Simplifying Development and Management of Software-Defined Networks

THÈSE N^{O} 7075 (2016)

PRÉSENTÉE LE 8 JUILLET 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'EXPLOITATION
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Peter PEREŠÍNI

acceptée sur proposition du jury:

Prof. E. Bugnion, président du jury
Prof. W. Zwaenepoel, Prof. D. Kostic, directeurs de thèse
Prof. M. Canini, rapporteur
Prof. L. Vanbever, rapporteuse
Prof. K. Argyraki, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Abstract (German)

Computernetzwerke sind ein wichtiger Teil unseres Lebens. Doch es ist schwierig, sie zu verwalten und zu betreiben. Der Paradigmenwechsel zu Software-Defined Networking (SDN) verspricht Netzwerkadministratoren einen neuen Weg dazu. SDN ist jedoch noch in der Anfangsphase. Im Vergleich mit traditionellen Netzwerken mangelt es an Korrektheit und Zuverlässigkeit. Diese Eigenschaften sind für Netzwerkbetreiber unabdingbar.

SDN entstehen zu einer Zeit, in der Rechenkapazität günstig ist wie nie zuvor. Wir profitieren von Moores Gesetz und von verbesserten Algorithmen, und verfügen heute über mehr Rechenkapazität denn je.

Mit dieser Dissertation will ich die Lücke verkleinern zwischen der Korrektheit und Zuverlässigkeit von traditionellen Netzwerken und SDN. Diese Arbeit soll zur Annahme von SDN beitragen und Werkzeuge bieten, mit denen Programmierer und Operatoren Vertrauen in ihre Netzwerke erlangen können. Diese Werkzeuge nutzen die Tatsache, dass grosse Mengen Rechenkapazität zunehmend allgemein verfügbar sind. Sie verbinden diesen Trend mit Forschungsergebnissen in systematischer Validierung, in Optimierung, und im Lösen von logischen Erfüllbarkeitsproblemen.

Das erste Werkzeug in dieser Dissertation, NICE, dient zur Fehlersuche in einem SDN-Kontroller. Obwohl SDN konzeptuell ein zentralisiertes System ist und von einem zentralisierten Kontroller gesteuert wird, ist es dennoch ein verteiltes System. Dadurch müssen Programmierer alle möglichen Ereignisreihenfolgen und Race Conditions bedenken, die zu Fehlern führen könnten. NICE findet solche Fehler, indem es die möglichen Netzwerkzustände systematisch erforscht. NICE kombiniert dazu auf neuartige Weise zwei Techniken: symbolische Programmausführung und Model Checking. Mit dieser Kombination und SDN-spezifischen Heuristiken kann NICE die möglichen Szenarien tiefer erforschen als bestehende Techniken.

Als Zweites präsentiert diese Arbeit Monocle, ein Werkzeug, das die Zuverlässigkeit von SDN-Switches garantiert. Monocle stellt laufend sicher, dass die Switches in der Data-Plane gemäss den Bestimmungen des Kontrollers arbeiten. Solch eine Überwachung ist wichtig, denn Fehler in der Firmware, Speicherzugriffsfehler oder vorübergehende Probleme können sonst unentdeckt bleiben. Monocle überprüft das Verhalten der Data-Plane-Switches, indem es Datenpakete konstruiert und diese ins Netzwerk einspeist. Dies ist kein einfaches Problem, denn ein Switch folgt einem komplizierten Mechanismus zum Verarbeiten eines Pakets, insbesondere wenn mehrere Regeln auf das selbe Paket zutreffen. Diese Arbeit beschreibt formelle Bedingungen, welche die generierten Pakete erfüllen müssen. Sie zeigt, wie ein handelsüblicher Satisfiability Solver diese Bedingungen lösen kann.

Als Drittes stellt diese Arbeit ESPRES vor, ein Werkzeug zum Dirigieren von Netzwerk-Updates. Ich stelle fest, dass grosse Updates wie z.B. Traffic Engineering oder störungsbedingte Umleitungen aus vielen Teil-Updates bestehen. Diese sind häufig unabhängig (z.B. wenn sie unterschiedliche Datenströme betreffen). Die Dauer des gesamten Updates wird vom langsamsten Switch bestimmt und kann nicht verbessert werden. Ich behaupte jedoch, dass die Mehrheit der Teil-Updates wesentlich schneller fertiggestellt werden kann. ESPRES erreicht dieses Zeil ohne vorherige Planung. Es passt die Reihenfolge der Update-Befehle mit einem Greedy-Algorithmus laufend an, unter Berücksichtigung des aktuellen Zustandes der Switches.

Keywords: *Software-Defined Networking, OpenFlow, Zuverlässigkeit, Netzwerkmanagement, Monitoring, Effiziente Updates*

Abstract (English)

Computer networks are an important part of our life. Yet, they are traditionally hard to manage and operate. The recent shift to Software-Defined Networking (SDN) is promised to change the way in which the networks are run by their operators. However, SDN is still in its initial stage and as such it lags behind traditional networks in terms of correctness and reliability; both the properties being vitally important for the network operators.

Meanwhile, general purpose computers were following Moore's law for years and as a result, together with algorithmic improvements, the costs of computing have been declining — we have more processing power available to us than anytime before.

In this dissertation I strive to reduce the correctness and reliability gap of SDN and help speed up the adoption of SDN by providing tools that help programmers and operators gain confidence in their networks. These tools leverage the today's trend in the increasing availability of vast amounts of computing power and combine it with the past research on systematic validation, optimization and satisfiability solving.

The first tool this thesis introduces, NICE, focuses on debugging of an SDN controller. In particular, while SDN is conceptually a centralized system with the controller in the command, SDN is still a distributed system. As such, programmers might miss various event interleavings and race conditions that lead to a buggy behavior. NICE uncovers such insidious bugs by systematically exploring possible network states by a novel combination of two techniques – symbolic execution and model checking. The combination of the two techniques and SDN-related heuristics let NICE explore possible scenarios deeper than any existing technique alone.

This thesis also introduces Monocle, a tool aimed at ensuring reliability of SDN switches. Specifically, Monocle continuously verifies that a switch data plane processes packets as configured by the SDN controller. Such monitoring is important in the presence of silent errors ranging from switch firmware bugs, through memory corruption, to transient inconsistencies. To verify that the switch data plane works as intended, Monocle constructs data plane packets and injects them into the network. Such packet construction is, however, a rather intricate problem when considering a complicated nature of the switch packet matching mechanism in the presence of multiple overlapping rules. In the thesis I formally describe the constraints that the generated packets need to satisfy, and I leverage an off-the-shelf satisfiability solver to solve them.

Finally, this thesis introduces ESPRES, a tool focused on scheduling network updates. I first observe that big network updates such as traffic engineering or failure re-routing usually contain many subupdates that are independent (*e.g.*, updates of different flows). Then I conjecture that while the length of the total update is bottlenecked by the slowest switch, the time when an individual subupdate finishes can be greatly improved for a majority of them. To achieve this goal in an online fashion, ESPRES uses a greedy mechanism to reorder individual switch commands according to the current situation on different switches.

Keywords: *Software-Defined Networking; OpenFlow; Reliability; Monitoring; Update performance*

Acknowledgements

Writing a dissertation and defending it is not a small feat. Here I would like to thank all the people who, during the last 5 years, helped me to successfully finish this chapter of my life.

First and foremost, I would like to thank my advisors Prof. Dejan Kostić and Prof. Willy Zwaenepoel. Dejan was an exceptional advisor who gave me the liberty to work on whatever I liked. This proved to be a successful strategy for keeping me interested in my studies till the end. Moreover, as an academic father, Dejan was patient and understanding when the research did not go as planned. Finally, I am grateful to Willy for adopting me, after Dejan moved to Spain, into his lab while letting me continue working with Dejan.

I would really like to thank my colleague Maciej without whom I cannot imagine publishing so many papers. Our collaboration went beyond what is usual for PhD students and we submitted almost all papers together. After Dejan moved, it helped me a lot to have a buddy with whom I could discuss the intricacies of the projects I was working on, as well as somebody who would push for the end result when my enthusiasm was dwindling.

I feel privileged to be able to work with my collaborators Prof. Marco Canini, Dr. Daniele Venzano, and Prof. Jennifer Rexford. Marco's constant questions, Daniele's great coding skills and Jennifer's attention to small details were invaluable for the NICE bug-finding tool as well as subsequent projects.

I would also like to thank all my colleagues from both NSL and LABOS: Deki, Nedeljko, Stanko, Dimitri, Calin, Jasmina, Laurent, Pamela, Florin, Kristina, Diego, Baptiste and Mihai. Without you, my life at EPFL would be really dull.

My thanks also go to Prof. Katerina Argyraki and Prof. Laurent Vanbever for being part of my committee as well as Prof. Edouard Bugnion for taking the post of committee president.

Living away from home sometimes feels isolating. Fortunately, I have many Slovak friends in Lausanne who made it clear that we are one big family here. Thanks to Ondrej, Matej, Andrej, Katka, Rudo, Janka, Kubo, Terka, Rado, Pavol, Miro, Fero and Janka for this. Thanks also to Jonas who is the best Swiss friend I have made. Special thanks to my family for supporting me all these years and bearing with me being always busy and not really responsive.

Finally, I would like to thank Swiss people for being Swiss and providing the perfect infrastructure to access the Alps. Hiking thousands of meters up during sunny weekends helped me to regain some sanity after sitting at a computer all day long.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Computer Networks of Today

Computer networks are becoming ubiquitous and an indispensable part of life. Even the most rudimentary actions such as looking up a forecast on a mobile phone require communication over multiple networks: First, your mobile uses a 4G/LTE network of your service provider to deliver the webpage. On the other end of the connection, load balancers, web servers and application backends use a high-speed network inside a datacenter to produce the web page delivered. Finally, the forecast would not be produced without a highly customized network of some supercomputer. If this is not enough, add to the list networks of big companies, each spanning multiple branch offices, networks of healthcare providers such as hospitals, networks between financial institutions, *etc.*, and it becomes clear that we require networks to be constantly up and functional. Moreover, the importance of computer networks is poised to increase over time – the global Internet traffic is forecast to exceed 2 zettabytes per year in 2019 [26], and it is expected that by the end of 2020 there will be several times more Internet-of-Things (IoT) devices connected to the Internet than people on this planet [33].

1.1.1.1 Traditional Network Management

The rapid expansion of the network needs puts their operators into a precarious position. In particular, as the operating margins are slowly dwindling, operators are hard-pressed with introducing new services on an increasingly faster schedule, while guaranteeing very high availability. However, today it can take months to introduce a new service in the current networks. [73] This is because traditionally, network management is a very tedious process. While the details change from network to network, in general, the networks are hard to manage because of the three main reasons: (*i*) lack of centralized control, (*ii*) lack of lingua franca and (*iii*) lack of operational best practices.

Lack of centralized control

To manage the ever increasing complexity of networks it is advantageous to configure the network in a central location. Yet, today we configure networks on a "per-box" basis. This introduces great risks for the operator. First, the networks are less flexible than they should be – for example, large-scale data center networks run Border Gateway Protocol (BGP, [69]) or Interior Gateway Protocol (IGP) to configure the routes between machines [56]. While such solutions address scalability problems of layer 2 (ethernet) broadcast domains, as the size increases, BGP and similar protocols hit a limit on their scalability and manageability; there is a growing list of companies working around the protocols to improve convergence times or customized traffic engineering setups [42, 77]. The situation is even worse for public data centers where operators need to separate tenants by hand-configuring VLANs across multiple routers, and enterprise networks that require certain traffic classes to be routed through appliances such as firewalls, intrusion detection systems, load-balancers, etc.

Lack of lingua franca

The "per-box" configuration of networks has another consequence. Often, these "boxes" are from different vendors. As such, pieces of network equipment from different vendors might use different, proprietary, approaches to configuration – one vendor might use a particular version of command line interface (CLI) while other vendors are likely using different CLI commands. This disconnect means that network administrators need to learn multiple configuration languages, sometimes with fine nuances between them.

