied when all merge transformations or one split transformation are selected. However, it is also satisfied when all merge transformations and one split transformation are selected. We can easily rectify this by replacing the OR ($\vee$) with an XOR ($\oplus$):

$$\forall r \in \text{rules}: \quad \bigwedge\Big[\text{merge-transformations}(r)\Big] \oplus \textsc{onehot}(\text{split-transformations}(r)) \tag{7.3}$$

Now consider the equation is satisfied by the left half, all merge transformations are selected. The left half being true implies the right half is false, and therefore, there is **not** precisely one split transformation selected. Herein lies a second problem, two or more split transformations can be selected, which is certainly not the intention of the original statement as written. Similarly, the expression is satisfied by precisely one split transformation and any number of merge transformations, other than all.

Assume that any number of merge transformations is acceptable. To fix the problem with split transformations, we can add additional partial expressions to disallow more than one split transformation at once:

$$\forall r \in \text{rules}: \quad \textsc{amo}(\text{split-transformations}(r)) \tag{7.4}$$

Equation (7.4) uses at-most-one (AMO) and not one-hot (ONEHOT); otherwise, it prohibits Equation (7.3) selecting all merge transformations.

Now that we have addressed hidden assumptions in the original written constraint, we will address practical considerations. AMO is an expensive operation, and ONEHOT uses it as its base with an additional clause disallow all expressions being false. A naive AMO implementation creates a clause for every possible pair of boolean expressions in the set to disallow both being true ($\neg a \vee \neg b$). Therefore, expressions should use AMO and ONEHOT sparingly.

Therefore, we can avoid this second call to AMO entirely by moving the

ONEHOT call in the original equation:

$$\forall r \in \text{rules}: \\ \textsc{onehot}\Big(\Big[\bigwedge \text{merge-transformations}(r)\Big] \cup \text{split-transformations}(r)\Big) \quad (7.5)$$

In Equation (7.5), one of the items given to ONEHOT is an expression rather than a simple variable. Expressions in AMO are expensive because AMO has to duplicate the expression many times over, which is much more expensive than a single variable. So in such cases, we can avoid this duplication by linking an expression to a variable. We use logical equivalence $\leftrightarrow$, which functions like variable assignment:

$$\forall r \in \text{rules}: \\ \Big(x \leftrightarrow \Big[\bigwedge \text{merge-transformations}(r)\Big]\Big) \wedge \\ \textsc{onehot}\Big(\{x\} \cup \text{split-transformations}(r)\Big) \quad (7.6)$$

Note that full equivalence is required to link the variable x correctly. On the surface, implication ($\rightarrow$) seems sufficient to make this linkage, but it is not. Consider $x \rightarrow \Big[\bigwedge ...\Big]$, x can be false when $\Big[\bigwedge ...\Big]$ is true, thus incorrectly allowing two expression to be true. Conversely, $\Big[\bigwedge ...\Big] \rightarrow x$ allows $\Big[\bigwedge ...\Big]$ to be False and x True, thus all expressions to be false.

In summary, written constraints do not always state all assumptions. However, to convert a written constraint into a partial expression, we must make these implicit assumptions explicit. Additionally, for practical reasons, we must use some operations sparingly to avoid significantly increasing the size of the expression.

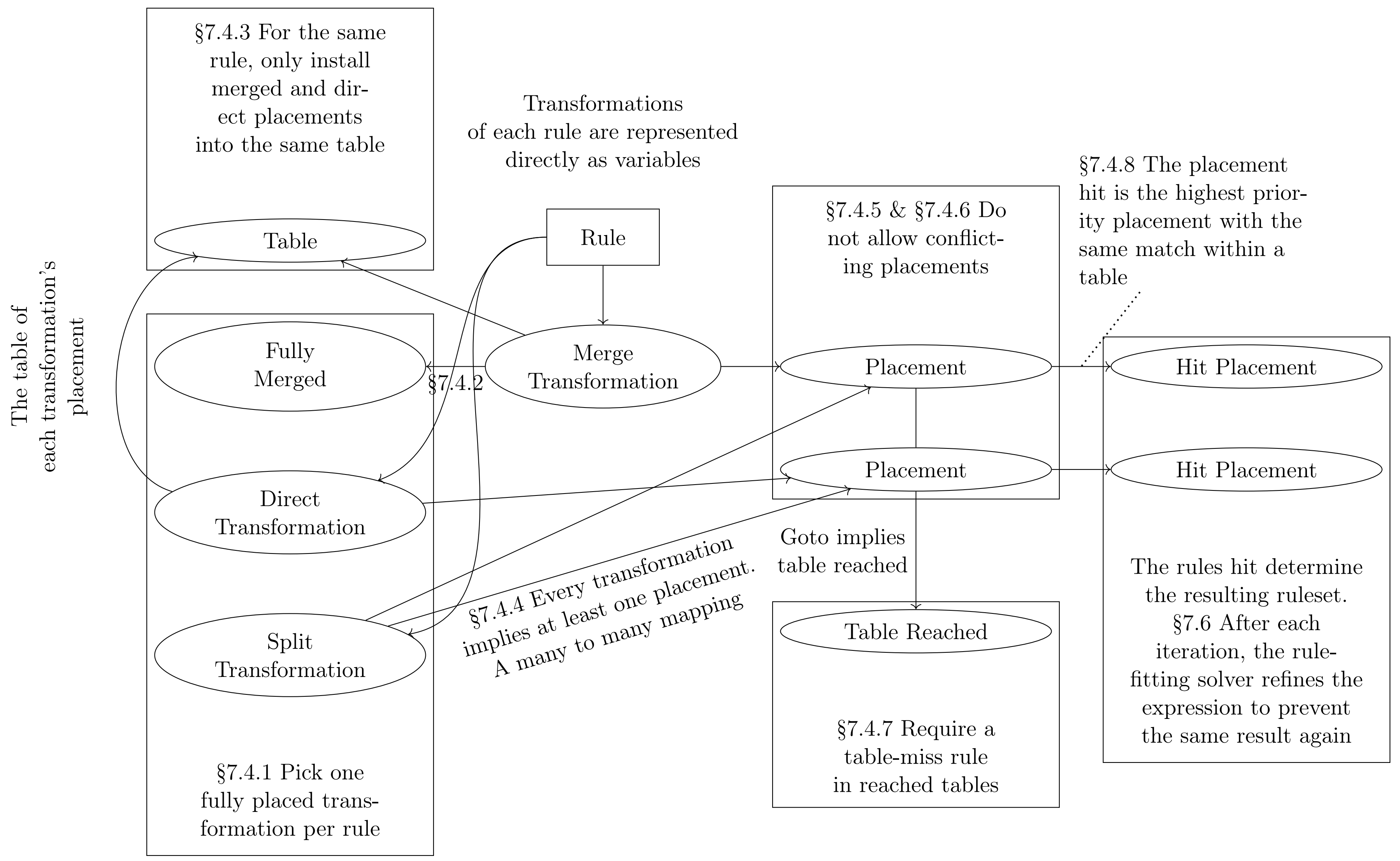

§7.4.3 For the same rule, only install merged and direct placements into the same table
Table
Transformations of each rule are represented directly as variables
Rule
§7.4.5 & §7.4.6 Do not allow conflicting placements
§7.4.8 The placement hit is the highest priority placement with the same match within a table
The table of each transformation's placement
Fully Merged
Merge Transformation
§7.4.2
Placement
Hit Placement
Placement
Hit Placement
Direct Transformation
Goto implies table reached
§7.4.4 Every transformation implies at least one placement. A many to many mapping
The rules hit determine the resulting ruleset. §7.6 After each iteration, the rule-fitting solver refines the expression to prevent the same result again
Split Transformation
Table Reached
§7.4.7 Require a table-miss rule in reached tables
§7.4.1 Pick one fully placed transformation per rule

Figure 7.3: An overview of the SAT expression the rule-fitting solver constructs. Elliptical nodes represent boolean variables within in the boolean expression. Edges show a relationship between variables and their label describes how the variables are derived. The rule-fitting solver adds a transformation variable for every transformation from its first stage. Each combination of 'hit placement' variables represents a unique ruleset. The groups of variables describe the constraints the rule-fitting solver adds to the expression between those variables.

Name	Name of Variable	Name of Set
Direct Transformation	d	direct
Merge Transformation	m	merge
Split Transformation	s	split
Transformation	t	transformations
Table	ft	tables
Table Reached	tr	tables-reached
Fully Merged	fm	fully-merged
Placement	p	placements
Hit Placement	h	hit-placements

Figure 7.4: Summary of the naming of boolean variables and their corresponding sets this chapter uses in equations. All transformations from the first stage of rule-fitting solver have corresponding variables in the SAT expression. The transformations set includes all direct, merge, and split transformations combined, and equations in this chapter use t to denote a transformation variable of any type. Equations can filter transformation sets by rule, e.g. direct(r) returns the set of direct transformations the first stage of the solver generated for the rule r.

7.4 The Initial SAT Expression

The rule-fitting solver builds a boolean expression to constrain the combinations of transformations the SAT solver returns. This expression encodes constraints we designed for the OpenFlow rule-fitting problem.

This section defines the partial expressions that the rule-fitting solver combines to create the initial SAT expression. Figure 7.3 shows an overview of the variables and constraints encoded into this expression. We designed these constraints to prevent the SAT solver returning candidate solutions which are extremely likely to be invalid, consequently reducing the search space. A transformation variable represents the mapping from rules in the input ruleset to placements from Section 6.3. The number of input rules and placements depends on the type of transformation. Multiple transformations can result in the same placement; a placement is a rule in the solution ruleset. A hit placement variable tracks whether a placement reachable by packets in the final ruleset (i.e. is ‘hit’). Fully shadowed placements will not be hit. These hit placements define the forwarding behaviour of a ruleset, and represent a unique candidate solution.

The initial SAT expression encompasses everything shown in Figure 7.3 with one exception, the hit placement constraint which the rule-fitting solver adds after every SAT solver iteration. After every iteration, the rule-fitting solver adds a constraint on hit placements to prevent rechecking the same solution. Additionally, the solver adds a more precise constraint to disallow the placements that caused a conflict in forwarding behaviour. Later, Section 7.6.1 defines both of these refinements to the boolean expression.

Figure 7.4 provides a reference to the variable and set names this chapter uses in its equations. When referring to a particular type of variable, we have used the variable name listed in Figure 7.4. We use subscript to disambiguate two variables of the same type, e.g. d_1 and d_2 to identify two different direct transformations. Sets contain all variables of that type.

The remainder of this section gives the equations for each partial expression in the initial SAT expression and explains the reasoning for each partial expression concerning the OpenFlow rule-fitting problem. The order of this section follows Figure 7.3 from left to right.

7.4.1 Include a Transformation of Every Rule

Every rule in an OpenFlow ruleset has a purpose which means a valid solution to the rule-fitting problem needs to represent every rule from the original ruleset. Therefore, the rule-fitting solver needs to select a transformation of every rule to create an equivalent ruleset. Exceptions do exist to this general case, such as unreachable rules which the solution does not need to represent. Ruleset processing (§6.2) has the responsibility of removing such cases, so we do not need to consider it here.

It is generally wrong to pick more than one transformation for a rule as it results in incorrect forwarding due to duplicate actions. For example, consider a rule placed twice in two consecutive tables, the ruleset will apply the original actions twice to a matching packet. Additionally, restricting the SAT solver to a single transformation per rule also greatly reduces the search space and improves performance.

Next, we look at precisely why picking more than one transformation for each type of transformation is undesirable.

Direct transformations: If the solver selects two direct transformations for a rule, there are two rules placed, and these must be in either the same table or different tables.

In the same table case, both placements have the same priority and match but different actions or instructions. Installing both rules results in undefined behaviour as per OpenFlow [12], so must be avoided.