The fear of change

Every successful software company knows that big software projects cannot happen without good design, source code version control, code reviews, automated testing and frequent release feedback. However, this is opposite of how networks are configured today — the configuration is usually only on the devices, and it cannot be easily rolled back or tested in isolation. This means that the network operators are rightfully wary of any change. They build full

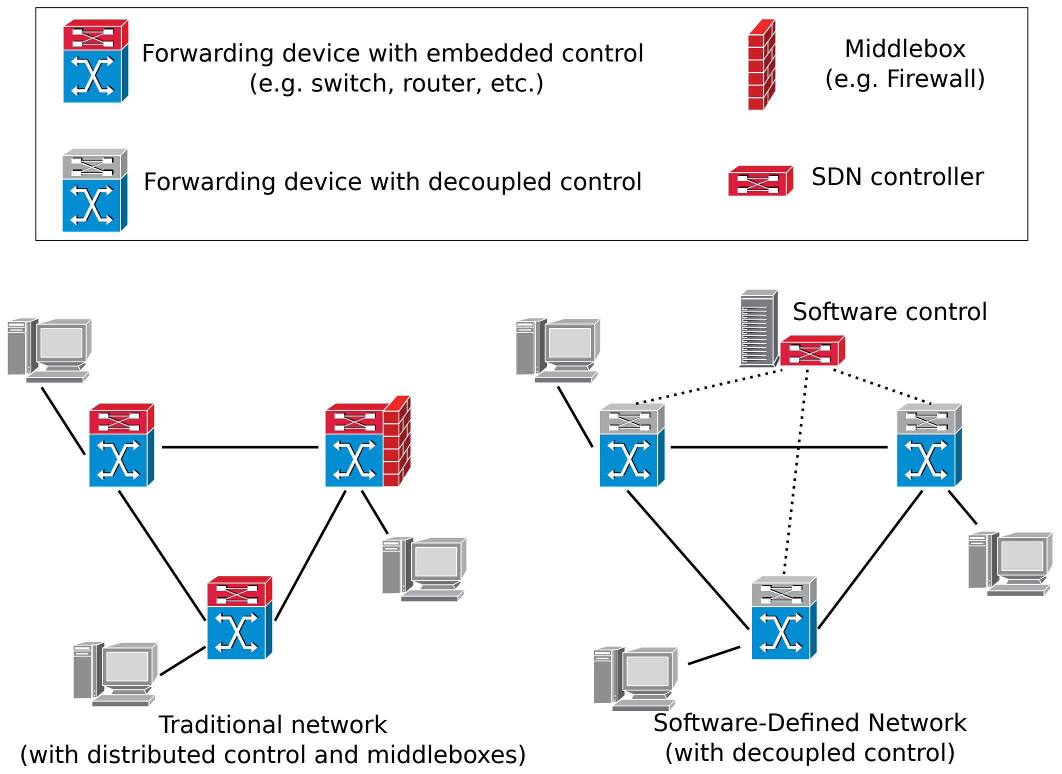

Forwarding device with embedded control
(e.g. switch, router, etc.)
Middlebox
(e.g. Firewall)
Forwarding device with decoupled control
SDN controller
Software control
Traditional network
(with distributed control and middleboxes)
Software-Defined Network
(with decoupled control)

Figure 1.1: Software-Defined Networking decouples data plane (*i.e.*, forwarding) from control plane (*i.e.*, configuration & management)

copies of their networks in labs just to test changes. Moreover, the changes themselves require hours of planning as the reconfiguration must be carried out in the correct order. And even after all of this the network configuration change might go wrong and network administrators have to relentlessly work to restore the network functionality.

1.1.2 Software Defined Networking

Software Defined Networking is a huge shift in a way we think about, design and manage computer networks. In its essence, SDN is defined by two ideas captured in Figure 1.1: (*i*) separate control plane (*i.e.*, decisions on how to handle traffic) from data plane (*i.e.*, forwarding the traffic according to the decisions made by control plane), and (*ii*) centralize control plane so that a single control program oversees multiple data plane elements. By doing this, SDN tries to solve all aforementioned problems of traditional network management.

First, SDN promotes a central configuration by design; in SDN a single (logically) centralized controller is in charge of the network, while the rest of network elements play a passive role. This greatly simplifies the management of the network as the network operators need to configure policies only in one place. A central configuration also helps with the planning of potential changes and network policy troubleshooting.

Second, SDN promotes interoperability. In fact, SDN works best when the network controller can talk to all network elements uniformly, no matter what their vendor is. As such, SDN favors a single open "language" over which the network elements talk (*e.g.*, OpenFlow [6]) instead of a multitude of command-line interfaces.

Finally, because of the separation of concerns, SDN facilitates better operational practices by helping operators overcome the fear of change: First, centralized network configuration makes it easier to inspect, track and even revert policy changes. Second, it is much easier to test software than a combination of hardware appliances. Moreover, it is easy to test the correctness of a switch separately from the correctness of a network controller. Finally, interoperability helps in avoiding vendor lock-in, which is of a big concern for some operators [72]. All in all, SDN is a huge improvement for the networking community.

Unfortunately, despite all its benefits, SDN is facing challenges in its adoption. This is no surprise as SDN, being fairly young and unproven, is far from being ready to use in the production environment where high availability is needed. Indeed, reports claim that the year 2014 was just a year of proof-of-concept demos, and the year 2016 is predicted to be the first year with more companies running SDN in a production [74]. To further speed up the adoption, it is therefore crucial to address the question of reliability and to do so on multiple levels: First, SDN programmers need a way to test whether controllers install correct forwarding rules under all conditions rather than being a victim of concurrency issues that might arise from configuring multiple switches (SDN network is inherently a distributed system; this brings the potential of race conditions). Second, to react quickly, SDN operators need a way to monitor their networks for any anomalies that might arise. Finally, as SDN becomes widely adopted and organizations start using it for new exciting use-cases such as frequent traffic engineering [17], SDN performance will become more and more important.

1.1.3 Software Industry Trends

1.1.3.1 Affordable Computing

Networking is not the only field experiencing massive changes. For the past four decades, the advances in CPU design and manufacturing followed Moore's law, and as a result, the number of transistors doubled around every two years. In the meantime, CPUs manufactured with smaller fabrication processes reduce energy consumption per computation. On the other hand, the sheer size of the market enables mass-production of chips, which keeps their price relatively low. Moreover, while the speed improvements of a single CPU core did not significantly improve from 2004, chip makers are packing an increasing number of cores on a single system [87].

Similarly, software performance benefits from years of active research. Today's programs run faster because of hard work done in compiler optimizations (one prime example is improvements in runtime techniques such as just-in-time compilation). Additionally, algorithms are improving as well – the need for solving hard problems such as Boolean satisfiability problem (SAT), formal verification or optimization (be it convex or integer linear problem optimizations) started a flurry of research resulting in many new heuristics applicable to solving real-world problems as opposed to their theoretical worst cases.

These developments suggest that today we can use more computational power and solve much harder problems at the fractional cost compared to just a decade ago.

1.1.3.2 Software Development Costs

Unlike computing which gets cheaper and cheaper, the change in the cost of software development is nowhere near this trend. In fact, software developers' salaries keep increasing and are going to pass $100,000 per year [82]. One may argue that today's programmers are much more effective given the advances in programming languages, supporting tools and available libraries to the extent that it might take only a fraction of the time to develop the same software as years ago. However, today's programmers have to cope with increasingly complex systems and requirements such as high availability, online upgrades, better security, *etc.*. It is, therefore, no wonder that programming remains a difficult task and that even the best practices cannot keep all the bugs out of the software products. Addressing this problem is nowhere easy – after all, it is human nature to make mistakes. What we can do, however, is make it even easier for programmers to tend to their code – starting with tools which help identify bugs in the programs, through tools which ensure that the code running in production is not behaving unexpectedly, to platforms that are abstracting and handling a part of the problem. Programmers need all the help they can get to improve their productivity and to identify and remove bugs in their software.

1.2 Simplifying Development and Management of SDN

The driving force behind this thesis is the fact that while we have excessive amounts of computational power available, we rarely use it to improve the life of software programmers and operators. This is particularly true for networking that is quite late in adjusting to the software-development mindset and thus lacks the tools for this jobs. Moreover, SDN is still in its infancy. As such, it is the prime time for researchers to have a significant impact on the industry — companies are still struggling to grasp all the implications, design changes and new exciting ways SDN networks might work.

With this motivation in mind, the main goal of this dissertation is to show that *today's abundance of computational power can greatly benefit both SDN programmers and operators.* In particular, the steady increase of computational power, as well as the recent advances in the field of satisfiability solvers, can be used to build tools which use the vast amount of available computing resources to improve correctness, reliability and performance of SDN. We believe that having such tools will have a positive influence on SDN well beyond helping programmers and operators. By helping programmers debug SDN controllers quicker we enable faster and cheaper development with fewer bugs, which in turn moves SDN quicker to its full deployment. Similarly, providing tools for monitoring the network is a key enabler for operators to resolve problems quickly and keep their networks running with minimal disruption.

No tool, however great, will be successful if it was hard to use. Therefore, a part of our goal is to design tools which require just minimal modifications to the existing SDN infrastructure. For example, if a programmer wants to test an SDN controller, she should be able to do this without needing to change the controller's code. Similarly, tools for monitoring and optimizing network updates should be easily integrated into an existing network.

1.2.1 Solution Overview

Towards our goal of making it simple to develop and manage SDN networks, this thesis describes three practical tools – NICE, Monocle, and ESPRES. These three tools address development and management difficulties in various parts of SDN.

We start with the observation that testing SDN controllers is hard. In particular, while SDN uses a centralized controller, the fact the SDN controller needs to coordinate many switches at the same time makes it an exemplary distributed system with all the consequences of its distributed nature. Our first tool, NICE, addresses the problem of testing SDN. NICE tries to uncover insidious and concurrency-related bugs in SDN controllers and thus it is a great benefit for the programmers.

As a testing and bug finding tool, NICE does not have to act in a real time. Instead, NICE can leverage seconds to minutes to even hours of offline computation, for example as a part of automated tests or just programmers running it over night. This allows NICE to thoroughly and systematically explore various states through which the controller and the network could evolve as the time progresses; NICE checks whether any of these executions leads to a buggy behavior. To explore the evolution of possible network state, NICE employs and combines two well-known techniques – model checking and symbolic execution described in more detail in Chapter 2. The main benefit of NICE is that it systematically tries different scenarios, often ones which contain a complex interleaving of events that was not envisioned by the programmers.

While NICE helps with the controller correctness, this is only a part of the smooth and errorless operation of the network. The main reason is that the packets in the network are ultimately moved around by the SDN switches and any switch malfunction, therefore, results in potential problems. While the controller can detect most such failures by observing that the affected switch stopped responding and/or indicated an error, there is an important class of problems that cannot be detected this way. Notably, a switch may fail in a silent way. This might be caused by a buggy implementation, ambiguities in the specification, or simply by random processes such as memory corruption.

Our tool Monocle is designed to address this issue by monitoring if the switch data plane behavior corresponds to what the controller installed in the control plane. Monocle computes and injects into the network special probe packets, and then it observes whether these packets are correctly forwarded (and optionally rewritten) by the switch. Unfortunately, generating probe packets might be tricky because of the possibly complicated overlaps between switch rules. Therefore, Monocle utilizes a satisfiability solver to find a correct probe for each rule. Finally, we design Monocle to be fast – on par with the switch update rate. This allows Monocle to see updates in flagrante and report to the controller that the update is installed in a data plane. Such service is a great benefit for consistent update techniques [68].

Finally, as SDN substantially enhances network programmability, we expect that the rate of network changes will keep increasing. As an example, frequent traffic engineering such as the one proposed in MicroTE [17] might require re-balancing the traffic several times per minute. Moreover, each act of re-balancing might require moving a large number of flows. This could be therefore a time-consuming operation, especially considering the fact that today's hardware switches are quite heterogeneous and sometimes slow in their performance [55].

With ESPRES, we strive to compensate for these inefficiencies by tackling update installation as the problem of scheduling which operations the SDN controller should send to which switch at any given time. In particular, while the total time of the update might be bounded by the bottleneck switch, we observe that a network update often can be considered as consisting of several sub-goals, one for each flow. We demonstrate that by taking into account these sub-goals, one can easily improve certain aspects of the whole update such as (i) a number of flows moved to a new path at intermediate times, and (ii) the mid-update rule overhead[1] of the update. ESPRES works by queuing the whole update sent by the controller in a virtual queue. This is important because it allows ESPRES to later decide on the exact order of the commands that will be sent to the switches (re-ordering of commands that are already sent to the switch is not possible). ESPRES then observes the current state of the switches (*i.e.*, the progress they are making in the rule installation) and uses one of our several heuristics to decide which command to pick next and send to each switch. It is because of this real-time nature that ESPRES can

[1] Certain types of updates might transiently keep both old and new rules installed on a switch, *e.g.* [68].

adapt to a variety of scenarios even if switches have unpredictable performance characteristics.

1.2.2 Thesis Contributions

The key contributions of this dissertation are:

1. I present the design and implementation of NICE, an automated bug-finding tool for testing the correctness of SDN controllers. The NICE prototype tests unmodified applications written in Python for the NOX platform.

    - NICE introduces a novel combination of model checking (to explore the evolution of network state under different network event orderings), symbolic execution (to find relevant packets that trigger different controller behavior), and domain-specific search strategies (to further reduce the space of event orderings)

    - The performance evaluation shows that: (*i*) even on small examples, NICE is five times faster than approaches that apply state-of-the-art tools, (*ii*) the OpenFlow-specific search strategies reduce the state space by up to 20 times, and (*iii*) the simplified switch model brings a 4-fold reduction on its own.