In the case of separate tables, if a packet can only reach one rule, the other rule is redundant. Otherwise, if a packet reaches both rules, then the ruleset applies the same actions twice, which is either wrong or redundant. The exception to this is a pipeline which splits into two, such that a different set of packets reach both rules and both are required. Due to the complexity of detecting when this case is required, we do not support it.

Split transformations: By definition, a split transformation installs multiple placements across multiple tables. The solver generates a split transformation for all valid combinations of placements as a path. As all paths start from the first table, two different split transformations must install two different placements in the same table. Two split placements within the same table will result in undefined behaviour.

Merged transformations: Merged transformations have a more complicated relationship with one another as they take two rules as input, not just one rule as direct and split transformations do. To fully represent the original forwarding behaviour, the solver must install all merges with rules in either the next table or previous tables. We call this rule fully merged. Otherwise, to maintain the same forwarding, a partially merged rule requires a direct placement in the same table to process packets not captured by the merged placements. The same arguments for not installing multiple direct rule transformations apply to not installing multiple fully merged transformations.

Therefore, the solver adds a constraint to pick exactly one direct, split, or fully merged transformation for each rule.

$$\forall r \in \text{ruleset}: \quad \textsc{onehot}\big(\text{direct}(r) \cup \text{split}(r) \cup \text{fully-merged}(r)\big) \tag{7.7}$$

Equation (7.7) shows how to generate partial expressions to ensure the output ruleset represents every rule in the input ruleset. For every rule, r, the equation uses ONEHOT to ensure precisely one transformation from the combined set of direct, split, and fully-merged transformations is selected. Direct and split transformations map directly to corresponding boolean variables in the SAT expression. Next, Section 7.4.2 defines the partial expression that generates the fully-merged variables this equation uses.

7.4.2 Fully-Merged Variables

As a merge transformation takes the intersection of two rules' match, it usually does not represent the forwarding behaviour of either rule. To fully represent the forwarding behaviour of a rule, the solver must install that rule merged with all rules either in the preceding tables or the following table. Otherwise, if a rule is not fully merged, the solver should include another type of transformation to match packets not matched by the merged rules.

The solver adds a new boolean variable to track if a rule is fully merged. There can be multiple merge transformations between any two given rules. The rule-fitting solver considers a rule fully merged so long as one merge transformation is selected from all relevant rule pairs. We must allow partially merged rules. If we did not, any rule merged with a fully merged rule would need to also be fully merged itself, and therefore the entire ruleset would need to be fully merged.

The rule-fitting solver builds an expression to check if a rule is fully merged with preceding rules and following rules and then combines these to create variables representing fully merged rules.

$$\text{FULLY-MERGED-PRECEDING}(r) = \begin{cases} \bigwedge\{\bigvee\{m \in \text{merge}[r_p, r]\} \\ \quad \textbf{where } r_p \in \text{PRECEDING}(r)\} & \textbf{if } |\text{PRECEDING}(r)| > 0 \\ \mathit{False} & \textbf{if } |\text{PRECEDING}(r)| == 0 \end{cases} \tag{7.8}$$

Equation (7.8) defines the function FULLY-MERGED-PRECEDING for a given rule, r, which returns a boolean expression that is satisfied when r is fully merged with preceding rules. Reading from the innermost clause out, r_p iterates all preceding rules, therefore all rules which goto r's table and where that rule's egress packet-space overlaps r's match. For each preceding rule, r_p, OR ($\bigvee$) ensures at least one merge transformation between r_p and r is selected. Thus the outermost AND ($\bigwedge$) is only satisfied when at least one transformation is selected for all preceding rules. In the case that no rules precede a rule, FULLY-MERGED-PRECEDING evaluates to false.

$$\text{FULLY-MERGED-FOLLOWING}(r) = \begin{cases} \bigwedge\{\bigvee\{m \in \text{merge}[r, r_f]\} \\ \quad \textbf{where } r_f \in \text{FOLLOWING}(r)\} & \textbf{if } |\text{FOLLOWING}(r)| > 0 \\ \mathit{False} & \textbf{if } |\text{FOLLOWING}(r)| == 0 \end{cases} \tag{7.9}$$

In the same fashion, Equation (7.9) defines the function FULLY-MERGED-FOLLOWING for a given rule, r, which returns a boolean expression that is satisfied when r is fully merged with the following rules. Where FOLLOWING returns the set of rules in the table that r goes to and r's egress packet-space overlaps the other rule's match.

Both FULLY-MERGED-FOLLOWING and FULLY-MERGED-PRECEDING can be satisfied when multiple merge transformations for the same pair of rules are selected. By the same reasoning for selecting only one transformation per rule, installing two transformations for the same pair of rules is unwanted. To this end, partial expressions fix this later: we 1) ensure all merges for a rule are in the same table (§7.4.3), and 2) disallow conflicting placements within the same table (§7.4.5).

$$\begin{aligned}&\forall r \in \text{ruleset}:\\ &fm_r \leftrightarrow \big(\textsc{fully-merged-preceding}(r) \vee \textsc{fully-merged-following}(r)\big)\end{aligned} \tag{7.10}$$

For all input rules, Equation (7.10) creates a new boolean variable, fm_r, in the expression to represent a fully merged rule, r. fm_r is true if, and only if, the corresponding rule is fully merged with all preceding rules, all following rules, or both.

These equations necessarily allow partially merged rules; otherwise, any fully-merged rule requires all other rules to be fully-merged also.

7.4.3 Ensure all Direct and Merge Placements Rules are in the Same Table

If a rule is only partially merged, a direct placement of the original rule must be placed under it in the same table to catch the unmatched portions of traffic to maintain forwarding. The solver requires all merge and direct placements for a rule to be in the same table. Therefore, the direct placement will capture all traffic missed by the merged placements. The same table requirement also applies if no direct placement is made, thus ensuring all merge placements are in the same table. This check does not consider split placements.

$$\begin{aligned}&\forall r \in \text{ruleset}:\\ &\quad \forall t \in \text{direct}(r) \cup \text{merge}(r):\\ &\qquad t \rightarrow ft_{r_{table}}\end{aligned} \tag{7.11}$$

Equation (7.11) links direct or merge transformations to their corresponding ft variable. The solver creates a unique ft variable for each combination of rule, r, and the table of the rule's placement, *table*. ft is linked using implies; otherwise, selecting one transformation would force selecting all other placements in the table for a rule.

$$\forall r \in \text{ruleset}: \qquad \text{AMO}(\{ft_{r_0} ... ft_{r_n}\}) \tag{7.12}$$

For each rule, r, Equation (7.12) creates a partial expression that constrains a rule's direct and merge transformations to the same table, by allowing at most one true ft_{r_x} variable.

Note that in this case, it is unnecessary to put the counter clause in for Equation (7.11) to ensure ft_{r_x} is only true if at least one corresponding transformation is selected. Because AMO will naturally force ft variables to be false in order to be satisfied.

7.4.4 Placement Variables

Thus far, the partial expression selects one transformation for all rules in the input ruleset, to ensure that the output ruleset includes all forwarding behaviour. However, many of these transformations will place the same or similar rules in the output ruleset. This section defines new 'placement' variables in the SAT expression, which represent the rules a transformation places into the solution ruleset. The following sections add partial expressions using these placement variables to prohibit invalid or redundant placement combinations.

It is common for transformations to create the same or similar placements, even those created from different rules. To understand why this happens, consider a multi-table ruleset, every table will have a table-miss rule. Each table-miss rule often only differs by its table, so for each, the solver will create the same transformations into all possible tables in the target pipeline. Additionally, a table-miss rule often acts as an identity to the merge operation. For different rules, split transformations often create the same rule to pass through a table without applying any actions.

To constrain placements, the solver introduces a new variable to represent each concrete placement. The rule-fitting solver maps each transformation to its corresponding concrete placements.

$$\begin{aligned}&\forall t \in \text{transformations}: \\ &\quad \forall p \in \text{placements of } t: \\ &\qquad t \rightarrow p \qquad (7.13)\end{aligned}$$

Equation (7.13) links, using implication, each transformation t, to its corresponding placements, p. A placement p represents a unique rule in the final ruleset with the same match, priority, table, and instructions.

Equation (7.13) is not sufficient by itself, as a placement can be true without any corresponding transformations selected. So to stop the SAT solver from selecting a placement without corresponding transformations, the rule-fitting solver adds a clause for each placement, p, implying that at least one corresponding transformation is selected.

$$\begin{aligned}&\forall p \in \text{placements}: \\ &\qquad p \rightarrow \bigvee\{t \in \text{transformations } \textbf{where } p \in \text{placements of } t\} \qquad (7.14)\end{aligned}$$

Equation (7.14) adds a partial expression for each placement variable, p, to ensure it is true only when at least one corresponding transformations, t, are true.

7.4.5 Disallow Same-Priority Conflicting Placements

In OpenFlow, rules at the same priority with overlapping matches but with different instructions and actions have undefined forwarding behaviour. The reason for which is simple: it is unclear which should take priority in this situation. So the rule-fitting solver adds constraints to the SAT problem so that it is unsatisfiable when conflicting placements are present.

$$\begin{aligned}
&\forall p_1 \in \text{placements}, \forall p_2 \in \text{placements } \textbf{where} \\
&\qquad p_1 \neq p_2 \wedge \\
&\qquad \textsc{priority}(p_1) = \textsc{priority}(p_2) \wedge \\
&\qquad \textsc{table}(p_1) = \textsc{table}(p_2) \wedge \\
&\qquad \textsc{match}(p_1) \cap \textsc{match}(p_2) \neq \emptyset \wedge \\
&\qquad \textsc{instructions}(p_1) \neq \textsc{instructions}(p_2) : \\
&\qquad\qquad \neg p_1 \vee \neg p_2
\end{aligned} \tag{7.15}$$

Equation (7.15) shows the partial expressions the solver generates to disallow such conflicts. The equation considers all pairs of rules at the same priority with an intersecting match, and disallows any combination with different instructions.

7.4.6 Disallow Placements with Conflicting Instructions

The transformations the solver uses to build a solution derive their placement priorities from the priority of the original rule. In the case of split and direct placements, the priority is the scaled priority of the original rule, and for merge placements, it is the sum of the scaled priorities. As a result, it is common for the SAT solver to return placements with overlapping matches but different priorities within the same table. Only the highest priority placement is actually 'hit' by packets. All of these placements except the highest priority placement shadowed.

Thus the placement hit must be a valid substitute for any placement it shadows; otherwise, the solution loses the forwarding behaviour of the shadowed placement. The rule-fitting solver generates split transformations for all valid placements in a table, even considering those with the 'wrong' placement

(§6.3.6). Therefore, for all valid substitute placements, a split transformation exists. If such a transformation does not exist with a given placement, then it is improbable that it is a valid substitute, so the rule-fitting solver can avoid exploring this solution.

Therefore, the rule-fitting solver adds partial expressions to disallow a combination of placements within the same table with the same match but different instructions or actions.

$$\begin{aligned}
&\forall p_{lo} \in \text{placements}, \forall p_{hi} \in \text{placements}\ \textbf{where}\\
&\qquad \textsc{priority}(p_{lo}) < \textsc{priority}(p_{hi}) \wedge\\
&\qquad \textsc{table}(p_{lo}) = \textsc{table}(p_{hi}) \wedge\\
&\qquad \textsc{match}(p_{lo}) \subseteq \textsc{match}(p_{hi}) \wedge\\
&\qquad \textsc{instructions}(p_{lo}) \neq \textsc{instructions}(p_{hi}) :\\
&\qquad\qquad \neg(p_{lo} \wedge p_{hi})
\end{aligned} \tag{7.16}$$

For all conflicting, fully shadowed placements, p_{lo}, and the corresponding higher priority rule, p_{hi}, Equation (7.16) creates a partial expression to disallow both placements. As a result, the final SAT expression is only satisfied when all shadowed placements in the same table matches have the same instructions as the placement hit instead.

The subtle difference between this restriction and placements at the same priority (§7.4.5) is that partial overlaps at the same priority are invalid at the same priority.