    - I apply NICE to three real OpenFlow applications and uncover *13* bugs. Most of the bugs I found are design flaws, which are inherently less numerous than simple implementation bugs. In addition, at least one of these applications was tested using unit tests.

2. I present the design and implementation of Monocle, a data plane correspondence monitoring tool that can operate on fine-grained timescales needed in SDN. In particular, Monocle goes beyond the state-of-the-art in its ability to quickly recompute the required monitoring information during a rule update.

    - I formulate a set of formal constraints the monitoring packets must satisfy. I handle unicast, multicast, ECMP, drop rules, rule deletions and modifications while carefully treating rule overlaps. When necessary, I provide proofs that my theoretical foundation is correct. I also optimize the conversion of the constraints a probe needs to satisfy into a form presented to an off-the-shelf SAT solver.

- I go beyond the state-of-the-art by describing how to convert the probe solution (computed in abstract header space) into a real packet. Moreover, I prove that the conversion is sound.

- I minimize Monocle's overhead (extra flow table space) by formulating and solving a graph vertex coloring problem. Our study shows that only several extra rules per switch suffice in most topologies.

- The evaluation demonstrates that Monocle: (*i*) detects failed rules and links in a matter of seconds while monitoring a 1000-rule flow table in a hardware switch, (*ii*) ensures truly consistent network updates by providing accurate feedback on rule installation with only several ms of delay, (*iii*) takes between 1.48 and 4.03 ms on average to generate a probe packet on two datasets, (*iv*) typically has small overhead in terms of additional packets being sent and received, and (*v*) works with larger networks as shown by delaying an installation of 2000 flows by only 350ms.

- I describe the experience of using Monocle showing that it can reveal switch problems that were previously unknown.

3. I present the design and implementation of ESPRES, a system that takes a big network update (*i.e.*, an update reconfiguring multiple flows) and then, to improve certain aspects of the network update, on-the-fly re-orders and schedules switch commands corresponding to this update.

   - I introduce a *per-switch virtual message queue* and describe how to manage it to minimize service times of requests sent to the switch. This enables ESPRES to easily re-order operations sent to the switch and thus optimize the update according to the current progress.

   - I observe that big network updates can be split into independent *sub-updates* and these sub-updates can be scheduled independently of each other.

   - I propose several online heuristics to schedule sub-updates. The heuristics aim to optimize different aspects of the whole update.

- The early results show that compared to a no-scheduler baseline, different ESPRES schedulers can achieve (i) 2.17-3.88 times quicker sub-update completion time for the 20th percentile of sub-updates and 1.27-1.57 times quicker for the 50th percentile, or (ii) cause only 3.5-17% intermediate switch rule overhead as compared to 62% without any scheduler.

1.3 Thesis Organization

The rest of this thesis is organized as follows: First, in Chapter 2 we provide the necessary background information on Software-Defined Networking, as well as software techniques for systematic problem solving/verification that we build upon in this thesis. Next, in Chapter 3 we describe a way to systematically check the correctness of OpenFlow controllers. Afterward, in Chapter 4 we look at the correctness problem from a different perspective — we develop a method of verifying that the switch data plane processes packets as configured by the control plane. Chapter 5 delves into the performance aspect of SDN updates; here we discuss automated ways to improve certain aspects of the update performance. Chapter 6 positions this thesis with the respect of the related work. Finally, we conclude the thesis and provide ideas for future work in Chapter 7.

1.4 Previously Published Work

This thesis is based on material previously published in the following peer-reviewed conference and workshop papers:

Chapter 3 includes material from

- M. Canini, D. Venzano, P. Perešíni, D. Kostić and J. Rexford. *A NICE Way to Test OpenFlow Applications.* The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2012.

- P. Perešíni, M. Kuźniar, M. Canini, D. Venzano and D. Kostić et al. *Systematically testing OpenFlow controller applications.* Computer Networks (Elsevier), 2015.

Chapter 4 builds on top of

- P. Perešíni, M. Kuźniar and D. Kostić. *Monocle: Dynamic, Fine-Grained Data Plane Monitoring.* The 11th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), Heidelberg, Germany, December 1-4 2015.
- P. Perešíni, M. Kuźniar and D. Kostić. *Dynamic, Fine-Grained Data Plane Monitoring with Monocle.* Under submission to IEEE/ACM Transactions on Networking, (2016).

Chapter 5 is based on

- P. Perešíni, M. Kuźniar, M. Canini and D. Kostić. *ESPRES: Transparent SDN Update Scheduling.* The Workshop on Hot Topics in Software Defined Networking (HotSDN).
- P. Perešíni, M. Kuźniar, M. Canini and D. Kostić. *ESPRES: Easy Scheduling and Prioritization for SDN.* Open Networking Summit (ONS) Research Track, 2014.

Chapter 2

Background

In this chapter, we present basic background information about Software Defined Networking and techniques to systematically test/verify software such as satisfiability solvers, model checking, and symbolic execution. A reader well versed in these topics might skip directly to chapter 3 although it is still advisable to skim over the content of this chapter to become acquainted with the terminology we use.

2.1 Computer Networks

2.1.1 Network Basics

A computer network is a collection of network nodes and links between them with the primary goal of enabling communication between the nodes. Nodes in the network can be split into two types: end hosts and network elements.

End hosts are the devices that want to communicate (*e.g.*, computers, mobile phones, *etc.*). Network elements, on the other hand, are placed in the network to facilitate the communication and come in different flavors (*e.g.*, hub, switch, router, *etc.*) depending on their exact function. The end hosts themselves communicate by exchanging messages. Because network communication is complex, its various functions are decomposed into several layers (known as OSI model) shown in Table 2.1. In the OSI model, each layer provides functionality to the layer above and uses layers below to exchange the messages; it does this by prepending layer-specific information (header) to the message from the layer above (payload). As such, at the data link layer, the frames exchanged contain a stack of nested headers from all protocols above. We will

Layer	Type of a message exchanged	Layer function
7. Application 6. Presentation 5. Session	data	High-level APIs. Translation of data between networking service and the application (*e.g.*, character encoding, compression, encryption, *etc.*). Managing communication sessions.
4. Transport	segment (TCP) datagram (UDP)	Reliable transmission of segments between any end nodes on a network.
3. Network	packet	Managing a multi-node network; includes addressing, routing and traffic control.
2. Data link	frame	Reliable transmission of frames between two nodes directly connected by a physical layer.
1. Physical	bit	Sending/receiving raw bitstreams over a physical medium.

Table 2.1: Network layers and their function (OSI model)

collectively refer to this nested header at layer 2 as a *packet header* and to the payload of transport layer as the *packet payload* (note that layers 5 to 7 are rarely split these days, rather, they are usually commonly referred to as an application layer).

Finally, network function can be split in another way between data and control planes. In this split, *data plane* is responsible for moving packets around the network whereas *control plane* is concerned with the question of what the data plane should do. Traditionally, the network elements collocate these two functions into a single device (*e.g.*, router), which then (*i*) forwards the traffic, and at the same time (*ii*) runs a logic that communicates with other routers and decides what to do. As we mentioned in the introduction, this has a potential drawback of complex non-trivial multi-device configuration in case this logic does not support the function the network operators want to achieve (*e.g.*, supporting only shortest-path routing versus custom traffic engineering).

2.1.2 Software Defined Networking

Software Defined Networking is a network architecture that, unlike traditional networks, separates the concepts of *data plane* forwarding (*e.g.*, moving packets across a network) and control plane configuration (*e.g.*, deciding what data plane forwarding should achieve) as depicted in Figure 1.1. As such, an SDN network consists of two major parts – an SDN controller which manages the network, and SDN switches which perform data plane forwarding, both of which communicate over a common protocol.

SDN switches: An SDN switch is a universal network element that performs data plane forwarding. Unlike traditional networks where we make distinctions between different type of data plane forwarding (*e.g.*, switches for layer 2 forwarding, routers for layer 3 forwarding, or stateless middleboxes such as simple firewalls working on layer 4) we do not make such distinction in SDN as an SDN switch can perform forwarding across different layers (see Table 2.2).

In particular, each SDN switch has one or more *flow tables.* Each flow table in the switch contains an ordered set list of flow entries (commonly referred to as *rules*) describing the processing of packets by the switch. Table 2.3 shows an overview of the rule structure: Each rule consists of a *match* (specifying which values of header fields this rule applies to) and *actions* (such as forwarding, dropping, flooding, or modifying the packets, or sending them to the controller). Rules further contain a *priority* (to distinguish between rules with overlapping patterns; if a packet matches multiple rules, the switch applies actions of the highest-priority rule only), and a timeout (indicating whether/when the rule expires). A match pattern can require an "exact match" on all relevant header fields (*i.e.*, a *microflow* rule), or have "don't care" bits in some fields (*i.e.*, a *wildcard* rule). For each rule, the switch maintains traffic counters that measure the number of bytes and packets processed so far. When a packet arrives, the packet matching process starts at the first flow table and may continue to additional flow tables. When matching a packet against a given flow table, the switch selects the highest-priority matching rule, updates the traffic counters, and performs the specified action(s). As an example of this process, consider a flow table with the following three rules (ordered by decreasing priority):

1. $R_{high} = match(EthType = IP, IP\_proto = TCP, TCP\_dport = 22) \rightarrow drop$
2. $R_{mid} = match(EthType = IP, IP\_dst = 1.2.3.4) \rightarrow forward(port_1)$
3. $R_{low} = match(EthType = IP, IP\_dst = 1.2.3.4) \rightarrow forward(port_2)$

For this flow table, all SSH packets (TCP destination port 22) will be dropped, all non-SSH packets destined to host with IP 1.2.3.4 will be forwarded to port 1, and no packet will be forwarded to port 2 because any packet matching R_{low} will be already covered by R_{high} or R_{mid}.

	Match										Action
	Switch port	**MAC src**	**MAC dst**	**Eth type**	**VLAN ID**	**IP src**	**IP dst**	**IP proto**	**TCP sport**	**TCP dport**	
L2 switching (switch)	*	*	01:23:45:67:89:ab	*	*	*	*	*	*	*	port2
Blocking traffic (firewall)	*	*	*	IP	*	*	*	TCP	*	22	drop
Routing (router)	*	*	*	IP	*	*	5.6.7.8	*	*	*	port2
Microflow switching (no traditional equivalent)	port1	aa:bb:cc:dd:ee:ff	01:23:45:67:89:ab	IP	-	1.2.3.4	5.6.7.8	TCP	12345	80	port2

Table 2.2: An illustration how a single SDN switch could replace different types of traditional network elements. Depending on the flow table rules, an SDN switch can act as an L2 switch, a router, a firewall or even have a new custom behavior. Moreover, the behavior can be mixed, *e.g.*, the switch can perform L2 forwarding for some packets but L3 forwarding for others.

Priority	**Match**	**Actions**	**Timeout**	**Counters**
Specifies the order in which the rules are matched	Specifies which packets this rule applies to	Specifies what the switch should do with a packet that matches this rule	Specifies how long should the switch keep this rule	Keeps statistics about the number of packets and bytes that were processed by this rule

Table 2.3: An overview of information that an SDN switch keeps for each flow table entry (rule).

SDN controller: Unlike typical networks where management and configuration is spread over the network elements (*e.g.*, individual configuration of traditional switches, routers, firewalls, *etc.*) an SDN network has a centralized programming model; that is, there is one (or a few) software controller(s) managing the underlying switches. To avoid vendor lock-in, the controller typically runs on a commodity server machine near the switches it controls.

A controller can be further split into two parts – (*i*) a controller platform (*e.g.*, NOX [36], POX [8], Ryu [10], OpenDaylight [4], ONOS [2], *etc.*) that manages the common part of interaction with the switches (*e.g.*, setting up connections, encoding/decoding messages, *etc.*), and (*ii*) one or more controller application(s) running on top of this platform and performing the business logic that network administrators want to achieve.

To achieve the business objectives, the controller application (which is a general-purpose program that can perform arbitrary computation and maintain arbitrary state), typically interacts with the switches (via the controller platform) by installing and uninstalling rules in the switches, reading traffic statistics collected by the switches, and responding to network events. For each event type (*e.g.*, packet arrival, rule timeout, switch join, *etc.*), the controller application defines a handler that typically reacts to the situation at hand; for example, a handler may install new rules or issue new requests for traffic statistics as a reaction to new flows appearing in the network).

A common idiom for controller applications popularized by Ethane [24] is to manage the network *reactively*, *i.e.*, the controller application sets up new flows in the network as they arrive. In a reactive setup when a new flow arrives at the switch, the switch informs the controller about the new flow via a “packet arrival” event. The controller responds to the packet arrival by installing a rule for handling subsequent packets directly in the data plane. Sending packets to the controller introduces overhead and delay, so most applications try to minimize the fraction of traffic that must go to the controller.