7.4.7 Require a Table-Miss Rule

In OpenFlow 1.3, a table-miss rule is placed at the lowest priority and matches all packets not matched by other rules in that table. A table-miss rule defines forwarding using instructions just as any other rule does. If a table-miss rule is not included, the default behaviour is to drop these unmatched packets. Note, that OpenFlow switches may allow an operator to override this default behaviour for a switch globally.

While it is valid not to install a table-miss rule and rely on the default behaviour of a switch, it is better to install these rules explicitly. The advantage of explicit table-miss rules is two-fold: 1) it does not rely on the default global table-miss action for a switch, and 2) it reduces the search space in the SAT solver. The search space reduction is because a ruleset without a table-miss rule is the same as a ruleset with a table-miss drop rule explicitly installed.

To this end, the rule-fitting solver first creates boolean variables to track the tables in the target pipeline that packets reach. Then the rule-fitting solver adds a constraint that requires a table-miss rule in all reached tables.

$$\begin{aligned}
&\forall p \in \{\text{placements } \textbf{where } p \text{ has a goto instruction}\} : \\
&\qquad\qquad p \rightarrow tr_x \ \textbf{where } tr_x \text{ represents the next table } x \\
&\textbf{and} \\
&\qquad\qquad tr_0 \qquad \triangleright \text{ table 0 is always reached}
\end{aligned} \tag{7.17}$$

Equation (7.17) maps every placement to the corresponding table it goes to, tr_x. Thus, when true, tr_x represents that packets reach table x. Additionally, because all packets enter the pipeline at the first table, its corresponding variable tr_0 must always be true.

$$\begin{aligned}
&\forall tr_x \in \text{tables-reached} : \\
&\qquad tr_x \rightarrow \bigvee \{p \in \text{table-miss placements } \textbf{where } \textsc{table}(p) = x\}
\end{aligned} \tag{7.18}$$

With table variables defined, now the rule-fitting solver can require a table-miss rule in every table packets reach. Equation (7.18) adds partial expression that requires reachable tables to a have a table-miss rule.

7.4.8 Hit Placement Variables

Combinations of placements can overlap each other, such that a high priority rule completely shadows a lower priority rule. In this situation, the lower priority rule is redundant, as packets never reach it and so it does not affect forwarding.

Within a table, if multiple placements have the same match, the highest priority rule determines forwarding, we say this highest priority placement is 'hit'. Including any combination of shadowed placements in the ruleset does not affect forwarding. Therefore, the solver should not consider the same combination of hit placements more than once, as only the hit placements determine forwarding.

It is not possible to disallow all shadowed rules, as every input rule requires a transformation (§7.4.6), a condition the solver might only be able to satisfy with a shadowed placement. So instead, the rule-fitting solver adds variables to the expression which represent the hit placements, to avoid rechecking equivalent solutions.

For each placement, the solver introduces a new variable to signify the placement can be 'hit', i.e. the highest priority rule in a table with a given match. To explain the logic we require, consider the placements p_1, p_2, p_3, and p_4 all within the same table with the same match but at different priorities, where p_1 has the highest priority and p_4 the lowest priority. For every placement p_x, there is a corresponding hit placement variable h_x set only if p_x is selected and the highest priority rule. Therefore, if p_1 is selected it always sets

h_1, however, h_3 is only set if p_1 and p_2 are not selected but p_3 is.

$$\forall p_x \in \text{placements} :$$
$$\{p_0, p_1 ... p_n\} \in \text{placements } \textbf{where}$$
$$\text{MATCH}(p_n) = \text{MATCH}(p_x) \, \wedge$$
$$\text{TABLE}(p_n) = \text{TABLE}(p_x) \, \wedge$$
$$\text{PRIORITY}(p_n) > \text{PRIORITY}(p_x) :$$
$$(p_x \wedge \neg p_0 \wedge \neg p_1 \wedge ... \wedge \neg p_n) \leftrightarrow h_x \quad (7.19)$$

For each placement, p_x, Equation (7.19) creates a partial expression that creates and maps the corresponding hit placement variable h_x to be true only when 'hit'. The partial expression assigns the hit placement variable (h_x) to true when the original placement (p_x) is true, but no higher priority placements that shadow it are true ($\{p_0, p_1 ... p_n\}$).

A unique set of hit placements is a unique solution, which could have different forwarding behaviour to another. Thus the solver, on subsequent iterations, adds a clause to ensure that it will not consider the same combination of hit placements again.

7.4.9 Built-in Rules

A Table Type Pattern might optionally specify built-in rules. Built-in rules are most often used to define table-miss behaviour. The solver maps built-in rules into the SAT problem like it does for direct transformations but ensures they are always selected. Built-in rules are not considered in conflicting instructions (§7.4.6) as they cannot be removed and therefore, can only be overridden by a rule with conflicting instructions.

7.5 Solving and Building the Solution Ruleset

The rule-fitting solver combines all partial expressions in the previous sections using AND to create the complete SAT expression. The rule-fitting solver expresses the SAT problem in the CNF DIMACS format, a format compatible with the majority of SAT solvers, allowing it to use any off-the-shelf SAT solver. In our implementation, we use the MiniSat2 SAT solver API [50], which allows us to add constraints incrementally, which improves performance on subsequent runs.

The SAT solver either returns a solution, or that the boolean expression is unsatisfiable. If unsatisfiable, the solver can not find a solution to the rule-fitting problem, so it exits. Otherwise, the SAT solver returns a solution; this solution has all boolean variables assigned to a concrete value, either true or false.

The rule-fitting solver collects all selected transformations and constructs the corresponding candidate ruleset. The solver then converts this candidate ruleset into a canonical Multi-Terminal Binary Decision Diagram (MTBDD) for equivalence checking using the technique described in Chapter 4. If the forwarding is equivalent to the input ruleset, the rule-fitting solver has found a valid solution which it can return.

Otherwise, the candidate ruleset was not equivalent, so the rule-fitting solver refines the SAT expression and reruns the SAT solver. The rule-fitting solver repeats this process until it finds a valid solution or determines the problem is unsatisfiable. Section 7.6 describes the additional clauses the solver adds to the SAT expression before rerunning the solver.

7.6 Refining the SAT Expression

If the solution was not successful or the solver is searching for all possible solutions, the rule-fitting solver will rerun the SAT solver to find another solution. However, first, the rule-fitting solver needs to add a clause to the SAT problem to avoid rechecking the same solution (§7.6.1). Additionally, in the case that the solution returned was unsuccessful, the solver analyses what went wrong and adds clauses to prohibit that particular combination of placements again (§7.6.2).

7.6.1 Ensure the Same Solution is Not Returned Again

As described in Section 7.4.8, the hit placement variables represent a unique solution ruleset with regards to forwarding behaviour. Once a solution is checked, the rule-fitting solver adds a clause to stop the SAT solver returning that exact combination of hit placements again.

$$\forall h \in \text{hit-placements} : \bigvee \left\{ \begin{array}{ll} \neg h, & \textbf{if } h = True \\ h, & \textbf{if } h = False \end{array} \right\} \tag{7.20}$$

Equation (7.20) shows how the solver constructs a partial expression which avoids rechecking the same hit placements and, therefore, solution. The expression the solver constructs requires at least one hit placement change its value either from true to false or false to true.

7.6.2 Isolating Forwarding Conflicts

While requiring the SAT solver to select different hit placements ensures the rule-fitting solver makes progress with each iteration, this does not reduce the search space. Limiting the search space is particularly useful to solve problems with many rules as these often have intractable search spaces.

With an invalid solution, part of the packet-space will observe the correct forwarding, and the other part incorrect forwarding. Therefore, if we can add constraints to stop the SAT solver returning this incorrect forwarding, these constraints can drastically cut down the search space. Instead of the rule-fitting solver only adding a constraint that a specific combination of hit placements is incorrect, the solver adds more specific constraints which express particular placements are not compatible.

The solver calculates the symmetric difference between the forwarding behaviour of the input ruleset and the candidate ruleset. More concretely, the rule-fitting solver uses the BDD difference operation as described in §4.2.3. The rule-fitting solver maps this difference, currently expressed as a BDD, back to concrete paths in the original ruleset. The solver finds these concrete paths by computing rules which match this difference in the single-table version of the ruleset, accounting for priority. This rule-fitting solver has already created this single-table version of the ruleset as part of equivalence checking. Each rule in the single-table saves a copy of the rules which formed it, therefore the corresponding path in the original ruleset.

Then, using the same technique, for each path through the original ruleset, the solver calculates the corresponding paths in the candidate ruleset that packets incorrectly take.

The transformations the solver created have correct forwarding in isolation; therefore, this difference in forwarding is because either packets do not reach these placements, or do so in the wrong order. Placements are unreachable when shadowed by other placements or in an unreachable table. A shadowed placement can be resolved by either removing any conflicting higher priority placements or moving the unreachable placement to be reachable. If a placement is in an unreachable table, then that placement needs to be moved, or a split transformation selected that includes a rule to direct packets to that table. If placements differ due to being in the wrong order, then the solver needs to select a different set of placements for the original rules. In all cases, either the solver needs to pick new placements for the conflicting rules in the input ruleset path or remove the conflicting placements hit in the candidate ruleset.

Therefore, a conflicting path between the input and candidate ruleset may be resolved by either picking a different transformation for a rule in the input path or by removing the hit placement in the candidate ruleset. The solver constructs the following partial expression to represent this constraint.

```
$\forall path_i \in$ conflicting input paths :
    $\forall path_r \in$ conflicting candidate paths of $path_i$ :
        **Let** $\{t_1...\}$ = the selected transformations for the rules in $path_i$
    **Let** $\{h_1...\}$ = the hit placements corresponding to the rules in $path_r$
```

$$\neg\Big[\bigwedge\left[\{t_1...\}\cup\{h_1...\}\right]\Big] \tag{7.21}$$

Equation (7.21) shows how the rule-fitting solver generates a partial expression to prohibit a path in the input ruleset, $path_i$, conflicting with a corresponding path in the candidate ruleset, $path_r$. The solver generates a partial expression for all conflicting paths in the input and candidate ruleset, hence the $\forall$ iteration over all items in both sets of paths. Both paths are a sequence of rules through each pipeline. From the rules in $path_i$, Equation (7.21) builds the set of corresponding selected transformations ($\{t_1...\}$), i.e. those transformations set true in the candidate solution for a rule. Similarly, for the rules (aka placements) in $path_r$, Equation (7.21) builds the set of corresponding hit placements ($\{h_1...\}$).

The partial expression disallows all of these selected transformations and hit placements to be true in future solutions. Therefore, for the next solution, the SAT solver must either replace one input transformation or remove a hit placement.

Chapter 8

Evaluation

This chapter presents an evaluation of the performance of our implementation of the rule-fitting solver described in Chapters 5 to 7. This chapter evaluates both the time taken to find solutions and the number of valid solutions, if any, found by the rule-fitting solver.

By far, the most substantial challenge of this research was constraining the search space to a tractable size. Hence the main focus of this evaluation is on the optimisations and techniques we developed to better guide the rule-fitting solver to finding valid solutions quickly.

It did not make sense to perform a direct comparison to other algorithmic methods, such as FlowConvertor [37] and FlowAdapter [38], as their approach heavily relied on metadata which our approach specifically avoids. The types of problems FlowConvertor and FlowAdapter aimed to solve are distinctly different from the problems we aimed to solve with our technique, making any direct comparison unpractical.

First, Section 8.1 presents our evaluation methodology, including the rulesets and pipelines used for the evaluation (§8.1.1). Then Section 8.2 presents our evaluation of the usefulness of converting a ruleset to a single-table as a preprocessing step. Section 8.3 presents our evaluation of the performance improvement of ruleset compression. Section 8.4 presents an evaluation of the constraints we add to the SAT expression and their ability to reduce the

```
Original ruleset size: 20
 After ruleset_hook: 42 210.0%
 After pre_solve (compression etc.): 10 50.0%
 Post solve size: 8
 Solution ruleset size: 25 125.0%
Iterations: 10000
Solutions Checked: 10000
Valid Solutions: 8229
Unique Solutions: 7
Re-actioning Splits Added: 30
SAT Variables: 52
SAT Sln Variables: 46
SAT Clauses: 100
SAT Search Space List: 4, 5, 4, 5, 4, 5, 5, 4, 5, 5
SAT Search Space: 4000000
```

Figure 8.1: An example of the internal metrics the rule-fitting solver collects. The ruleset metrics include a count of the input ruleset size at different stages. In this example: ruleset hook is after conversion to a single-table (§4.2.1), pre solve is after ruleset compression (§6.2.3), post solve the solution size for the compressed ruleset, and the final solution size is after applying the solution back to the original ruleset. Iterations is the number of calls to the SAT solver, and, typically matches the number of solutions checked. The valid solutions metric is the number of solutions the SAT solver returned that are equivalent to the input forwarding. A difference between valid and unique solutions indicates the solver is unnecessarily rechecking the same solution; for this example the SAT constraints which normally prevent this are disabled. SAT solution variables are the number of variables which represent a unique solution and must change on the next iteration. The SAT search space is the number of transformation combinations, given only the primary constraint that the SAT solver picks one placement for each rule.

size of the problem without excluding valid solutions. Finally, Section 8.5 discusses the difficulties present in trying to fit a real-world ruleset to a real-world pipeline and the limitations of our approach.

8.1 Measurement Methodology

All performance testing was performed on a Ubuntu 16.04 machine, with an Intel i7-4790 @ 3.6Ghz (boost 4.0Ghz), with 8GB of RAM and the Linux 4.15 kernel. We limited the RAM available to the rule-fitting solver to 3.5GB. The rule-fitting solver is a single-threaded Python application which we ran on Python 2.7. Internally, the solver uses unordered data structures, and run-to-run may explore a different number of candidate solutions, due to the order candidate solution are returned influencing the refinement constraints the solver adds.

We used a script to collect timing results; the script repeated each test 10 times. Each test was preceded by an additional warm-up run so that accessed files were cached in memory to ensure the best consistency possible between runs. For timing results, we report the mean along with the 95% confidence interval.

We instrumented the rule-fitting solver to collect both extensive timing and internal metrics. Figure 8.1 gives an example of the internal metrics that the rule-fitting solver collects. These metrics include the size of the ruleset as it progresses through the solver, the number of solutions found, the number of iterations of the SAT solver, and metrics about the size of the SAT problem.

Figure 8.2 shows an example of the timing information that the rule-fitting solver collects. The solver reports wall-clock times for each major processing stage as a hierarchy. Times nested under another (parent) time are subtasks of the parent. A parents time includes the time of its subtasks. Not all nested times add to 100% because we only timed select tasks. The total time is measured from within the rule-fitting solver and therefore excludes the time to load Python and the libraries the rule-fitting solver imports. Our script additionally records the entire wall-clock of a run, and we have found this unrecorded overhead to be consistently 0.6s. As this overhead is constant and fundamentally uninteresting, we do not report it in the results shown in this chapter.

8.1.1 Pipelines and Rulesets for Evaluation

For the evaluation of the effectiveness of the ruleset preprocessing and the boolean satisfiability constraints, we constructed two pipelines and corresponding rulesets. We perform this analysis on these small rulesets, as without all

```
Total Runtime: 13.167580s
  Loading TTP: 0.001442s (0%)
  Loading Ruleset: 0.007606s (0%)
  Pre Solve: 0.001464s (0%)
    Compress Ruleset: 0.001446s (99%)
  Solver Init: 0.000494s (0%)
  Run Solver: 13.103029s (100%)
    Compute Dependencies: 0.000580s (0%)
    Generating Transformations: 0.019510s (0%)
      Split Placements: 0.014709s (75%)
      Direct Placements: 0.003125s (16%)
      Merge Placements: 0.000033s (0%)
      Compress Priorities: 0.000436s (2%)
      Re-actioning: 0.001133s (6%)
    Build SAT Expression: 0.001680s (0%)
    SAT Solving Time: 2.529629s (19%)
      Init SAT Solver: 0.004831s (0%)
    Solution Building: 2.329341s (18%)
    Solution Compare: 8.100654s (62%)
  Post Solve: 0.052857s (0%)
    Applying Model: 0.052793s (100%)
      Verifying Solution: 0.027161s (51%)
```

Figure 8.2: An example of the timing information the rule-fitting solver collects. Times are from a wall-clock and are reported in seconds. Each level of indentation represents the task occurs within the parent time; the associated percentage is relative to the parent. The total runtime is measured from within Python and does not include the time to load libraries.

SAT constraints or preprocessing optimisations the size of the problem grows immensely, and larger problems become intractable.

Figure 8.3 shows the first pipeline, the **5-table pipeline**, which we based on the OF-DPA bridging and routing pipeline. The only significant difference from the original OF-DPA pipeline is that it does not require a VLAN match in all of its tables. We have chosen this because the rule-fitting solver lacks the transformations to handle adding new VLANs to packets in this situation. We crafted the other pipeline, the **2-table pipeline**, shown in Figure 8.4, with a contrasting table layout while still supporting the same forwarding. To convert rules between these two pipelines, the rule-fitting solver must make significant transformations to the opposing ruleset.

Our 5-table pipeline only retains the tables necessary for switching and

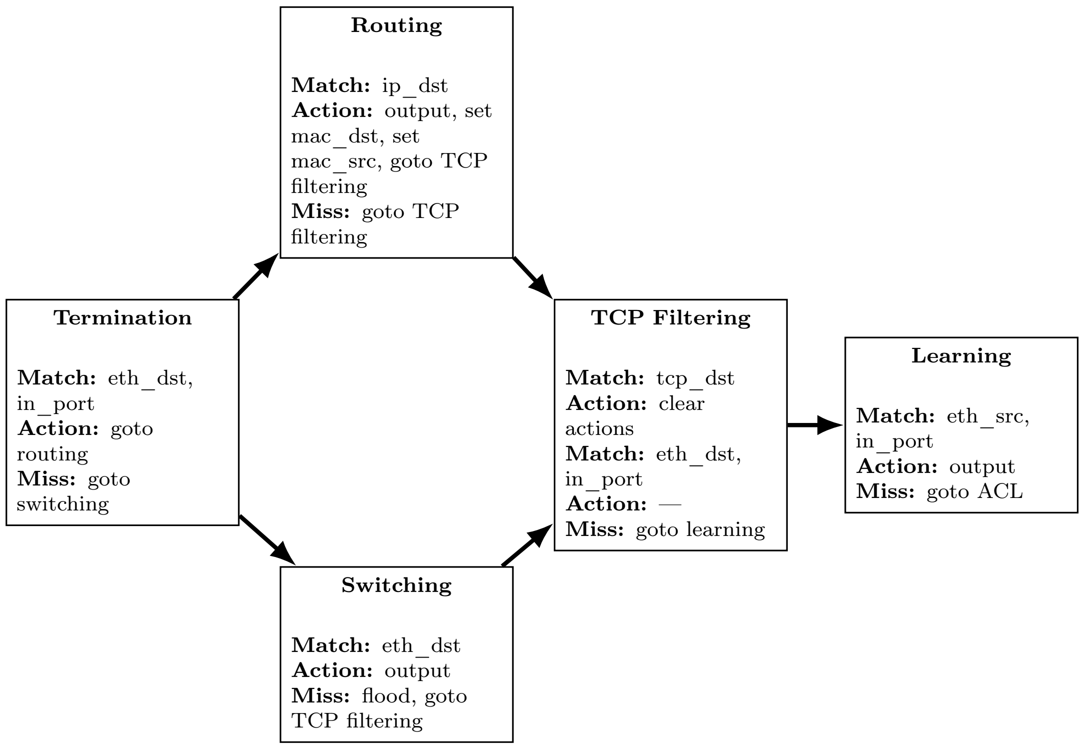
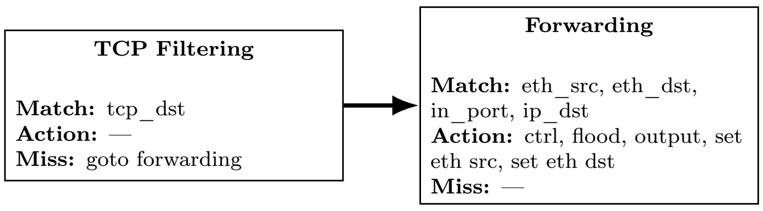


Figure 8.3: **5-table pipeline**: A bridging and routing pipeline which uses write and clear-actions. This pipeline is based on the OF-DPA pipeline and represents its complexity without packets requiring a VLAN to traverse the tables. All rules must write actions to the packets action set so the TCP filtering table can reverse any forwarding decision. The output to controller action in the learning table is the one exception which is applied immediately. The first table, termination, splits packets into either the routing or switching tables based on Ethernet destination. We placed the learning table at the end of the pipeline so that the pipeline does not send filtered packets to the controller. Note: the learning table is a special table in OF-DPA that synchronises its entries from the switching table, we define it explicitly as our solver cannot handle this non-standard behaviour.

routing from the OF-DPA pipeline. The 5-table pipeline still includes complexity; it uses a separate table for each different network function. The pipeline splits into two parallel paths, one for routing and the other for switching. The routing and switching tables recombine for filtering. Another complexity of the 5-table pipeline is that it uses write-actions to store forwarding decisions in a set against each packet until the final table executes these actions. Rules can drop (filter) unwanted packets by clearing this action set. Due to this action set, to fit rules to this pipeline, the solver must find the correct 'wrong' action for rules and remove it later in the pipeline as described in Section 6.3.6.



TCP Filtering
Match: tcp_dst
Action: —
Miss: goto forwarding
Forwarding
Match: eth_src, eth_dst, in_port, ip_dst
Action: ctrl, flood, output, set eth src, set eth dst
Miss: —

Figure 8.4: **2-table pipeline**: A simple two-table pipeline compatible with the 5-table OF-DPA based pipeline shown in Figure 8.3. We designed the pipeline to contrast with the 5-table pipeline such that conversion between pipelines requires the extensive transformation of the ruleset. The pipeline performs all forwarding in the second table using apply-actions, rather than using write-actions spread over multiple tables as in the 5-table pipeline. The first table filters unwanted packets by applying no action, rather than the second to last table in the 5-table pipeline, which drops packets using clear-actions.

Additionally, we have exposed the learning table at the end of the 5-table pipeline. The OF-DPA learning table synchronises with the switching table and sends packets from unknown hosts to the controller for learning. Broadcom's OF-DPA TTP description does not include the learning table because OpenFlow 1.3 cannot represent a synchronised table. The OF-DPA documentation does not specify the exact point the pipeline send packets for learning to the controller. It is reasonable to assume the pipeline would filter packets before sending them to the controller. Thus, we have placed the learning table at the end of the pipeline, after filtering, to match this assumption.

The 2-table pipeline moves the filtering table to the start of the pipeline, which is a more natural location. Then we elected to combine all forwarding decisions into one table as this requires the rule-fitting solver to perform extensive splitting or merging of rules to convert between these rulesets.