Reactive networking allows for very fine-grained policies because the controller decides what to do with each of the network flows. This is in a contrast with a fully *proactive* setup where the controller usually sets up a coarse wildcard flows before any traffic enters the network. Proactive setup is more similar to traditional routing, albeit the controller has much higher control over what parts of the traffic follow what paths in the network. All in all, although the term SDN is usually more associated with the reactive style of managing the network, the SDN architecture does not dictate whether the controller should use reactive or proactive, or even a mixed way of managing the network.

Communication protocol between the switches and the controller: The SDN controller and switches have to communicate over a well-defined protocol. One such protocol that gained popularity over the past years is *OpenFlow* [5]. Through OpenFlow, the controller configures and manages the switch, receives events from it and sends packets out the switch.[1] The OpenFlow protocol itself is an application-layer protocol, *i.e.*, it requires a channel over which it exchanges messages (typically, the channel is TCP, TLS or Unix domain sockets in case of a virtual switch running on the same machine as the controller). Instead, OpenFlow deals with specifying an exact format of messages that the switches and the controller can exchange. There are three types of OpenFlow messages: (*i*) controller-to-switch, (*ii*) asynchronous (switch to the controller), and (*iii*) symmetric (can be sent both by a switch and a controller). Here, we give an overview of the most important messages and the curious reader is referred to the full specification for more details.

`Hello`, `FeaturesRequest`, `FeaturesResponse` are messages used during the connection setup. Controller applications usually do not need to take special care of them as the controller platform should be responsible for this part of the controller-switch interaction.

`FlowMod` is perhaps the most important message. `FlowMod` is a controller-to-switch message that tells the switch to install, modify or remove rule(s) from its data plane. A `FlowMod` command consists of a `command` (insert, modify or delete a rule), a `match`, a `priority` and `actions`. Modify and delete com-

[1] The controller might want to send packets such as LLDP probes/responses, ARP and DNS responses, *etc.*.

mands come in two varieties – “strict” (*e.g.*, they act only on the exact rule described by the match) or “non-strict” (*e.g.*, they update/delete all rules that match wildcards of the flow modification). A typical application of non-strict commands is all-wildcarded delete sent by some controllers at the beginning to bulk-clear the switch flow table of any residual rules.

`PacketIn` message is sent by a switch after it receives a packet which is supposed to be sent to the controller. PacketIn contains a reason field which explains why the switch sent the packet to the controller; the possible reason values are “no-match” (*i.e.*, no rule in the flow table matched this packet), or “action” (*i.e.*, the controller explicitly specified forwarding the packet to the controller as a part of some rule’s action list).

`PacketOut` message is a controller-to-switch message which instructs the switch to send out a given packet from the switch. Typically, this can be used by the controller to inject data plane packets such as LLDP requests/responses, ARP responses, DNS responses, *etc.*. In our work we use PacketOut to inject probe packets into the data plane (Chapter 4).

`FlowStatsRequest` message is a controller-to-switch message that tells the switch to report the flow counters associated with the rules specified in the `FlowStatsRequest` message. The switch replies with a `FlowStatsResponse` message listing for each flow its match and the associated counter statistics. FlowStats is, therefore, an important message for controllers wishing to monitor flows in the network.

`BarrierRequest` is a controller-to-switch message which functions as a serialization primitive — a switch needs to process all messages received before `BarrierRequest` before it can send a `BarrierReply` and only then it can continue processing other messages (note that without barriers, an OpenFlow switch is free to re-order commands at will to improve performance. It is, therefore, crucial to use barriers between any commands which depend on each other). Unfortunately, the OpenFlow specification is rather unclear whether the processing needs to happen before sending `BarrierReply` is only in the control plane (*e.g.*, guarantee that there will be no error in installing a rule) or in the data plane as well (*e.g.*, switch actually forwards packets according to the installed rule). As we show in Chapter 4, some switches do not fully implement the barrier command and therefore, a careful monitoring of the switch behavior is required.

2.2 Systematic Solving and Verification Techniques

In this section, we briefly discuss techniques and tools used to verify software and to systematically solve satisfiability and optimization problems. These techniques are an integral part of today's computing as many difficult problems can be solved with their help — this thesis is a proof of it.

2.2.1 Constraint Solvers

2.2.1.1 SAT

Boolean satisfiability problem (commonly abbreviated as SAT) is one of the cornerstones of theoretical computer science. SAT is the problem of determining whether there exists an interpretation that satisfies a given boolean formula. In other words, SAT asks whether the variables of a boolean formula can be assigned values $True$ or $False$ such that the formula evaluates to $True$. In practice, the boolean formula in SAT is often restricted to be in the *conjunctive normal form* (CNF), *i.e.*, the formula is a conjunction (logical AND) of *clauses* (logical OR of variables or their negations). As an example, formula $(a \vee \neg c) \wedge (b \vee a \vee c) \wedge (\neg b)$ is in the CNF form but formula $(a \wedge b) \vee c$ is not.

The importance of SAT, which is one of the first problems proven to be NP-complete, is that many hard problems are easily reducible to SAT (*e.g.*, by formulating them as boolean formulas over variables). Despite SAT being an NP-complete problem, many good heuristics were developed over the last decade for solving SAT on real-world problems. There is even an annual SAT competition [11] in which different implementations compete on a set of real-world SAT instances. In Chapter 4 we will show how one can use SAT to systematically test whether a switch data plane is behaving correctly.

2.2.1.2 SMT

An extension of SAT solvers are satisfiability modulo theory (SMT) solvers. SMT solvers work not only over boolean formulas but also over specialized theories. In this sense, SMT solvers are working on a representation which is closer to the real world than plain SAT. For example, popular SMT solvers such as Z3 [27] and STP [32] support both a theory of bit-vector integers and a theory of integer-addressed arrays of variables. It is, therefore, possible to encode constraints such as $(x * y == 4) \wedge (array[x] == 5) \wedge (array[y] == 4)$ with a solver providing one of the two valid solutions $x = 1,\ y = 2,\ array[1] = 5,\ array[4] = 4$ or $x = 4,\ y = 1,\ array[1] = 4,\ array[4] = 5$ (the assignment $x = 3,\ y = 3$ does not satisfy array access constraints as $(array[2] == 5) \wedge (array[2] == 4)$ is not satisfiable).

SMT solvers work by encoding the constraints from a given theory into a plain SAT instance and using a SAT solver underneath. However, a straightforward conversion is often inefficient – SMT solvers thus use various techniques to preprocess the constraints in an effort to reduce the size and complexity of the final SAT instance, or even progressively refine the conversion (an example of such optimization is bit-blasting [32] for efficient bit-vector support).

In this thesis, we use SMT solvers for exploring program paths with symbolic execution (Chapter 3) as well as a potential backend for solving constraints on packet header fields (Chapter 4).

2.2.1.3 Integer Linear Programs

Linear programming (LP) is another widely-used technique for solving a broad range of problems. However, unlike SAT and SMT, linear programming is designed to solve optimization problems. As such, linear programming can not only find a valid solution for a given set of constraints, but it can also efficiently find the solution that minimizes/maximizes a given objective.

A linear program, as its name suggests, is a set of constraints in a linear form, *i.e.*, it can be written as a set of several inequalities between a linear combination of variables and a constant. An example linear program would be

$$\begin{aligned} x + 2y + 3z &\geq 3 \\ 2x + z &\geq 5 \\ y &\leq -3 \end{aligned}$$

with an objective to minimize $x + z$.

A variation of the linear program – an integer linear program (ILP, sometimes also called mixed integer programming MIP) – is simply achieved by restricting some variables to be only integer numbers. While this does not seem like a big change, it changes the complexity with which the problem can be solved – while a linear program can be solved in a polynomial time, adding integer constraints makes the problem NP-complete. This can be easily seen by converting SAT to ILP as one can simply encode binary variables as integer values $0 \leq x_i \leq 1$ and encode each clause of as a sum of variables or their negations, *e.g.*, encoding $x_1 \vee \neg x_2 \vee x_3$ as $x_1 + (1 - x_2) + x_3 > 0$. The fact that ILP can be used to solve SAT (albeit not as efficiently as a dedicated SAT solver) is particularly useful in scenarios where we need to find a feasible solution to the SAT problem but at the same time optimize some criterion. Through this thesis, we will use this aspect of ILP to find an optimal solution for different auxiliary tasks (*e.g.*, minimizing the number of switch labels while avoiding collisions between neighboring switches in Chapter 3, and calculating an optimal offline schedule for a network update in Chapter 5).

2.2.2 Model Checking

Model checking is a formal method for verifying correctness of finite-state systems. In its essence, model checker takes a model of a system and automatically and exhaustively checks whether all possible evolutions of the system in time adhere to the given safety and liveness properties.

2.2.2.1 Modeling the State of a System

Typically, a model of a distributed system consists of multiple *components* that communicate asynchronously over *message channels, i.e.*, first-in-first-out buffers (*e.g.*, see Chapter 2 of [15]). Each component has a set of variables, and the component state is an assignment of values to these variables. The *system state* is the composition of the component states (at a given instant of time). To capture in-flight messages, the system state also includes the contents of the channels.

2.2.2.2 Evolution of the System State in time

The state of a system changes over time, *e.g.*, by the system sending or receiving new messages. A *transition* represents such a change from one state to another (*e.g.*, due to sending a message). At any given state, each component maintains a set of enabled transitions, *i.e.*, the state's possible transitions. For each state, the enabled system transitions are the union of enabled transitions at all components. A *system execution* corresponds to a sequence of these transitions, and thus specifies a possible behavior of the system.

```
pending_states = []
explored_states = []
errors = []
initial_state = create_initial_state()
for t in initial_state.enabled_transitions:
    pending_states.push([initial_state, t])
while len(pending_states) > 0:
    state, transition = choose(pending_states)
    try:
        next_state = run(state, transition)
        if next_state in explored_states:
            continue # already explored
        check_properties(next_state)
        explored_states.add(next_state)
        for t in state.enabled_transitions:
            pending_states.push([next_state, t])
      except PropertyViolation as e:
          errors.append([e, trace])
```

Figure 2.1: Pseudo-code of the basic model-checking loop.

2.2.2.3 Model-Checking Process

Given a model of the state space, performing a search is conceptually straightforward. Figure 2.1 shows the pseudo-code of the basic model-checking loop. First, the model checker initializes a stack of states with the initial state of the system. At each step, the checker chooses one state from the stack and one of its enabled transitions. After executing this transition, the checker tests the correctness properties on the newly reached state. If the new state violates a correctness property, the checker saves the error and the execution trace. Otherwise, the checker adds the new state to the set of explored states (unless the state was added earlier) and schedules the execution of all transitions enabled in this state (if any).

The model checker can run until the stack of states is empty, or until detecting the first error.

2.2.2.4 Model-Checking Optimizations

While basic model checking is a sound approach to verifying correctness, it suffers from a state-space explosion problem — the bigger the system is, there are exponentially more possible states. There are multiple potential ways to mitigate this state-space explosion.

First of all, model simplification is the most common remedy to this problem. As its name suggest, model simplification works by taking the model of a system and removing parts of it that are not essential to the property being verified. While this works well, the potential downside of model simplification is that it is often a manual and tedious process. Moreover, it might be error-prone as oversimplifying the model might yield both false positives and false negatives.

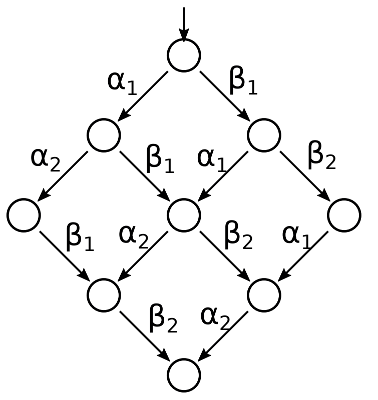
The second way to mitigate state-space explosion, or more precisely the number of transitions explored by the model checker is to never re-explore the same state. To do this, the model checker must be able to check whether the state reached from the current transition was already explored or not, *i.e.*, the model checker must be able to explicitly track all of the system state. In order to save memory, the model checker might also decide to only store a hash of the canonical representation of the state — seeing the same hash means (with high probability) that we already explored that state.

Finally, it might not be necessary to explore all of the system states to prove that system works (or find an error) as there might be a redundancy between the information that we can learn from the different system states. One of such techniques is described in the next section.

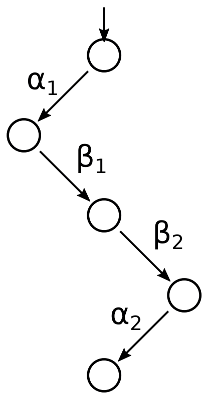
2.2.2.5 Partial Order Reduction

Partial-Order Reduction (POR) is a technique that reduces the size of the state space to be searched by a model-checking algorithm. In this section, we give a brief overview of how the POR technique works (for more in-depth review of POR, we kindly redirect the reader to Chapter 8 of [15]).