Figure 8.5 shows the ruleset for the 5-table pipeline we use for our testing. This ruleset contains 20 rules. Because we performed most of our experiments with ruleset compression enabled, the size of the input ruleset is not a significant factor. With compression enabled, all similar rules are compressed into one regardless of the ruleset size. For our evaluation of ruleset compression, we vary the size of this ruleset. We vary the size of the ruleset by replacing the first three rules in the switching and learning tables with the number corresponding

Termination	
Match	Action
eth_dst	goto
...:01	Routing
...:02	Routing
—	Switching

Routing		
Match	Write Actions	
ip_dst	output	goto
1.0.0.0/8	20	filtering
10.0.0.0/8	20	filtering
—	21	filtering
—	—	filtering

Switching		
Match	Write Actions	
eth_dst	output	goto
...:0a	10	filtering
...:0b	11	filtering
...:0c	12	filtering
—	flood	filtering

TCP filtering			
Match		Write Actions	
tcp_dst	eth_dst	clear-actions	goto
80	—	yes	—
443	—	yes	—
—	...:01	no	—
—	...:02	no	—
—	—	no	learning

Learning		
Match		Apply Actions
eth_src	in_port	output
...:0a	10	—
...:0b	11	—
...:0c	12	—
—	—	controller

Figure 8.5: The 5-table ruleset corresponding to the 5-table pipeline we used in our evaluation of the rule-fitting solver. The first table of the ruleset is the termination table. All rules are listed top to bottom from highest to lowest priority. In addition to the actions shown, the rules in the routing table rewrite the Ethernet source and destination. In the routing table, the third rule provides a default route which overrides the fourth rule, which is the built-in table-miss rule. This ruleset compresses well, all rules with the same match and actions compress to a single rule.

to the hosts learnt for that experiment.

We derived the ruleset for the 2-table from the 5-table ruleset by putting the two TCP filtering rules in the first table. Then we merged all other paths through this ruleset into single rules placed in the second table. The 2-table equivalence of this ruleset contains 25 rules.

8.2 Evaluation of Single-Table Preprocessing

This section evaluates the usefulness of converting a ruleset to an equivalent single-table as a preprocessing step in the rule-fitting solver as described in Section 6.2.1. There is no additional overhead for the rule-fitting solver to convert a ruleset to a single-table, as the rule-fitting solver already converts the ruleset to a single-table to check its equivalence.

In theory, the key advantages to preprocessing the input ruleset to a single-table are that every rule represents the complete end-to-end forwarding behaviour of the pipeline and the rule-fitting solver does not need to generate merge

Input	Total Runtime (ms)		Ruleset Size	Solutions	Iterations	Search Space
Multi-Table	139	±2%	8	1	4	96
Single-Table	134	±1%	7	1	4	16

Table 8.1: A performance comparison of the rule-fitting solver between a multi-table and single-table input ruleset when converting from the 2-table ruleset to the 5-table pipeline. The runtime is the total time from loading the inputs to generating an output; the ruleset size is the input ruleset size after compression. The search space is the number of possible combinations of transformations if you pick one for each rule. There is very little difference between fitting the multi-table input and the single-table input. This lack of difference is because the 2-table ruleset puts all forwarding (the majority of its complexity) in one table, so the multi-table input is very similar to the single-table ruleset.

transformations. However, a single-table ruleset typically contains more rules than its multi-table equivalent, which would take longer to process. In contrast, the key advantage of using a multi-table input ruleset is that it retains the logical separation of network functions as rules remain split between tables which can guide the solver.

We evaluate the advantages and disadvantages of preprocessing to a single-table compared to the original input by converting between the 2-table and 5-table pipelines (§8.1.1). For this experiment, we configured the rule-fitting solver with ruleset compression enabled and to generate all possible solutions. The rule-fitting solver compresses the ruleset after converting it to a single-table. We ran each test 10 times. In this discussion we refer to the experiment with preprocessing to a single-table enabled, simply as using single-table input and with this preprocessing disabled, using a multi-table input. Note however, this conversion happens in the ruleset preprocessing stage of the rule-fitting solver.

Tables 8.1 and 8.2 show a performance comparison of the rule-fitting solver between a multi-table and single-table input ruleset. Table 8.1 shows the results for converting the 2-table ruleset to the 5-table pipeline, and Table 8.2 shows the results for converting the 5-table ruleset to the 2-table pipeline.

In both cases, the single-table ruleset includes fewer rules than the original ruleset. While we expect the equivalent single-table ruleset to be larger, ruleset

Input	Total Runtime (ms)		Ruleset Size	Solutions	Iterations	Search Space
Multi-Table	1902	±9%	12	0	558	352800000
Single-Table	125	±2%	10	10	10	4000000

Table 8.2: A performance comparison of the rule-fitting solver between a multi-table and single-table input ruleset when converting from the 5-table ruleset to the 2-table pipeline. With multi-table input, the rule-fitting solver does not find a solution and takes longer than with the single-table input to complete. This longer time is due to the larger search space, from considering more transformations of rules, which requires more iterations of the SAT solver to explore fully.

compression was able to reduce the ruleset size more than with the multi-table input. For rulesets with more complex relationships between rules, the compressed single-table might still be larger than a compressed multi-table ruleset. Anecdotally, we have found that even with more complex rulesets compression reduces the expansion from converting a ruleset to a single-table to manageable levels.

Table 8.1 shows little difference between the single-table and multi-table inputs when converting the 2-table ruleset to the 5-table pipeline. The results are similar because the 2-table ruleset has the majority of its logic condensed into one table, so the rule-fitting generates transformations which are very similar to the single-table input.

Table 8.2 shows a significant performance difference between the single-table and multi-table input, a mean total runtime of 125ms compared with 1.9s. Also, the rule-fitting solver does not find a solution for the multi-table input, but does for the single-table input.

Merge transformations primarily explain the increased runtime with a multi-table input. The first stage of the rule-fitting solver cannot predict which transformations a solution requires, so it generates all possible transformations. Generating these merge transformations can get expensive for a ruleset with many tables. The longest path a packet can take through the 5-table pipeline is four tables long. The first stage of the solver must generate merge transformations for rules between two adjacent tables in this path, then for rules across three tables, and then finally four. The final stage of the solver must search this larger number of transformations which explains the increase in the runtime and iterations of the solver.

The lack of a solution with a multi-table input is most likely due to individual rules not representing the complete forwarding behaviour, but this is difficult to verify. Because transformations only represent partial forwarding, it is harder for the second stage of the solver to find the correct combination. Additionally, the rule-fitting solver does not consider combinations of split and merge transformations, which misses searching rule transformations with a multi-table input. Ignoring these combinations does not affect a single-table input as there are no merge transformations to consider.

Overall we have found that single-table input is easier to fit into a new pipeline. We have found the increased ruleset size of a single-table is mitigated by ruleset compression, and that merge transformations for multi-table inputs become unwieldy for long pipelines.

8.3 Evaluation of Ruleset Compression

This section evaluates the performance trade-off of ruleset compression, as described in Section 6.2.3. With compression enabled, the rule-fitting solver compresses the ruleset into a form with fewer rules while maintaining the interactions between those rules. Compression groups similar rules from the input ruleset into one representative rule. Once the rule-fitting solver finds a valid solution for the compressed ruleset, the solver applies this solution to the original ruleset.

In theory, compressing a ruleset reduces the size of the problem the solver is working with, which should be faster. However, compressing a ruleset incurs the additional overhead of compressing the ruleset and applying the solution found back to the original ruleset. This section quantifies this trade-off between the overhead of compressing a ruleset and the decreased time to solve the problem.

To quantify the trade-off, we designed and ran an experiment to compare the performance of the rule-fitting solver with and without compression for rulesets of different sizes. The experiment compared the performance of fitting the 5-table ruleset back into the 5-table pipeline (§8.1.1). The rule-fitting solver was configured to convert the pipeline to a single-table as part of preprocessing, thus making the task of fitting to the same pipeline non-trivial. The rule-fitting solver converted the ruleset to a single-table before compressing it.

We changed the size of the ruleset by varying the number of learnt hosts in the pipeline. For each host learnt, we placed a corresponding rule in both the Switching and Learning table. Because the solver generates the single-table ruleset from the Cartesian product of all tables, the number of rules in the preprocessed single-table ruleset scales with the number of hosts squared.

In addition to the host rules, the ruleset contained a fixed number routing and filtering rules to represent complexity. The termination table contained two rules which directed two Ethernet destinations to the routing table. The routing table contained two /24 routes and a default. The filtering table contained two rules which dropped packets on TCP destination port, and two rules corresponding to the termination table to stop learning routed packets.

Our experiment collected metrics from the rule-fitting solver when solving a problem with between 0 and 50 learnt hosts with compression both enabled and disabled. We configured the solver to find the first valid solution, rather than all solutions. We only take the first solution because an uncompressed ruleset can generate a large number of solutions, from picking different transformations for similar rules. We ran each test ten times.

Figure 8.6 plots the total runtime for the experiment comparing the performance of the rule-fitting solver with and without compressing the ruleset. In all cases, the rule-fitting solver is faster when it compresses the ruleset first when compared with using the uncompressed ruleset. Consider the ruleset with zero hosts learnt, therefore when the fewest rules can be compressed, and the relative overhead of compress will be its highest. This ruleset still con-

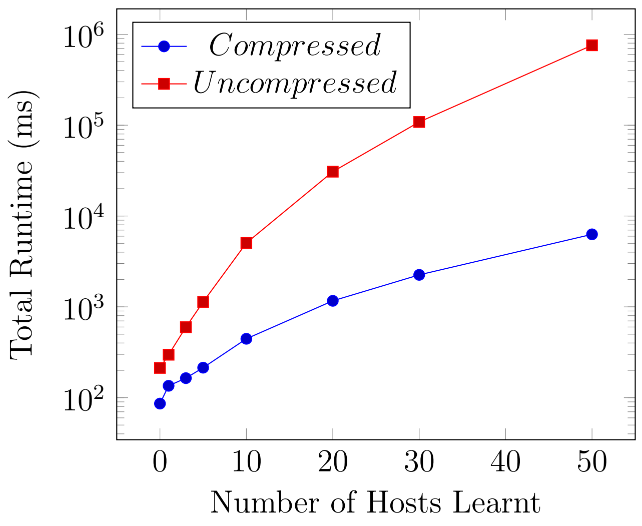


Figure 8.6: The total runtime of the rule-fitting solver to find the first valid solution fitting a variable-sized 5-table ruleset, first merged to a single-table, back into the 5-table pipeline. The y-axis reports the rule-fitting solver's mean total runtime in milliseconds on a log scale. The x-axis reports the number of hosts learnt in the ruleset. For each host learnt, two rules are added to the ruleset. The compressed ruleset significantly outperforms the uncompressed ruleset in all cases. With zero hosts learnt, the rule-fitting solver finds a solution to the compressed ruleset in 86ms compared to 213ms for the uncompressed ruleset. The difference is much higher with 50 hosts learnt, with the compressed ruleset taking 6.1s compared to 12min 38s when uncompressed.

tained two fixed routing and filtering rules which the solver compressed from two rules in each table down to one rule. The overhead of compression for this ruleset was insignificant and took 0.55 ms of the total time of 86ms while applying the solution back to the original ruleset took 11ms.

Table 8.3 shows a breakdown of where the rule-fitting solver spent time for the compressed rulesets and Table 8.4 shows the same breakdown for uncompressed rulesets. Comparing the one-off costs between the two, the combined time of compression and initialisation for a compressed ruleset is always significantly less than the initialisation time of the uncompressed ruleset. The iteration time for the compressed ruleset is constant and always lower than without ruleset compression.

The time to apply the compressed ruleset back to the uncompressed input is a one-off cost for a compressed ruleset, but an uncompressed ruleset incurs a similar amount of work on every iteration. Applying the compressed ruleset

Test	Timing (ms)					
Hosts Learnt	Total	Compress	Init	Iter	Apply	Iterations
0	86 ±1%	0.55 ±0%	56 ±0%	3.1 ±1%	11 ±1%	1
1	135 ±1%	0.79 ±9%	89 ±1%	8.8 ±2%	18 ±2%	1
3	164 ±1%	1.5 ±4%	89 ±1%	8.7 ±3%	43 ±1%	1
5	214 ±1%	2.9 ±2%	89 ±1%	9.0 ±4%	85 ±2%	1
10	445 ±5%	12 ±5%	91 ±4%	9.5 ±1%	287 ±7%	1
20	1165 ±2%	72 ±5%	94 ±4%	10 ±4%	907 ±2%	1
30	2250 ±1%	236 ±1%	89 ±1%	9.2 ±1%	1789 ±1%	1
50	6285 ±1%	1363 ±3%	90 ±1%	10.0 ±2%	4553 ±1%	1

Table 8.3: Distribution of rule-fitting solver time with ruleset compression enabled. The initialisation time is the one-off time taken to find transformations and construct the SAT problem. Iteration time is the cumulative time spent running the SAT solver and checking solutions. In all cases, the rule-fitting solver found a valid solution to every problem after one iteration. Other than with zero hosts learnt, the initialisation and iteration time is identical as the compressed ruleset always contains the same number of rules. With no hosts learnt the compressed ruleset is smaller as it does contain any rules to represent learnt hosts and therefore is solved faster. The difference between the total time and sum of all other times listed is the time to load the ruleset and Table Type Pattern from disk.

back to the input ruleset firsts builds the output ruleset based on the solution to the compressed ruleset and then verifies the output ruleset is equivalent to the original. With an uncompressed ruleset, every SAT solver iteration incurs a similar cost as it must build and verify the full-sized ruleset. A compressed ruleset still incurs a cost per iteration, but the cost is much smaller because the ruleset is smaller.

Thus, it is not surprising that the results show the time spent solving and applying a compressed ruleset is always lower than the time spent solving the same uncompressed ruleset. Table 8.4 shows that uncompressed rulesets required more iterations to solve, three iterations with 50 hosts learnt, compared to the compressed ruleset, which was always solved on the first iteration.

The number of iterations for an uncompressed ruleset is larger because with more rules and therefore transformations there is a higher chance of conflicts. Much of the iteration time with the larger uncompressed rulesets comes from detecting and adding SAT clauses to prevent these conflicts. The rule-fitting solver spent 91% of its total time detecting and adding SAT clauses when

Test	Timing (ms)					
Hosts Learnt	Total	Compress	Init	Iter	Apply	Iterations
0	213 ±1%	—	188 ±1%	8.5 ±1%	—	1
1	297 ±5%	—	240 ±5%	38 ±12%	—	2
3	598 ±1%	—	388 ±1%	186 ±1%	—	3
5	1133 ±1%	—	653 ±1%	450 ±2%	—	3
10	5048 ±2%	—	1881 ±1%	3120 ±2%	—	3
20	30765 ±1%	—	6087 ±1%	24588 ±1%	—	3
30	108454 ±1%	—	12759 ±2%	95548 ±1%	—	3
50	758012 ±1%	—	34811 ±1%	722886 ±1%	—	3

Table 8.4: Distribution of rule-fitting solver time with an uncompressed ruleset. Overall increasing the number of hosts and therefore rules in the ruleset increased the time the rule-fitting solver took to find a valid solution. For the smaller problems with five or fewer hosts learnt, the one-off initialisation costs exceed the iteration time spent checking candidate solutions. For ten or more hosts, the iteration time exceeds the initialisation time. These larger rulesets produce larger candidate rulesets, hence the time to build and verify the solutions equivalence increases on each iteration. Additionally, there are more iterations: iterations increases from 1 to 3. Therefore three candidate solutions needed to be built and verified before the solver found a valid solution. A significant portion of the iteration time comes from refining the SAT problem, which significantly reduces the iterations required to find a valid solution.

	Original Multi-Table		Single-Table	
Ruleset	Uncompressed	Compressed	Uncompressed	Compressed
Faucet Access	1,937	70	3,901	94
Faucet Router	582	360	5,281	902

Table 8.5: The results of compressing real-world Faucet rulesets. The original rulesets use multiple tables, and we compare the number of rules in each ruleset when compressed and uncompressed. Additionally, we compare the results of compression on each ruleset when first converted to a single table.

fitting the uncompressed ruleset with 50 hosts learnt.

Overall, we have found that the time to compress a ruleset is insignificant compared to the time it saves when computing a solution. Even if a ruleset is incompressible, the overhead of attempting would be insignificant. We have always found ruleset compression to be beneficial, in many cases improving the performance of the solver by orders of magnitude. For example, the solver found a valid solution when fitting the ruleset with 50 learnt hosts in 12min 38s when uncompressed compared to 6.1s when compressed.

8.3.1 Compression of Real-World Rulesets

This section presents the compression ratio achieved on two real-world rulesets. We use the same two rulesets as we used in the evaluation of the equivalence checking (Section 4.3.3). Faucet Router and Faucet Access as used in the evaluation of the equivalence checking. Faucet Router and Faucet Access were collected from two OpenFlow switches in a real-world enterprise deployment [42] which were programmed by the Faucet [7] controller. The Faucet controller was configured to perform VLAN switching, IPv4/6 routing, and stateless firewalling. Faucet Router has more complexity than Faucet Access as it was connected to the upstream and carried routes. Faucet Access does not carry routes, but had a larger ruleset due to having more ports, each with a stateless firewall policy applied.

Table 8.5 shows the reduction in ruleset size achieved by running the compression algorithm on these two real-world Faucet rulesets. The compression achieved on the Faucet Access ruleset was substantial, the multi-table ruleset compressed to 3.6% of its original size, while the single-table ruleset compressed to 2.4% of the single-table size. For the Faucet Access, the compressed single-table ruleset is only 34% larger than the original compressed ruleset.

Faucet Router did not compress as well, due to additional complexity in its pipeline. Compression decreased the Faucet Router multi-table ruleset to 67% of its original size, and the single-table ruleset to 17% of its original size. Overall, these results show that our ruleset compression technique can real-world rulesets, despite their complexity.

8.4 Evaluation of SAT Constraints

This section presents an evaluation of the effectiveness of the SAT constraints described in Chapter 7. Our evaluation considers both the impact on performance and the solutions returned to ensure a constraint does not exclude a valid solution. The evaluation started from the least constrained SAT problem (with the solutions to explore): picking one transformation for each rule. Then we introduced one constraint at a time until all constraints described in Chapter 7 were added. Our evaluation script repeats each test 10 times and collects the metrics from the solver for comparison, as described in Section 8.1.

Following is the list of SAT constraints we add, starting from the least constrained problem and cumulatively adding constraints. The remainder of this section uses the **bolded** friendly name to reference the cumulative constraints.

One Transformation (§7.4.1 to 7.4.3): The least constrained problem possible. Constrained to pick only one transformation per input rule, this includes the SAT constraints to identify fully-merged rules and place merged rules in the same table. The SAT solver picks a different combination of transformations each iteration.

Placements (§7.4.4): In addition, links transformations to their corresponding placements in the SAT expression. The solver picks a different combination of placements each iteration.

Placement Conflicts (§7.4.5 and 7.4.6): In addition, adds constraints to prevent overlapping placements at the same priority with conflicting instructions and to prevent fully-shadowed rules with the conflicting instructions. The solver picks a different combination of placements each iteration.

Table-Miss (§7.4.7): In addition, adds a constraint to ensure that every table with a rule installed includes a table-miss rule. Without an explicit table-miss rule the solver assumes the a default drop behaviour. The solver picks a different combination of placements each iteration.

Hit Placements (§7.4.8): In addition, links variables to represent the placements which are actually hit by packets. The solver picks a different combination of hit placements each iteration.

Constraints (Cumulative)	Timing (ms)				Iterations
	Total	Build SAT	Solve SAT	Verify	
One Transformation	13323 ±1%	6.7 ±4%	2325 ±2%	10774 ±1%	10000[a]
Placements	439 ±3%	8.3 ±4%	60 ±16%	262 ±3%	240
Placement Conflicts	200 ±2%	8.3 ±5%	19 ±12%	65 ±1%	64
Table-Miss	187 ±3%	8.7 ±5%	15 ±10%	57 ±4%	49
Hit Placements	134 ±6%	10 ±16%	4.0 ±10%	14 ±10%	10
Forwarding Conflicts	128 ±3%	9.0 ±4%	3.9 ±9%	13 ±3%	10

[a]The experiment was limited to the first 10000 solutions out of 4 million

Table 8.6: Timing results of the rule-fitting solver converting from the 5-table ruleset to the 2-table pipeline. Build SAT reports the one-off cost to build the SAT expression and initialise the SAT solver. Solve SAT measures the time spent in the SAT solver, and Verify measures the time spent building and checking the equivalence of the candidate solution. Both Solve SAT and Verify times are cumulative across all of SAT solver iterations. The one-off time to build the SAT problem is insignificant compared to the Solve and Verify times. The total time includes everything from loading the ruleset to outputting a solution. All cases see a better or equal runtime after adding more constraints.

Forwarding Conflicts (§7.6.2): In addition, each iteration computes constraints based on the specific conflicts between the expected forwarding and incorrect forwarding in the candidate solution. This is in addition to picking a different combination of hit placements each iteration.

We present an evaluation of the usefulness of these SAT constraints when converting between the 2-table pipeline and 5-table pipeline (§8.1.1). For ruleset preprocessing, we enabled both single-table conversion and ruleset compression. The 5-table ruleset once compressed and converted to a single-table contained ten rules, while the 2-table ruleset contained seven rules. Although both of these rulesets are small, the size of the problem can still grow large, so we place an upper limit on the number of SAT solver iterations at 10,000.

Table 8.6 shows the timing results of converting the 5-table ruleset to the 2-table pipeline and Table 8.7 shows the corresponding metrics from the rule-fitting solver. Both tables are ordered from least constrained (i.e. largest search space) at the top to all constraints at the bottom. With only the one transformation constraint there were 4 million combinations of transformations, however, we stopped the solver after considering 10 thousand. Because many transformations share placements, adding placements variables and de-

Constraints	Solutions		SAT Metrics			
(Cumulative)	Valid	Uniq.	Iterations	Var.	Sln Var.	Clauses
One Transformation	8229	7	10000[a]	52	46	100
Placements	155	10	240	66	14	184
Placement Conflicts	49	10	64	66	14	190
Table-Miss	49	10	49	68	14	197
Hit Placements	10	10	10	82	14	237
Forwarding Conflicts	10	10	10	82	14	237

[a]The experiment was limited to the first 10000 solutions out of 4 million

Table 8.7: Solver metrics when converting from the 5-table ruleset to the 2-table pipeline. Iterations reports the number of unique solutions to the SAT expression. We want to lower iterations as much as possible without decreasing the number of unique solutions. An increase in the number of clauses indicates the solver has added more constraints, but not if they are effective. Var. reports the number of boolean variables in the SAT expression, and Sln Var. the number of variables which define a unique solution. Using placement variables (instead of transformations) to define a unique solution dropped the combinations returned by the SAT solver from 4 million down to 240. Without hit placements and table-miss constraints, a unique combination of solution variables does not always map to a unique candidate ruleset. The difference between valid and unique solutions shows how many times the SAT solver returned a valid solution that was the same effective ruleset as another already seen. A difference other than zero indicates duplicate processing of solutions. The addition of hit placements reduces this difference to zero as expected. Once the rule-fitting solver adds hit placements, all candidates returned from the SAT solver are valid solutions, so there are no forwarding conflicts to add.

fining a unique SAT solution by these placements reduces the search space significantly to 64. Table 8.6 shows that additional constraints added minimal extra compute time to the one-off cost of building the SAT problem, yet significantly reduced the time spent verifying solutions. With additional constraints, the number of iterations of the SAT solver decreased significantly; thus, the rule-fitting solver had fewer candidate solutions to verify and was faster.

Table 8.7 shows the number and types of solutions each constraint removes. Adding placement variables and using them to define a unique SAT solution significantly reduced the number of solutions returned by the SAT solver from 4 million to 240, as iterations shows. Adding placement conflict constraint reduced iterations further to 64. Adding the table-miss constraint removed 15

Constraints	Timing (ms)				
(Cumulative)	Total	Build SAT	Solve SAT	Verify	Iterations
One Transformation	262 ±2%	5.6 ±5%	4.9 ±14%	149 ±2%	16
Placements	248 ±1%	6.5 ±3%	6.2 ±6%	140 ±1%	16
Placement Conflicts	130 ±1%	6.7 ±4%	2.1 ±6%	26 ±1%	4
Table-Miss	131 ±1%	7.2 ±4%	2.1 ±7%	26 ±1%	4
Hit Placements	132 ±1%	7.8 ±3%	2.1 ±7%	26 ±6%	4
Forwarding Conflicts	135 ±1%	7.8 ±2%	3.1 ±7%	27 ±2%	4

Table 8.8: Timing results converting from the 2-table ruleset to the 5-table pipeline. After adding placement variables, all additional constraints failed to reduce the number SAT solver iterations. After which, there is a slight increase in run-time as we add more constraints due to the one-off cost of adding additional constraints.