In its essence, POR takes advantage of the fact that many events that are happening in the system are independent, *i.e.*, they commute. It is therefore enough to explore just one arbitrary order of such events. As an example, consider a model of two concurrent systems A and B that do not communicate with each other. If the size of the state-space of A is $|A|$ and the size of state-space of B is $|B|$, then the state of the system AB is $|A| * |B|$. Checking all states of a parallel system such as AB is therefore obviously undesirable as the state-space quickly explodes for even a moderate number of such systems. Instead, if the systems A and B are really independent, it is enough to explore just a single ordering of the independent events as depicted in Figure 2.2. Of course, in reality, systems A and B are rarely completely independent.



(a) **Exploration of all states**



(b) **Sufficient exploration if α_1, α_2 is independent of β_1, β_2**

Figure 2.2: If two systems A and B are independent, it is enough to explore only a single ordering between events of these systems.

Instead, these systems might have some events happening concurrently that are independent and some events which affect each other.

Formally, let T be a set of transitions; we say transitions $t_1 \in T$ and $t_2 \in T$ are independent (adapted from [30]) if for all states s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'.

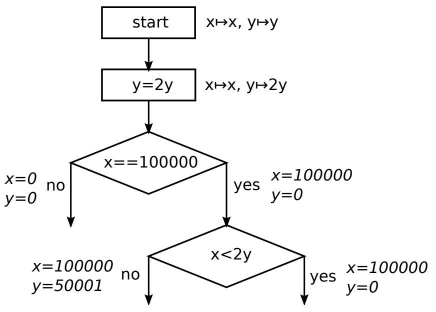
2. if t_1 and t_2 are enabled in s, then there is a state s' that $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$.

In other words, independent transitions do not enable or disable each other and if they are both enabled they commute. If two transitions are not independent, they are dependent. Further, transitions that are dependent are *worth reordering* because they lead to different system behaviors.

In practice, to check if transitions are independent, POR algorithms use the notion of *shared objects*. Transitions that use the same shared object are dependent and the ones that use disjoint sets of shared objects are independent. Identifying the right granularity for shared objects is crucial because if the choice is too conservative, transitions that are not worth reordering would be considered dependent and make POR less effective (*e.g.*, an extreme case of a shared object is shared global state). On the other hand, missing a shared object generally means that the search is no longer complete.

```
1 void f(int x, int y) {
2    y = 2*y;
3    if (x == 100000) {
4        if (x < y) {
5            assert(0); /* error */
6        }
7    }
8 }
```

(a) Example code snippet



(b) Exploration

Figure 2.3: Symbolic execution of a code snippet. Symbolic execution engine explores all paths, collects constraints along them and queries an SMT solver to check if a path is feasible.

2.2.3 Symbolic Execution

Symbolic execution is a systematic method that systematically explores all *feasible* code paths through a program. By exploring all feasible code paths, symbolic execution can figure out if there is a specific code path that leads to an assertion or other error. In such case, symbolic execution can provide an example of the input values such that the program follows the specific path. Symbolic execution is therefore very useful as a bug-finding tool.

2.2.3.1 Symbolic execution

Symbolic execution works by running the tested program with *symbolic variables* (*i.e.*, apriori-unspecified values) as inputs. The symbolic-execution engine tracks the use of these symbolic variables and records their assignments as well as constraints on their possible values. For example, in line 2 of Figure 2.3, the engine learns that $y \mapsto 2y$ (*e.g.*, the current value of variable y maps to $2y$). At any branch (*e.g.*, `if` statement), the engine queries an SMT solver for two assignments of symbolic inputs—one that satisfies the branch predicate and one that satisfies its negation (*i.e.*, takes the "else" branch)—and logically forks the execution to follow the feasible paths. For example, the engine determines that to reach line 5 of Figure 2.3, the value of y (currently mapped as $y \mapsto 2y$) must be greater than x. The symbolic execution engine then asks an SMT solver to solve the constraint $(x = 100000) \wedge (x < 2y)$. Because there exists a solution (one example is $x = 100000$, $y = 50001$), the symbolic execution engine concludes that it is possible to reach the assertion and reports this example as an error.

2.2.3.2 Concolic Execution

A particular variant of symbolic execution is concolic execution[2] first introduced by DART [35] and CUTE [75] and further extended by KLEE [21]. Concolic execution tries to address the problem of symbolic execution that once the constraint becomes too complex for the constraint solver, symbolic execution cannot proceed exploring this path (as it does not know whether the path is feasible or not). Concolic execution works around this problem by executing the program with concrete inputs and tracing the execution. This way, the concolic engine can easily track all constraints on symbolic variables. The concolic execution engine then chooses one branch that it tries to negate, and uses the solution as a new concrete input for the program. This way, the concolic execution engine never reports any false positives (if we reach an assertion with concrete inputs, we have a failing example) while at the same time it might try to work around complex (unsolvable) constraints by simplifying them (in the worst-case the new inputs might lead to an unanticipated path through a program).

2.2.3.3 Limitations of Symbolic and Concolic Execution

Unfortunately, symbolic/concolic execution does not scale well with the size of the program; this is because the number of code paths can grow exponentially with the number of branches and the size of the inputs. Also, symbolic execution does not explicitly model the state space, which can cause repeated exploration of the same system state unless expensive and possibly undecidable state-equivalence checks are performed. In addition, despite exploring all code paths, symbolic execution does not explore all system execution paths, such as different event interleavings (techniques exist that can add artificial branching points to a program to inject faults or explore different event orderings [20,71], but at the expense of extra complexity.)

As such, symbolic execution alone is insufficient for testing of OpenFlow applications. Instead, our tool NICE (Chapter 3) uses model checking to explore system execution paths (and detect repeated visits to the same state [49]), and symbolic execution to determine which inputs would exercise a particular state transition.

[2] *Concolic* stands for *conc*rete+symb*olic*.

Chapter 3

Testing SDN Controllers With NICE

While lowering the barrier for introducing new functionality into the network, Software-Defined Networking (SDN) also raises the risks of software faults (or *bugs*). Even today's networking software—written and extensively tested by equipment vendors, and constrained (at least somewhat) by the protocol standardization process—can have bugs that trigger Internet-wide outages [1, 9]. In contrast, programmable networks offer a much wider range of functionality, through software created by a diverse collection of network operators and third-party developers. SDN therefore heavily depends on having effective ways to test applications in pursuit of achieving high reliability. In this chapter, we present NICE, a tool that efficiently uncovers bugs in OpenFlow programs, through a combination of model checking and symbolic execution.

3.1 Bugs in OpenFlow Applications

On the surface, the centralized programming model of SDN should reduce the likelihood of bugs. Yet, a software-defined network is inherently distributed and asynchronous, with events happening at multiple switches and inevitable delays affecting communication with the controller. Moreover, the installed rules might interact in an unanticipated fashion, either between the different switches, between the switches and the controller or even at the same switch.

Multiple packets of a flow reaching the controller: A common idiom in programming SDN networks is to direct a packet to the controller, and then install a rule for the switches to handle the remaining packets of a flow in

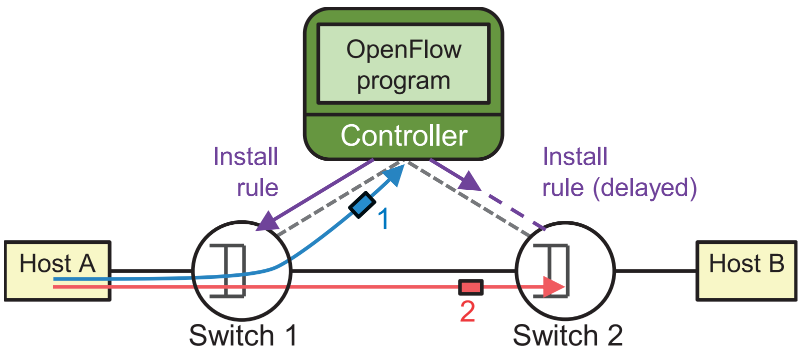


Figure 3.1: Due to delays between the controller and switches, the packet may not encounter an installed rule in the second switch.

the data plane. Yet, a race condition can easily arise if additional packets arrive at the switch while the controller is in the middle of installing the rule. Because the rule is not installed yet, these packets subsequently arrive at the controller as well. A program that implicitly expects to see just one packet (*i.e.*, the first packet of a flow) may behave incorrectly when multiple packets arrive because of the race condition [23]. For example, imagine a program that intends to directs all packets in a flow to the same randomly-selected server. The arrival of a second packet may trigger the application to install a second rule that directs packets to a different server replica. The program would behave correctly in the common case where the subsequent packets enter the network only after the rule is installed, but break if a burst of packets arrives at the controller.

No atomic update across multiple switches: Many applications need to install rules at multiple switches (*e.g.*, to direct the packets over a particular path through the network). These rules are not installed atomically, so some switches may start applying new rules before other switches have installed their rules. This can lead to the unexpected behavior shown in Figure 3.1, where an intermediate switch may encounter a packet that must go to the controller for handling. Implicitly assuming that rules are installed atomically can, therefore, lead to subtle bugs that only manifest themselves under certain packet timings and rule-installation delays. And while installing rules from "back to front" (from the end of the path to the beginning) can prevent this mistake, the programmer may not choose to install the rules this way.[1]

Previously-installed rules limit the controller's visibility: The con-

[1] Even worse, the switches may not reliably indicate when they finished installing rules. We address this problem with Monocle in Chapter 4.

troller program is really just one part of a distributed system that includes the processing performed by the underlying switches. Installing a rule (*e.g.*, that forwards or drops all matching packets) not only dictates what processing a *switch* performs, but also what packets the *controller* sees in the future! For example, imagine a program implementing a learning switch. Installing a wildcard rule to forward traffic based only on the destination MAC address would keep the controller from seeing some packets sent by new source MAC addresses—preventing the network from "learning" how to reach these addresses. While still successfully delivering traffic, this program would lead to inefficient delivery (*e.g.*, via unnecessary flooding) in some corner cases.

Composing functions that affect the same packets: Networks often perform multiple tasks that affect the handling of the same packets. For example, routing determines which path carries each packet (*e.g.*, based on destination IP address), and monitoring determines which packets should be grouped together for accumulating statistics (*e.g.*, based on TCP port number). Combining functionality is complicated, potentially involving the "cross product" of the rules needed for each function independently. OpenFlow switches rely on rule priority to disambiguate between overlapping groups of packets (*e.g.*, to ensure a rule with destination address 1.2.3.4 and port 80 gets precedence over another rule that matches all traffic to destination 1.2.3.4). Subtle mistakes in setting the priorities can lead to a program that operates correctly except for certain packets, or packets arriving in a particular order.

Interaction with end-host software: Some controller applications rely on implicit assumptions about the software running on the end host (*e.g.*, new TCP connections start with a SYN packet, or a Web download idle for more than 60 seconds has completed). These applications may have subtle bugs that only arise when hosts generate traffic that violates these assumptions. For example, imagine a server load-balancing application that directs client traffic to different Web server replicas (*e.g.*, sending traffic from source IP addresses starting with 0 to one replica and starting with 1 to another) [88]. Any changes to the load-balancing policy should ensure any *ongoing* TCP connection completes on the same server. By installing a rule that temporarily directs traffic to the controller, the application could inspect the *next* packet of each flow to install a microflow rule directing new flows (*i.e.*, if the packet is a SYN) to a new server and ongoing flows (*i.e.*, if the next packet is not a SYN) to the old server. However, the TCP state machine allows a host to retransmit SYN packets, raising the possibility of duplicate SYN packets which could lead the application to wrongly classify an ongoing connection as new.

3.2 Challenges of Testing SDN Applications

Testing SDN applications is challenging because controller behavior depends on the larger execution environment. The end-host applications sending and receiving traffic—and the switches handling packets, installing rules, and generating events—all affect the program running on the controller. The need to consider the larger environment leads to an extremely large state space, which “explodes” along three dimensions:

Large space of switch state: Switches run their own programs that maintain state, including the many packet-processing rules and associated counters and timers. Further, the set of packets that match a rule depends on the presence or absence of other rules, due to the “match the highest-priority rule” semantics. As such, testing OpenFlow applications requires an effective way to capture the large state space of the switch.

Large space of event orderings: Network events, such as packet arrivals and topology changes, can happen at any switch at any time. Due to communication delays, the controller may not receive events in order, and rules may not be installed in order across multiple switches. Serializing rule installation, while possible, would significantly reduce application performance. As such, testing OpenFlow applications requires efficient strategies to explore a large space of event orderings.

Large space of input packets: (Reactive) SDN Applications are *data-plane* driven, *i.e.*, controller applications must react to a huge space of possible packets. Controller applications may react differently to different packets (basing their decisions on packet header fields such as MAC addresses, IP addresses, TCP/UDP port numbers, *etc.*) and based on the switch port the packet arrived to. Moreover, the controller can perform arbitrary processing based on all of this information and modify the network state accordingly. This means that in order to explore all distinct network behaviors, one needs to consider combinations of potentially all header fields of packets arriving at the controller. As such, testing OpenFlow applications requires effective techniques to deal with the large space of inputs.