Constraints	Solutions		SAT Metrics			
(Cumulative)	Valid	Uniq.	Iterations	Var.	Sln Var.	Clauses
One Transformation	1	1	16	17	12	23
Placements	1	1	16	38	21	82
Placement Conflicts	1	1	4	38	21	89
Table-Miss	1	1	4	42	21	106
Hit Placements	1	1	4	63	21	157
Forwarding Conflicts	1	1	4	63	21	157

Table 8.9: Solver metrics when converting from the 2-table ruleset to the 5-table pipeline. After adding placement conflicts, iterations remained at 4; thus, all additional constraints, despite adding clauses, did not reduce the size of the problem.

invalid candidate solutions from consideration, as evidenced by the decrease in SAT solver iterations with the number of valid solutions remaining unchanged. Adding hit placements and using them to define a unique SAT solution entirely eliminated rechecking the same candidate ruleset. Hit placements reduced the SAT solutions by 39 to 10, all of which are valid solutions and resulted in unique rulesets. Forwarding conflicts had no effect, as the rule-fitting solver only generates them from invalid solutions.

With only the one transformation constraint, only seven unique solutions were found out of a possible ten because we limited the rule-fitting solver to check only the first 10,000 candidate solutions. After that, the rule-fitting solver always finds ten unique solutions, which means that no SAT constraints removed search space that contained valid solutions.

Table 8.8 shows the timing results of converting the 2-table ruleset to the 5-table pipeline and Table 8.9 shows the corresponding metrics from the solver. For this conversion, the rule-fitting solver had very few options for each input rule's placement. With only the one transformation constraint, the SAT solver returned only 16 candidate solutions. Adding the placement variable constraints reduced this to 4, no other constraints reduced the number of candidate solutions the SAT solver returned. As a result, Table 8.8 shows all additional constraints increased the solve time by a minimal amount, due to the extra work in computing these constraints. Table 8.9 shows additional constraints increased the number of clauses, so the rule fitting-solver added extra SAT constraints. But as the number of iterations remained unchanged, these new clauses were already encompassed by the existing constraints.

Overall we have found the SAT constraints are beneficial to performance and do not exclude search space that contains valid solutions. Often constraints significantly improve performance by orders of magnitude, by reducing the number of solutions to the SAT expression. Such as taking a search space of 4 million down to 10. We have not found any cases where these constraints incorrectly exclude valid solutions.

8.5 Discussion

8.5.1 Real-World Considerations

Thus far, we have evaluated our rule-fitting solver against our handcrafted rulesets and pipelines. One of which, the 5-table pipeline, was based on the real-world OF-DPA pipeline, and included many of its complexities. The rule-fitting solver was able to fit a different ruleset to this pipeline. However, we have been unable to get the rule-fitting solver to fit real-world rulesets to real-world pipelines. Here we discuss some reasons for this.

For a real-world ruleset and a real-world pipeline, it may simply not be possible to fit the ruleset. However, due to the complexity of the problem, there is also no way to know if it is possible. We have found example cases where, given the transformations our rule-fitting solver generated, it was not possible to fit a ruleset. One example is from trying to fit the ruleset collected from a Faucet controller into a modified version of the OF-DPA pipeline. The solver typically completes after 5-10 minutes without finding a solution; the time varies depending on the order the rule-fitting solver checks candidate solutions. We modified the OF-DPA pipeline to remove some non-standard tables, which used vendor extensions, that our rule-fitting solver would not interpret correctly. The Faucet ruleset dropped packets destined to particular Multicast Ethernet addresses in its first table. In the OF-DPA pipeline, these drop rules can be installed in the ACL table; however, each rule must additionally match the VLAN present bit. Because all packets must be assigned a VLAN when they enter the OF-DPA pipeline, this will match all packets. However, our rule-fitting solver does not consider this placement valid, because it does not correctly match untagged packets like the original rule does.

In this case, this was a limitation of our rule-fitting solver. However, we found modifying the Table Type Pattern, to allow the VLAN match to be omitted, still did not result in the rule-fitting solver finding a valid solution.

The other common failure scenario is running out of memory when fitting to a less constrained pipeline. The solver exhausts memory because it generates all possible transformations in its first stage. For a pipeline with fewer constraints, the rule-fitting solver generates more placements for each rule. A longer pipeline exacerbates this problem as split transformations traverse all possible paths through these tables, each table multiplies the size of the problem by the number of partial placements. We encountered this issue when trying to refit a ruleset collected from a Faucet controller back into its own pipeline, for which we had created the Table Type Pattern. Fitting this ruleset ran out of memory because the pipeline is eight tables long and most tables accepted rules with an output action. For input rules with multiple output actions, this resulted in many unique placement combinations with these actions spread throughout the pipeline.

This failure represents the trade-off between searching for all possible ways to place a ruleset and keeping the problem size tractable.

8.5.2 Assumptions and Limitations of our Approach

In this section, we highlight the significant assumptions made by and limitations of our approach. Many of these we have highlighted or alluded to previously in the relevant section.

Our technique generates all possible transformations of each input rule, which grows exponentially with the number of choices available in the target pipeline. As mentioned in the discussion above, this works for constrained pipelines where the number of choices is small, for example, where only one or two tables can match a header field or apply an action. However, for a flexible pipeline such as a software pipeline where all tables support all types of actions and matches, our algorithm will run out of memory as it tries all possible combinations of actions and matches split across all tables. A complementary technique is needed for this flexible target pipeline case and remains an area for future research.

Our technique assumes a single path through the target pipeline can fully represent the forwarding of an input rule. Otherwise, rule-fitting fails. A counter-example to this assumption is a target pipeline that splits into two separate paths, for example, one set of tables for handing IPv4 packets and another for IPv6. Rule-fitting to a split pipeline like this could be resolved by designing a new type of transformation for a split path or removing the one transformation per input rule SAT requirement (§7.4.1). Both options will increase the number of candidate solutions the solver needs to explore.

Our approach finds valid solutions; these are not necessarily good solutions. Our research has not considered the optimality of the rule-fitting solution. The optimality of the solution remains a future direction for research.

Our approach finds a valid solution for one ruleset at a point in

time. Our approach does not guarantee the stability of the result generated, that means rerunning the solver with the same input might result in a completely different output ruleset. This also means an incremental update to the input ruleset (either adding or removing a rule) might drastically change the solution ruleset which might not be possible to install to the switch without temporarily interrupting forwarding. We believe the model created by ruleset compression (§6.2.3) could be applied to incremental updates, in many cases, to avoid a complete recalculation. Additionally, we decided to solve the problem of fitting a ruleset into a pipeline, rather than all combinations of rules a controller could generate; this means an incremental change the controller makes to the input ruleset can result in an unsolvable problem. This is not suitable for deployment. Result stability and incremental updates remain a significant area for future research.

We strictly enforce ruleset equivalence when checking if a solution is valid, relaxing this enforcement would find more valid solutions. If a particular type of packet is known not to be present on a network, the forwarding a switch applies to this type of packet is irrelevant. For example, rule-fitting for a switch in the core of an MPLS network could exclude all non-MPLS packets from the equivalence check and thus find more valid solutions. While, from a security perspective, it is preferable to install a rule to drop non-MPLS packets explicitly, the target hardware or existing controller might not support it. One possible way to relax the equivalence check in such a scenario is to add an extra table to the start of both pipelines to drop non-MPLS packets before running the equivalence check.

In a similar vein, we have only considered the rule-fitting problem for a single switch and how to fit a specific forwarding behaviour. By opening this problem up to find a solution for an entire network or fitting high-level language concepts (such as a tunnel between two hosts, rather than a specific VLAN tunnel), it might be possible to find more solutions. However, this comes at the cost of drastically increasing the size of the problem.

8.6 Summary

This chapter evaluated three techniques we developed to improve the performance of the rule-fitting solver: preprocessing the ruleset to a single table, compressing the ruleset during preprocessing, and adding the SAT constraints. We found that the rule-fitting solver was faster with a single-table ruleset input compared to using the original multi-table input. Additionally, the solver failed to find a solution for the original multi-table ruleset, when it did for the single-table input. We found that compressing the ruleset as a preprocessing step always improved the performance of the rule-fitting solver. For a large ruleset, compression reduced the time to find a solution from 12min 38s to 6.1s. Our evaluation of the SAT constraints found that overall adding constraints reduce the solve time, and at worst add negligible overhead. In this evaluation, these constraints only prevented searching invalid and repeated solutions and did not remove any valid solutions.

Finally, this chapter discussed the limitations of our approach and difficulties we encountered when working with real-world rulesets and pipelines.

Chapter 9

Conclusion

9.1 Summary of Thesis

This thesis presented research towards the goal of improving Software-Defined Networking (SDN) device interoperability. The ultimate goal of this work was to create a general algorithmic approach to the rule-fitting problem for constrained fixed-function pipelines. In this research, we developed a rule-fitting solver to convert an existing OpenFlow 1.3 ruleset to a new target pipeline.

Towards the goal of solving the rule-fitting problem, we encountered and found solutions to two significant problems.

The first problem considered was how best to represent the constraints of an OpenFlow hardware pipeline. This thesis compared two existing solutions to this problem: Table Type Patterns [3] and OpenFlow Feature Messages [12]. We found that Table Type Patterns were the best choice for our research, as they could fully describe the constraints of fixed-function pipelines. However, the ecosystem around Table Type Patterns was limited, and there were practically no existing tools to interpret or create them.

This thesis introduced a library and a set of tools for working with Table Type Patterns. The tools presented assist developers with reading and verifying Table Type Patterns, and can produce a helpful recommendation as to where mistakes lie and possible remediation. Additionally, this library was designed to find valid placements for OpenFlow rules in the target pipeline. These Table Type Pattern tools are valuable to other SDN researchers and developers who wish to use Table Type Patterns. Towards the goal of device interoperability, this Table Type Pattern library can verify if a ruleset is compatible with a network device's pipeline.

The second problem considered was how best to check if two rulesets were equivalent; required to check if the output of the rule-fitting solver was correct. One difficult part of checking ruleset equivalence was representing the set of packets which observe the same forwarding.

This thesis compared representing sets of packets as TCAM-style matches and as a Binary Decision Diagram (BDD) [22]. The TCAM-style matches considered included OpenFlow [12] matches and Header Space [39] wildcards. We found that these TCAM-style representations could not efficiently represent the difference between two sets of packets, but could efficiently represent the intersection of two sets of packets. We found BDDs could efficiently represent the difference between two sets of packets, along with all other set operations, and were a canonical representation.

The thesis developed a comprehensive method of checking the equivalence between two OpenFlow rulesets. This method builds an Multi-Terminal Binary Decision Diagram (MTBDD) representation of the ruleset, which maps sets of packets to their corresponding forwarding behaviour. This MTBDD is a canonical representation and is trivial to compare to another. We provided a comprehensive way to identify equivalent actions by combining OpenFlow write-actions, apply-actions, and groups into a canonical representation. We found our technique took in the order 10's of seconds to build this MTBDD representation for real-world rulesets. This method is a useful tool for the SDN community, including developers to check for regressions in their application code, and researchers who are rewriting rulesets to verify their modifications. Towards the goal of OpenFlow interoperability, this equivalence checking can be used to verify two OpenFlow applications are generating equivalent rulesets.

Finally, to directly address the goal of improving OpenFlow interoperability, this thesis presented a general algorithmic approach to the rule-fitting problem. We targeted our work to fitting a ruleset into a constrained fixed-function pipeline. We studied the requirements of fitting a ruleset into a real-world fixed-function pipeline, Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) pipeline. We discovered that our solver would need to deal with complexities, including fitting to a pipeline that required the use of write-actions and clear-actions, and fitting to a pipeline without support for arbitrary metadata.

The solver we designed has two stages. The first stage preprocesses the ruleset to simplify it and then generates transformations for rules and paths which fit the target pipeline. The second stage tries to find a combination of these transformations that result in an equivalent ruleset. This thesis described both stages in detail.

Within these stages, a key problem we needed to solve was reducing the problem to a tractable size. This thesis presented a method of compressing a ruleset without losing information that the rule-fitting problem required. The evaluation of this technique showed sizeable gains in rule-fitting performance compared to an uncompressed ruleset. In one example, reducing the time to find a solution from 12min 38s to 6.1s. To improve performance, developers could apply this technique to other rule-fitting solvers and it could even be applicable to other network analysis tools.

This thesis presented the types of rule and path transformations our rule-fitting generates. We designed these transformations to find solutions in complex circumstances, such as through paths that use write and clear-actions or require the 'wrong' action.

This thesis presented a method to express picking combinations of these transformations as a boolean satisfiability problem. We presented the constraints that we used to filter out combinations of transformations which were unlikely to contain valid solutions. Our evaluation found that these constraints were successful in reducing the problem size without excluding valid solutions.

We have demonstrated that the rule-fitting solver can solve problems with complexity in a handcrafted scenario based on the OF-DPA pipeline. Our technique does not rely on OpenFlow metadata to link paths through the ruleset it outputs, which allows it to fit to pipelines without metadata support. While our implementation targets OpenFlow, the key principles behind our approach apply to other match-action style pipelines. This thesis discussed the difficulties of fitting a real-world ruleset to a real-world pipeline and the major assumptions and limitations of our approach.

9.2 Future Work

We have demonstrated, through implementation, that our equivalence checking method works with OpenFlow rulesets. A future direction for this research is to investigate if the same equivalence checking can be applied to rulesets from different standards, such as P4 [10]. Another direction is to investigate whether it is feasible to represent the forwarding behaviour of multiple connected network devices using this technique, for use in network analysis tools. A packet sent to another networking device, starts its processing in the first table of that device, which can be viewed as an extension of the current pipeline. However, this introduces complexities to deal with, such as loops.

The ruleset compression presented in Section 6.2.3 significantly speed up our rule-fitting solver and could have a broader application more generally in network analysis tools. Network analysis tools make calculations based on the forwarding information of network devices within a network. Further work is needed to formalise the information lost by compression and investigate if compression is suitable for tools which analyse a entire network. Additionally, we previously identified that the problem of how to best compress routing tables remains unanswered (§6.2.3.5).

In Section 6.3.8, we provided a list of additional transformations which require further research into how to generate them and their usefulness. In addition to generating the additional transformations, future research would need to consider how these transformations interact with other transformations and if new SAT constraints are necessary.

Our rule-fitting solver searches for the first valid solution it can find; this might not be a good solution. For example, the ruleset output might contain too many rules to install in hardware, despite a better solution existing. More research is required into finding if the SAT constraints can be modified to guide the solver towards more optimal solutions. In addition, future work could consider if a different approach to picking combinations of transformations is more suited to finding optimal solutions. Such research would also be useful to optimise for other metrics such as power efficiency.

FlowAdapter and FlowConvertor have demonstrated that algorithmic rule-fitting techniques can be run in real-time as a middle layer between controller and switch [38, 37]. Further investigation is needed to explore the possibility of running our rule-fitting technique in real-time. One insight is that by compressing the ruleset we created a more general solution which we mapped back to the original ruleset. Therefore, it might be possible to apply this mapping to rules incrementally as a controller adds and removes rules without having to rerun the full solver each time.

One problem we faced during our the research was obtaining real-world data-sets for our testing. It was difficult to source rulesets from SDN applications and pipeline descriptions. A corpus of SDN rulesets and more generally network data-sets would be a great asset to the research community.

We have presented one possible approach to the rule-fitting problem, and we hope that our work provides insights for future researchers to develop new approaches.

Bibliography

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] Rule-fitting overview - a collection of the libraries and tools creating during this thesis [source code]. [Online]. Available: https://github.com/wandsdn/rule-fitting-overview

[3] "Openflow table type patterns," Tech. Rep., 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlowTableTypePatternsv1.0.pdf

[4] E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J.-Y. Yang, and X. Zhao, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *International Workshop on Logic Synthesis*, May 1993.

[5] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[6] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[7] J. Bailey and S. Stuart, “Faucet: Deploying SDN in the enterprise,” *Queue*, vol. 14, no. 5, p. 30, 2016.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, “B4: Experience with a globally-deployed software defined WAN,” in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1145/2486001.2486019

[9] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern, “Forwarding and control element separation (ForCES) protocol specification,” Internet Requests for Comments, RFC Editor, RFC 5810, March 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5810.txt

[10] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, “P4: Programming protocol-independent packet processors,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[11] H. Song, “Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 127–132.

[12] Open Networking Foundation. (2015, Mar.) OpenFlow Switch Specification - Version 1.3.5. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[13] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, “The design and implementation of Open vSwitch.” in *NSDI*, 2015, pp. 117–130.

[14] Broadcom Corporation, "Broadcom-switch/of-dpa: Openflow data plane abstraction," 2016. [Online]. Available: https://github.com/Broadcom-Switch/of-dpa

[15] Big Switch Networks, "Hardware support and certification," Jun. 2018. [Online]. Available: http://opennetlinux.org/hcl

[16] Edge-Core, "AS5710-54X-EC 10GbE L3 switch with SDN capability, supporting OpenFlow 1.3 and OF-DPA," Oct. 2017. [Online]. Available: https://www.edge-core.com/_upload/images/AS5710-54X_EdgeCOS_DS_R02_20171005.pdf

[17] R. Sanger, M. Luckie, and R. Nelson, "Identifying equivalent SDN forwarding behaviour," in *Proceedings of the 2019 ACM Symposium on SDN Research*. ACM, 2019, pp. 127–139. [Online]. Available: https://doi.org/10.1145/3314148.3314347

[18] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1066–1075.

[19] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013, pp. 99–111.

[20] C.-Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell Labs Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[21] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on computers*, no. 6, pp. 509–516, 1978.

[22] R. E. Bryant, "Graph-based algorithms for boolean function manipula-

tion," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[23] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE.* IEEE, 1990, pp. 40–45.

[24] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 2009.

[25] M. G. Gouda and X.-Y. Liu, "Firewall design: Consistency, completeness, and compactness," in *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on.* IEEE, 2004, pp. 320–327.

[26] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," in *Proceedings of the 30th international Design Automation Conference.* ACM, 1993, pp. 54–60.

[27] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for netkat," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 328–341, 2015.

[28] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proc. 2016 ACM SIGCOMM Conf.*, 2016, pp. 29–43.

[29] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, pp. 887–900, 2013.

[30] S. Hazelhurst, A. Fatti, and A. Henwood, "Binary decision diagram representations of firewall and router access lists," *Department of Computer Science, University of the Witwatersrand, Tech. Rep*, 1998.

[31] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," in *Security and Privacy, 2006 IEEE Symposium on.* IEEE, 2006, pp. 15–pp.

[32] A. Prakash and A. Aziz, "Oc-3072 packet classification using bdds and pipelined srams," in *Hot Interconnects 9, 2001.* IEEE, 2001, pp. 15–20.

[33] T. Inoue, T. Mano, K. Mizutani, S.-i. Minato, and O. Akashi, "Fast packet classification algorithm for network-wide forwarding behaviors," *Computer Communications*, vol. 116, pp. 101–117, 2018.

[34] A. Ronacher, "Flask (a python microframework)." [Online]. Available: http://flask.pocoo.org/

[35] Microsoft, "Extension methods (c# programming guide)," Jul. 2015. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

[36] Equivalence checking implementation [source code]. [Online]. Available: https://github.com/wandsdn/ofequivalence

[37] H. Pan, G. Xie, Z. Li, P. He, and L. Mathy, "Flowconvertor: Enabling portability of SDN applications," in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE.* IEEE, 2017, pp. 1–9.

[38] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The flowadapter: Enable flexible multi-table processing on legacy hardware," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking.* ACM, 2013, pp. 85–90.

[39] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 113–126.

[40] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Transactions on computers*, vol. 45, no. 9, pp. 993–1002, 1996.

[41] F. Somenzi, "CUDD: CU decision diagram package release 3.0. 0," 2015.

[42] WAND. Redcables SDN Network @ WAND, Waikato University. [Online]. Available: https://redcables.wand.nz/

[43] D. Meyer. (2001) University of oregon route views archive project. [Online]. Available: https://routeviews.org/

[44] L. Yang, B. Ng, W. K. Seah, and L. Groves, "Equivalent forwarding set evaluation in software defined networking," in *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on.* IEEE, 2017, pp. 576–579.

[45] Broadcom Corporation, "Openflow data plane abstraction (OF-DPA$^{\text{TM}}$): Abstract switch specification version 2.01," Tech. Rep., Jan. 2016. [Online]. Available: https://github.com/Broadcom-Switch/of-dpa/blob/v2.0.4.1/OFDPAS-ETP100-R.pdf

[46] M. Yu, A. Wundsam, and M. Raju, "NOSIX: A lightweight portability layer for the SDN OS," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 28–35, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602209

[47] Y. Tseng, "ONOS intents and northbound," ONOS Build 2017 [Slides]. [Online]. Available: https://onosproject.org/wp-content/uploads/2018/01/4-ONOS-Build-2017-Northbound.pdf

[48] M. Bruyère, E. Fernandes, I. Castro, S. Uhlig, R. Lapeyrade, P. Owezarski, A. Moore, and G. Antichi, "Umbrella: a deployable SDN-enabled IXP switching fabric," in *ACM symposium on SDN Research*, 2018, p. 2p.

[49] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 39–50.

[50] N. Een, A. Mishchenko, and N. Sorensson, "Applying logic synthesis for speeding up SAT," *Sat*, vol. 7, pp. 272–286, 2007.

[51] ONF, "Open networking foundation releases Atrium open SDN software distribution," [Online]. Available: https://www.opennetworking.org/news-and-events/press-releases/2327-open-networking-foundation-releases-atrium-open-sdn-software-distribution, [Accessed 15 September 2015].

[52] ——, "Atrium docs - home," [Online]. Available: https://github.com/onfsdn/atrium-docs/wiki, [Accessed 25 September 2019].

[53] ——, "CORD platform | central office rearchitected as a datacenter | ONF," [Online]. Available: https://www.opennetworking.org/cord/, [Accessed 25 September 2019].

[54] D. Parniewicz, R. Doriguzzi Corin, L. Ogrodowczyk, M. Rashidi Fard, J. Matias, M. Gerola, V. Fuentes, U. Toseef, A. Zaalouk, B. Belter *et al.*, "Design and implementation of an openflow hardware abstraction layer," in *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing.* ACM, 2014, pp. 71–76.

[55] X. Sun, T. Ng, and G. Wang, "Software-defined flow table pipeline," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, March 2015, pp. 335–340.

[56] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX NSDI*, 2015.

[57] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. ACM Symposium on SDN Research (SOSR)*, 2016.

[58] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* ACM, 2018, pp. 476–489.

[59] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing.* ACM, 1971, pp. 151–158.

[60] L. A. Levin, "Universal sequential search problems," *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 115–116, 1973.

[61] "The international SAT competitions web page," [Online]. Available: http://www.satcompetition.org.

[62] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon, "The international SAT solver competitions," *AI Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[63] DIMACS Challenge, "Satisfiability: Suggested format," *DIMACS Challenge. DIMACS*, 1993.