To simplify the problem, we could require programmers to use domain-specific languages that prevent certain classes of bugs [31, 64, 86]. However, the adoption of new languages is difficult in practice. Not surprisingly, most OpenFlow applications are written in general-purpose languages, like Java and Python.

Alternatively, developers could create abstract models of their applications, and use formal-methods techniques to prove properties about the system. However, these models are time-consuming to create and easily become out-of-sync with the real implementation. In addition, existing model-checking tools like SPIN [40] and Java PathFinder (JPF) [85] cannot be directly applied because they require explicit developer inputs to resolve the data-dependency issues and sophisticated modeling techniques to leverage domain-specific information. They also suffer state-space explosion, as we show in Section 3.9. Instead, we argue that testing tools should operate directly on *unmodified* OpenFlow applications, and leverage *domain-specific knowledge* to improve scalability.

3.3 NICE Overview

To address the scalability challenges, we present NICE (*No bugs In Controller Execution*)—a tool that tests unmodified controller programs[2] by automatically generating carefully-crafted streams of packets under many possible event interleavings. To use NICE, the programmer supplies the controller program and the specification of a topology with switches and hosts. The programmer can instruct NICE to check for generic correctness properties such as no forwarding loops or no black holes, and optionally write additional, application-specific correctness properties (*i.e.*, Python code snippets that make assertions about the global system state). By default, NICE systematically explores the space of possible system behaviors and checks them against the desired correctness properties. The programmer can also configure the desired search strategy. In the end, NICE outputs property violations along with the traces to reproduce them.

The whole process of using NICE is summarized in Figure 3.2. Finally, on top of automatic systematic execution, programmers can also use NICE as a simulator to perform manually-driven, step-by-step system executions or random walks on system states. By combining these two features – gathering event traces that lead to bugs and step-by-step execution – developers can effectively debug their applications.

Our design uses explicit state, software model checking [50,63,85,91] to explore the state space of the entire system—the controller program, the OpenFlow switches, and the end hosts—as discussed in Section 3.4. However, applying

[2] NICE only requires access to the controller state.

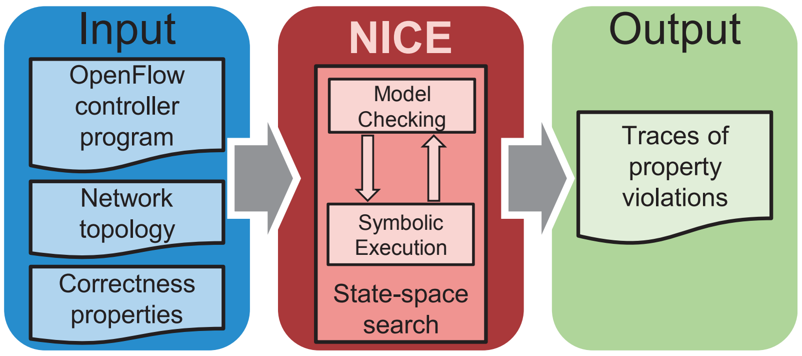


Figure 3.2: NICE usage overview. Given an OpenFlow program, a network topology, and correctness properties, NICE performs a state-space search and outputs traces of property violations.

model checking "out of the box" does not scale. While simplified models of the switches and hosts help, the main challenge is the event handlers in the controller program. These handlers are data dependent, forcing model checking to explore all possible inputs (which doesn't scale) or a set of "important" inputs provided by the developer (which is undesirable). Instead, we extend model checking to *symbolically execute* [20,21] the handlers, as discussed in Section 3.5. By symbolically executing the packet-arrival handler, NICE identifies equivalence classes of packets—ranges of header fields that determine unique paths through the code. NICE feeds the network a representative packet from each class by adding a state transition that injects the packet. Finally, to reduce the space of event orderings, we propose several domain-specific search strategies that generate event interleavings that are likely to uncover bugs in the controller program, as discussed in Section 3.6.2.

Bringing these ideas together, NICE combines model checking (to explore system execution paths), symbolic execution (to reduce the space of inputs), and search strategies (to reduce the space of event orderings). The programmer can specify correctness properties as snippets of Python code that assert global system state, or select from a library of common properties, as discussed in Section 3.7.

3.4 Model Checking a Software-Defined Network

The execution of a controller program depends on the underlying switches and end hosts; the controller, in turn, affects the behavior of these components. As such, testing is not just a matter of exercising every path through the controller program—we must consider the state of the larger system. The requirements for systematically exploring the space of system states, and checking correctness in each state, naturally lead us to consider *model checking* techniques. To apply model checking, we need to identify the system states and the transitions between states.

3.4.1 Transition Model for OpenFlow Apps

Model checking relies on having a model of the system, *i.e.*, a description of the state space. This requires us to identify the states and transitions for each component—the controller program, the OpenFlow switches, and the end hosts. However, we argue that applying existing model-checking techniques imposes too much work on the developer and leads to an explosion in the state space.

3.4.1.1 Controller Program

Modeling the controller as a transition system seems straightforward. A controller program is structured as a set of event handlers (*e.g.*, packet arrival and switch join/leave for the MAC-learning application in Figure 3.3), that interact with the switches using a standard interface, and these handlers execute atomically. As such, we can model the state of the program as the values of its global variables (*e.g.*, `ctrl_state` in Figure 3.3), and treat event handlers as transitions. To execute a transition, the model checker can simply invoke the associated event handler. For example, receiving a packet-in message from a switch enables the `packet_in` transition, and the model checker can execute it by invoking the corresponding event handler.

However, the behavior of event handlers is often data-dependent. In line 9 of Figure 3.3, for instance, the `packet_in` handler assigns `mactable` only for unicast source MAC addresses, and either installs a forwarding rule or floods a packet depending on whether or not the destination MAC is known. This leads to different system executions. Unfortunately, model checking does not cope well with data-dependent applications (see Chapter 1 of [15]). Since enumerating all possible inputs is intractable, a brute-force solution would require developers to specify "relevant" inputs based on their knowledge of the application. Hence, a controller transition would be modeled as a pair consisting of an event handler and a concrete input. This is clearly undesirable.

```
ctrl_state = {} # State of the controller is a global variable

# Handles new packets
def packet_in(switch_id, inport, pkt, bufid):
  mactable = ctrl_state[switch_id]
  is_bcast_src = pkt.src[0] & 1
  is_bcast_dst = pkt.dst[0] & 1
  if not is_bcast_src: # learn source
    mactable[pkt.src] = inport
  if (not is_bcast_dst) and (mactable.has_key(pkt.dst)):
    # install rule to destination
    outport = mactable[pkt.dst]
    if outport != inport:
      match = {
          DL_SRC: pkt.src,
          DL_DST: pkt.dst,
          DL_TYPE: pkt.type,
          IN_PORT: inport,
      }
      actions = [(OUTPUT, outport)]
      install_rule(
          switch_id, match, actions,
          soft_timer=5, hard_timer=PERMANENT
      )
      send_packet_out(switch_id, pkt, bufid)
      return
  flood_packet(switch_id, pkt, bufid)

# Handles when a switch joins
def switch_join(switch_id, stats):
  if not ctrl_state.has_key(switch_id):
    ctrl_state[switch_id] = {}

# Handles when a switch leaves
def switch_leave(switch_id):
  if ctrl_state.has_key(switch_id):
    del ctrl_state[switch_id]
```

Figure 3.3: Pseudo-code of a MAC-learning switch, based on the `pyswitch` application bundled with the NOX platform. The `packet_in` handler learns the input port associated with non-broadcast source MAC addresses; if the destination MAC is known, the handler installs a forwarding rule and instructs the switch to send the packet according to that rule. Otherwise, it floods the packet.

NICE overcomes this limitation by using *symbolic execution* to automatically identify the relevant inputs, as discussed in Section 3.5.

3.4.1.2 OpenFlow Switches

To test the controller program, the system model must include the underlying switches. Yet, switches run complex software, and this is not the code we intend to test. A strawman approach for modeling the switch is to start with an existing reference OpenFlow switch implementation (*e.g.*, [3]), define its state as the values of all variables, and identify transitions as the portions of the code that process packets or exchange messages with the controller. However, the reference switch software has a large state (*e.g.*, several hundred KB), not including the buffers containing packets and OpenFlow messages awaiting service; this aggravates the state-space explosion problem. Importantly, such a large program has many sources of nondeterminism and it is difficult to identify them automatically [91].

Instead, we manually create a switch model that omits inessential details. Indeed, creating models of some parts of the system is common to many standard approaches for applying model checking. Further, in our case, this is a one-time effort that does not add burden to the user. Following the OpenFlow specification [5], we view a switch as a set of communication channels, transitions that handle data packets and OpenFlow messages, and a flow table.

Simple communication channels: Each channel is a first-in, first-out buffer. Packet channels have an optionally-enabled fault model that can drop, duplicate, or reorder packets, or fail the link. The channel with the controller offers reliable, in-order delivery of OpenFlow messages, except for optional switch failures. We do not run the OpenFlow protocol over SSL on top of TCP/IP, allowing us to avoid intermediate protocol encoding/decoding and the substantial state in the network stack.

Two simple transitions: The switch model consists of only `process_pkt` and `process_command` transitions — for processing data packets and OpenFlow messages, respectively. We enable these transitions if at least one packet channel or the OpenFlow channel is nonempty, respectively.

To match the controller program's expectations about the environment, our switch model includes buffers that temporarily store packets awaiting further instruction from the controller. However, to improve scalability, we do not include these buffers in our definition of the state space (the buffered packet is already a part of the controller's `packet_in` transition and the exact value of `buffer_id` is irrelevant as long as it is unique.) A final simplification we make is in the switch's `process_pkt` transition. Here, the switch dequeues the first packet from each packet channel, and processes *all* these packets according to the flow table. So, multiple packets at different channels are processed as a single transition. This optimization is safe because the model checker already systematically explores the possible orderings of packet arrivals at the switch.

Merging equivalent flow tables: A flow table can easily have two states that appear different but are semantically equivalent, leading to a larger search space than necessary. For example, consider a switch with two microflow rules. These rules do not overlap—no packet would ever match both rules. As such, the order of these two rules is not important. Yet, simply storing the rules as a list would cause the model checker to treat two different orderings of the rules as two distinct states. Instead, as often done in model checking (*e.g.*, [78]), we construct a canonical representation of the flow table that derives a unique order of rules by sorting them in a deterministic manner. Additionally, the switch flow table does not store any packet statistics. Instead, we automatically infer the values that might change the controller behavior in Section 3.5.

3.4.1.3 End Hosts

Modeling the end hosts is tricky, because hosts run arbitrary applications and protocols, have a large state, and have behavior that depends on incoming packets. We could require the developer to provide the host programs, with a clear indication of the transitions between states. Instead, NICE provides simple programs that act as clients or servers for a variety of protocols including Ethernet, ARP, IP, and TCP. These models have explicit transitions and relatively little state. For instance, the default client has two basic transitions—`send` (initially enabled; can execute C times, where C is configurable) and `receive`—and a counter of sent packets. The default server has the `receive` and the `send_reply` transitions; the latter is enabled by the former. NICE also includes a more realistic refinement of this model — a mobile host that includes the `move` transition that moves the host to a new $(switch, port)$ location. The programmer can also customize the models we provide, or create new models.

3.4.2 Limitations of the Strawman Model Checking

So far, it would appear that model checking is a good match for testing SDN networks. However, this is not the case, and we now discuss what we believe the two main issues are.

Developer input for data-dependent handlers. As a systematic exploration of all possible packet inputs is infeasible (*e.g.*, just enumerating all IPv4 source and destination takes 2^{64} combinations), the only way to bound the model checking search is for the developer to manually specify a set of "relevant" inputs. Our desire is to do better than that. We suggest to analyze the controller program code and automatically infer which inputs cause which distinct behavior. Intuitively, this means *uncovering the possible distinct data-dependent transitions* at the controller.

State-space explosion. Second, the scalability of model checking is severely limited by the state-space explosion problem, *i.e.*, the number of states may be too large in practice.

In part, this is because we use a real switch implementation: this has a lot of states due to implementation details that are not necessary for testing.

There are several traditional ways to mitigate this problem. State tracking, which avoids exploring previously encountered states, is the first remedy commonly employed. Partial order reduction can be applied in the cases where the relative ordering of transitions does not alter the final state; so, it explores with just one arbitrary sequence. Finally, model checking experts can find the appropriate abstractions to obtain simplified system models.

Before discussing how we manage the state-space explosion, we next report how the existing tools perform in our problem domain.

3.4.2.1 Can Existing Tools Cope?

We wish to understand how large of an issue state-space explosion is for us and what is causing it. Thus, we experiment with two state-of-the-art model checkers: SPIN [40] and JavaPathfinder (JPF) [85]. Here, we briefly summarize the results for SPIN and we refer the interested reader to the Section 3.9 for further details.

We use an OpenFlow-based MAC-learning switch controller. The system has two end hosts and two switches interconnected in a linear topology with the switches in the middle. The basic system behavior is that host 1 sends a packet to host 2; host 2 replies with another packet. We analyze the complexity of fully exploring the state space as we increase the number of packets that host 1 sends (concurrently).

Although SPIN reduces the state space by using an abstract model of the system and uses partial order reduction, already with 8 packets the exhaustive search reaches the memory limit of 64 GB in our machine after exploring ≈ 50 M states. This huge number of states for a tiny OpenFlow system is caused by the high level of *concurrency*—transitions enabled at different components are interleaved in every possible way.

Note that the process of uncovering transitions at the controller is important for thorough testing but aggravates state-space explosion. Therefore, to cope with the state-space explosion, we (*i*) propose a number of simplifications for the OpenFlow switch model and (*ii*) derive several search heuristics based on a domain-specific knowledge.

3.5 Extending Model Checking with Symbolic Execution

To systematically test the controller program, we must explore all of its possible code paths as the behavior of an event handler depends on the inputs (*e.g.*, the MAC addresses of packets in Figure 3.3). Rather than explore all possible inputs, NICE identifies which inputs would exercise different code paths through an event handler. Systematically exploring all code paths naturally leads us to consider *symbolic execution* (SE) techniques. In this section, we describe how we apply symbolic execution to controller programs. Then, we explain how NICE combines model checking and symbolic execution to explore the state space of the whole network effectively.

3.5.1 Symbolic Execution of Event Handlers

Applying symbolic execution to the controller event handlers is relatively straightforward, with two exceptions. First, to handle the diverse inputs to the `packet_in` handler, we construct *symbolic packets*. Second, to minimize the size of the state space, we choose a *concrete* (rather than symbolic) representation of controller state.

Symbolic packets. The main input to the `packet_in` handler is the incoming packet. To perform symbolic execution, NICE must identify which (ranges of) packet header fields determine the path through the handler. Rather than view a packet as a generic array of symbolic bytes, we introduce *symbolic packets* as our symbolic data type. A symbolic packet is a group of symbolic integer variables that each represents a header field. To reduce the overhead for the constraint solver, we maintain each header field as a lazily-initialized, individual symbolic variable (*e.g.*, a MAC address is a 6-byte variable), which reduces the number of variables. Yet, we still allow byte- and bit-level accesses to the fields. This also tells us whether the program is agnostic to particular protocols (*e.g.*, ignoring transport header fields), allowing us to select a simpler host model for generating the input packets. In some cases, we go even further and constrain the possible values of header fields (*e.g.*, the MAC and IP addresses used by the hosts and switches in the system model, as specified by the input topology).

Concrete controller state. The execution of the event handlers also depends on the controller state. For example, the code in Figure 3.3 reaches line 12 only for unicast destination MAC addresses stored in `mactable`. Starting with an empty `mactable`, symbolic execution cannot find an input packet that forces the execution of line 12; yet, with a non-empty table, certain packets could trigger line 12 to run, while others would not. As such, we must incorporate the global variables into the symbolic execution. We choose to represent the global variables in a concrete form. We apply symbolic execution by using these concrete variables as the initial state and by marking as symbolic the packets and statistics arguments to the handlers. The alternative of treating the controller state as symbolic would require a sophisticated type-sensitive analysis of complex data structures (*e.g.*, [49]), which is computationally expensive and difficult for a dynamically typed language like Python. In addition, having purely symbolic controller state could cause NICE to test spurious states that are not reachable in practice due to the constraints imposed by the larger environment.

3.5.2 Combining Symbolic Execution with Model Checking

With all of NICE's parts in place, we now describe how we combine model checking (to explore system execution paths) and symbolic execution (to re-

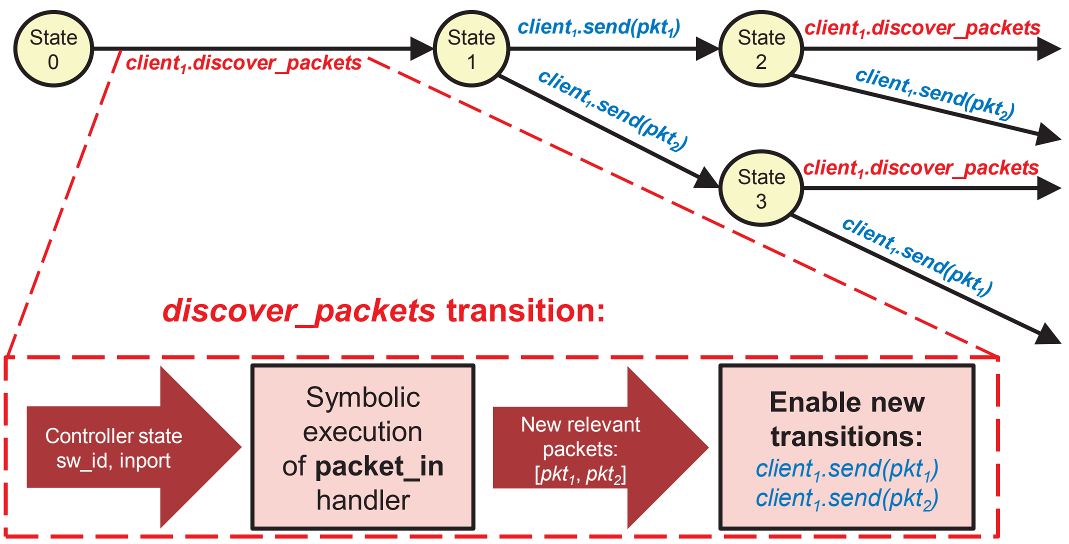

State 0
client₁.discover_packets
State 1
client₁.send(pkt₁)
client₁.send(pkt₂)
State 2
client₁.discover_packets
client₁.send(pkt₂)
State 3
client₁.discover_packets
client₁.send(pkt₁)
discover_packets transition:
Controller state sw_id, inport
Symbolic execution of packet_in handler
New relevant packets: [pkt₁, pkt₂]
Enable new transitions: client₁.send(pkt₁) client₁.send(pkt₂)

Figure 3.4: Example of how NICE identifies relevant packets and uses them as new enabled `send` packet transitions of $client_1$. For clarity, the circled states refer to the controller state only.

duce the space of inputs). At any given controller state, we want to identify packets that each client should send—specifically, the set of packets that exercise all feasible code paths on the controller in that state. To do so, we create a special client transition called `discover_packets` that symbolically executes the `packet_in` handler. Figure 3.4 shows the unfolding of controller's state-space graph.

Symbolic execution of the handler starts from the initial state defined by (*i*) the concrete controller state (*e.g.*, State 0 in Figure 3.4) and (*ii*) a concrete "context" (*i.e.*, the switch and input port that identify the client's location). For every feasible code path in the handler, the symbolic-execution engine finds an equivalence class of packets that exercise it. For each equivalence class, we instantiate one *concrete packet* (referred to as the relevant packet) and enable a corresponding `send` transition for the client. While this example focuses on the `packet_in` handler, we apply similar techniques to deal with traffic statistics, by introducing a special `discover_stats` transition that symbolically executes the statistics handler with symbolic integers as arguments. Other handlers, related to topology changes, operate on concrete inputs (*e.g.*, the switch and port ids).

Figure 3.5 shows the pseudo-code of our search-space algorithm, which extends the basic model-checking loop of Figure 2.1 in two main ways described next.

```
pending_states = []
explored_states = []
errors = []
initial_state = create_initial_state()
for client in initial_state.clients
    client.packets = {}
    client.enable_transition(discover_packets)
for t in initial_state.enabled_transitions:
    pending_states.push([initial_state, t])
while len(pending_states) > 0:
    state, transition = choose(pending_states)
    try:
        next_state = run(state, transition)
        ctrl = next_state.ctrl # Reference to controller
        ctrl_state = ctrl.get_state() # Serialized controller state
        for client in state.clients:
            if not client.packets.has_key(ctrl_state):
                client.enable_transition(discover_packets, ctrl)
        if process_stats in ctrl.enabled_transitions:
            ctrl.enable_transition(discover_stats, state, sw_id)
        check_properties(next_state)
        if next_state not in explored_states:
            explored_states.add(next_state)
            for t in next_state.enabled_transitions:
                pending_states.push([next_state, t])
    except PropertyViolation as e:
        errors.append([e, trace])

def discover_packets_transition(client, ctrl):
    sw_id, inport = switch_location_of(client)
    new_packets = SymbolicExecution(ctrl, packet_in,
                                    context=[sw_id, inport])
    client.packets[state(ctrl)] = new_packets
    for packet in client.packets[state(ctrl)]:
        client.enable_transition(send, packet)

def discover_stats_transition(ctrl, state, sw_id):
    new_stats = SymbolicExecution(ctrl, process_stats, context=[sw_id])
    for stats in new_stats:
        ctrl.enable_transition(process_stats, stats)
```

Figure 3.5: The state-space search algorithm used in NICE for finding errors. The highlighted parts, including the special "*discover*" transitions, are our additions to the basic model-checking loop of Figure 2.1.

Initialization (lines 5-7): For each client, the algorithm (*i*) creates an empty map for storing the relevant packets for a given controller state and (*ii*) enables the `discover_packets` transition.

Checking process (lines 14-20): Upon reaching a new state, the algorithm checks for each client (line 17) whether a set of relevant packets already exists. If not, it enables the `discover_packets` transition. In addition, it checks (line 19) if a `process_stats` transition is enabled in the newly-reached state, meaning that the controller is awaiting a response to a previous query for statistics. If so, the algorithm enables the `discover_stats` transition.

Invoking the `discover_packets` (lines 29-35) and `discover_stats` (lines 37-40) transitions allows the system to evolve to a state where new transitions become possible—one for each path in the packet-arrival or statistics handler. This allows the model checker to reach new controller states, allowing symbolic execution to again uncover new classes of inputs that enable additional transitions, and so on.

By symbolically executing the controller event handlers, NICE automatically infers the test inputs for enabling model checking without developer input, at the expense of some limitations in coverage of the state space which we discuss in Section 3.11.

3.6 Reducing the State Space Search

Even with our optimizations from the last two sections, the model checker cannot typically explore the entire state space, since it may be prohibitively large or even infinite. In this and the next section, we describe both general-purpose and domain-specific techniques for reducing the search space.

3.6.1 Dynamic Partial Order Reduction

To apply Partial Order Reduction (Section 2.2.2.5) in our context, we identify the following shared objects:

1. **input buffers of switches and hosts:** All transitions that read from or write to a buffer of a given node (send or receive a packet) are dependent.

2. **switch flow tables:** The order of switch flow table modifications is important because it matters if a rule is installed before or after a packet

matching that rule arrives at the switch. However, we exploit the semantics of flow tables to identify more fine-grained shared objects and hence make POR more effective. In particular, a pair of transitions that only read from the flow table are independent. Also, transitions that operate on rules with non-overlapping matches are independent. All remaining transition combinations are dependent.

3. **the controller application:** Because NICE treats the controller state as a single entity all transitions related to the controller are dependent. No further specialization is possible without relying on more fine-grained analysis of the controller state.

Rather than relying on a priori knowledge about what shared objects are used by which transition, we use a dynamic variant of POR, namely Dynamic Partial-Order Reduction (DPOR) [30], which gathers the dependency information at run-time while executing transitions, and modifies the set of enabled transitions at each visited state accordingly.

We base our DPOR algorithm on the one presented in [30] and extend it to work for our context. The base DPOR algorithm relies on the fact that a single execution path ends when there are no enabled transitions. NICE typically bounds the search depth and uses state matching to avoid exploring the same state multiple times. Therefore, some search paths *end before* all possible transitions that could be worth reordering are considered by DPOR. This was already recognized as a problem by the authors of [93]. To avoid this problem, we conservatively consider all enabled transitions as worth exploring unless specifically marked otherwise.

We first introduce some notation: Let E_i be the set of transitions enabled in state s_i; let $SE_i \subseteq E_i$ be the set of transitions strongly enabled in s_i (*i.e.*, the transitions that cannot be ignored); let $D_i \subseteq E_i$ be the set of transitions enabled in s_i that are temporarily disabled by DPOR.

The search path is represented by a transition sequence: $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \xrightarrow{t_n} s_{n+1}$. For each transition t_j in this sequence, we keep track of the transition t_i that enabled t_j. In other words, we look for i such that $t_j \notin E_i$ and $t_j \in E_{i+1}$. Let $P(j) = i$, we call $t_{P(j)}$ a first level predecessor of t_j, $t_{P(P(j))}$ a second level predecessor of t_j, and so on.

```
1  def DPOR(current_path):
2      for t_i in current_path[1 : n-1]:
3          state = get_start_state(t_i)
4          for t_j in current_path[idx(t_i) + 1 : n]:
5              if is_worth_reordering(t_i, t_j):
6                  if t_j in state.enabled:
7                      t_k = t_j
8                  else:
9                      t_k = get_predecessor(state, t_j)
10                 state.strongly_enabled.add(t_k)
11                 state.force_order(t_k, (t_j, t_i))
12             elif t_j in state.enabled:
13                 state.unnecessary.add(t_j)
14
15 def get_next_transition(state):
16     for t in state.enabled:
17         if (t in state.unnecessary and
18                 t not in state.strongly_enabled):
19             state.enabled.remove(t)
20     return state.enabled.pop()
```

Figure 3.6: Pseudo-code of our variation of the DPOR algorithm. The `get_next_transition` method prunes the enabled actions and returns the next transition worth executing. The DPOR method runs when the search ends in a state with no enabled transitions.

Figure 3.6 presents the pseudo-code of our DPOR algorithm. The function `DPOR` is invoked for the current execution path once the model checker reaches a state s_n with no enabled transitions. For every transition t_i with $i \in [1, n]$, we identify all transitions t_j where $j \in [i+1, n]$ that t_i is worth reordering with (line 5). In this case, we want to enforce a new search path that, starting from s_i, would first execute t_j and then t_i (lines 6-11). If $t_j \in E_i$, we simply add t_j to SE_i. If $t_j \notin E_i$, we find a transition $t_k \in E_i$ that is a predecessor of t_j (line 9) and add it to SE_i (line 10). If t_i is not worth reordering with $t_j \in E_i$, we add t_j to D_i (line 13). Marking t_k for exploration is necessary to ensure search completeness, but it is not sufficient to efficiently prune states. After executing t_k, the model checker would again have two possible orderings of t_i and t_j. To achieve the desired state space reduction, we enforce that t_j is executed before t_i on this particular search path (line 11). Before selecting the next transition to explore in a given state, the `get_next_transition` function disables all transitions enabled in this state, that were considered not worth reordering while exploring other paths (line 18).

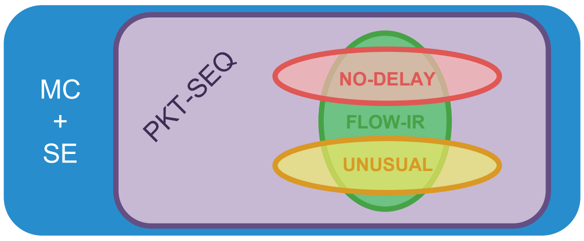


Figure 3.7: Illustration of the state space explored by our search strategies in relation to the entire state space.

3.6.2 OpenFlow-Specific Search Strategies

We propose several domain-specific heuristics that substantially reduce the space of event orderings while focusing on scenarios that are likely to uncover bugs. The general overview of the subspace of the state space explored by these heuristics is depicted in Figure 3.7.

PKT-SEQ: Limiting relevant packet sequences. The effect of discovering new relevant packets and using them as new enabled `send` transitions is that each end-host generates a potentially-unbounded tree of packet sequences. For example, for our `pyswitch` controller (Figure 3.3), NICE would just generate new, so far unseen, MAC addresses each time the MAC table changes.

To make the explored state space finite and smaller, PKT-SEQ heuristic reduces the search space by bounding the possible end host `send` transitions (and thus indirectly bounding the search tree). The heuristic bounds the search along two dimensions, each of which can be fine-tuned by the user. First, we limit *the maximum number of packets* an end host can send. Effectively, this also places a hard limit on the issue of infinite execution trees due to symbolic execution. Second, we limit the *maximum number of outstanding packets*, or in other words, the length of a packet burst. For example, if $client_1$ in Figure 3.4 is allowed only a 1-packet burst, this heuristic would disallow both `send(`pkt_2`)` in State 2 and `send(`pkt_1`)` in State 3. Effectively, this limits the level of "packet concurrency" within the state space. To introduce this limit, we assign each end host with a counter c; when $c = 0$, the end host cannot send any more packets until the counter is replenished. Because we are dealing with communicating end hosts, the default behavior is to increase c by 1 for every received packet. However, this behavior can be modified in more complex end host models, *e.g.*, to mimic the TCP flow and congestion controls.

NO-DELAY: Instantaneous rule updates. When using this simple heuristic, NICE treats each communication between a switch and the controller as a single atomic action (*i.e.*, not interleaved with any other transitions). In other words, the global system runs in "lock step". This heuristic is useful during the early stages of development to find basic design errors, rather than race conditions or other concurrency-related issues. For instance, it would allow the developer to realize that installing a rule prevents the controller from seeing other packets that are important for program correctness. For example, a MAC-learning application that installs forwarding rules based only on the destination MAC address would prevent the controller from seeing some packets with new source MAC addresses.

UNUSUAL: Unusual delays and reorderings. With this heuristic, NICE only explores event orderings with uncommon and unexpected delays aiming to uncover race conditions. For example, if an event handler in the controller installs rules in switches 1, 2, and 3, the heuristic explores transitions that reverse the order by allowing switch 3 to install its rule first, followed by switch 2 and then switch 1. This heuristic uncovers bugs like the example in Figure 3.1.

FLOW-IR: Flow independence reduction. Many OpenFlow applications treat different groups of packets independently; that is, the handling of one group is not affected by the presence or absence of another. In this case, NICE can reduce the search space by exploring only one relative ordering between the events affecting each group. To use this heuristic, the programmer provides `isSameFlow`, a Python function that takes two packets as arguments and returns whether the packets belong to the same group. For example, in some scenarios different microflows are independent, whereas other programs may treat packets with different destination MAC addresses independently.

Summary. PKT-SEQ is complementary to other strategies in that it only reduces the number of `send` transitions rather than the possible kind of event orderings. It is enabled by default and used in our experiments (unless otherwise noted). The other heuristics can be selectively enabled.

3.7 Specifying Application Correctness

Correctness is not an intrinsic property of a system—a specification of correctness states what the system should do, whereas the implementation determines what it actually does. NICE allows programmers to define correctness properties as Python code snippets, and provides a library of common properties (*e.g.*, no loops or black holes).

3.7.1 Customizable Correctness Properties

Testing correctness involves asserting safety properties (*“something bad never happens”*) and liveness properties (*“eventually something good happens”*), defined more formally in Chapter 3 of [15]. Checking for safety properties is relatively easy, though sometimes writing an appropriate predicate over all state variables is tedious. As a simple example, a predicate could check that the collection of flow rules does not form a forwarding loop or a black hole. Checking for liveness properties is typically harder because of the need to consider a possibly infinite system execution. In NICE, we make the inputs finite (*e.g.*, a finite number of packets, each with a finite set of possible header values), allowing us to check some liveness properties. For example, NICE could check that, once two hosts exchange at least one packet in each direction, no further packets go to the controller (a property we call “NoControllerInvolved”). Checking this liveness property requires knowledge not only of the system state, but also which transitions were executed.

To check safety and liveness properties, NICE allows correctness properties to (*i*) access the system state, (*ii*) register callbacks invoked by NICE to observe important transitions in system execution, and (*iii*) maintain local state. In our experience, these features offer enough expressiveness. For ease of implementation, the properties are represented as snippets of Python code that make assertions about the system state. NICE invokes these snippets after each transition. For example, to check the NoControllerInvolved property, the code snippet defines local state variables that keep track of whether a pair of hosts has exchanged at least one packet in each direction, and would flag a violation if a subsequent packet triggers a `packet_in` event. When a correctness check signals a violation, NICE records the execution trace that recreates the problem.

3.7.2 Library of Correctness Properties

NICE provides a library of correctness properties applicable to a wide range of OpenFlow applications. A programmer can select properties from a list, as appropriate for the application. Writing these correctness modules can be challenging because the definitions must be robust to communication delays between the switches and the controller. Many of the definitions must intentionally wait until a “safe” time to test the property to prevent natural delays from erroneously triggering a violation. Providing these modules as part of NICE can relieve the developers from the challenges of specifying correctness properties precisely, though creating any custom modules would require similar care.

- *NoForwardingLoops*: This property asserts that packets do not encounter forwarding loops. It is implemented by checking if each packet goes through any <switch, input port> pair at most once.

- *NoBlackHoles*: This property states that no packets should be dropped in the network, and is implemented by checking that every packet that enters the network and is destined to an existing host, ultimately leaves the network (for simplicity, we disable optional packet drops and duplication on the channels). In the case of host mobility, an extended version of this property — *NoBlackHolesMobile* — ignores the inevitably dropped packets in the period from when the host moves to when the controller can realize that the host moved.

- *WaitPathSetup*: This property checks that a given packet reaches the controller at most once. Essentially, this checks whether the controller properly finishes installing the path to the destination before releasing the packet back to the network. As an example, this check would detect the race condition described in Figure 3.1. The *WaitPathSetup* property is useful for many OpenFlow applications, though it does not apply to the MAC-learning switch, which requires the controller to learn how to reach both hosts before it can construct unicast forwarding paths in either direction.

- *NoControllerInvolved*: This property checks that, after two hosts have successfully delivered at least one packet of a flow in each direction, no successive packets reach the controller. This checks that the controller has established a direct path in *both* directions between the two hosts.

- *NoForgottenPackets*: This property checks that all switch buffers are empty at the end of system execution. A program can easily violate this property by forgetting to tell the switch how to handle a packet. This can eventually consume all the available buffer space for packets awaiting controller instruction; after a timeout, the switch may discard these buffered packets[3]. A short-running program may not run long enough for the queue of awaiting-controller-response packets to fill, but the *NoForgottenPackets* property easily detects these bugs. Note that a violation of this property often leads also to a violation of *NoBlackHoles*.

3.8 Implementation Highlights

We have built a prototype implementation of NICE written in Python so as to seamlessly support OpenFlow controller programs for the NOX platform (which provides an API for Python).

As a result of using Python, we face the challenge of doing symbolic execution for a dynamically typed language. This task turned out to be quite challenging from an implementation perspective. As pure symbolic execution would require us to modify the Python interpreter, we use concolic execution instead. Another consequence of using Python is that we incur a significant performance overhead, which is the price for favoring usability. We plan to improve performance in a future release of the tool.

NICE consists of three parts: (*i*) a model checker, (*ii*) a concolic-execution engine, and (*iii*) a collection of models including the simplified switch and several end hosts. We now briefly highlight some of the implementation details of the first two parts: the model checker and concolic engine, which run as different processes.

Model checker details. To checkpoint and restore system state, NICE takes the approach of remembering the sequence of transitions that created the state and restores it by replaying such sequence, while leveraging the fact that the system components execute deterministically. State-matching is done by comparing and storing hashes of the explored states. The main benefit of this approach is that it reduces memory consumption and, secondarily, it is simpler to implement. Trading computation for memory is a common approach for other model-checking tools (*e.g.*, [50, 91]). To create state hashes, NICE first normalizes the state and then it serializes it via the `json` module after which it applies the built-in hash function to the resulting string.

[3] In our tests of the ProCurve 5406zl OpenFlow switch, we see that, once the buffer becomes full, the switch starts sending the *entire contents* of new incoming packets to the controller, rather than buffering them. After a ten-second timeout, the switch deletes the packets that are buffered awaiting instructions from the controller.

Alternatively, NICE can be configured to store the serialized states themselves, at the cost of higher memory usage. This approach has a potential to reduce NICE running time, but the exact benefits depend mostly on the time required to save and restore the controller state. Moreover, saving the state required for DPOR is challenging and we do not support it in the current prototype.

Concolic execution details. A key step in concolic execution is tracking the constraints on symbolic variables during code execution. To achieve this, we first implement a new "symbolic integer" data type that tracks assignments, changes and comparisons to its value while behaving like a normal integer from the program point of view. Second, we reuse the Python modules that naturally serve for debugging and disassembling the byte-code to trace the program execution through the Python interpreter.

Further, before running the code symbolically, we normalize and instrument it since, in Python, the execution can be traced at best with single code-line granularity. Specifically, we convert the source code into its abstract syntax tree (AST) representation and then manipulate this tree through several recursive passes that perform the following transformations: (*i*) we split composite branch predicates into nested if statements to work around short-circuit evaluation (*i.e.*, explicitly encode Python's logic "if left-hand side of `and` operator is `False`, the right-hand side is not evaluated"; similarly for `or`), (*ii*) we move function calls contained in conditional expressions before the `if` statement to ease tracking of the symbolic variables through the conditional statements, (*iii*) we instrument branches to inform the concolic engine on which branch is taken, (*iv*) we substitute the built-in dictionary class `dict` with a special stub to track uses of symbolic variables, and (*v*) we intercept and remove sources of nondeterminism (*e.g.*, seeding the pseudo-random number generator). The AST tree is then converted back to a source code for execution.

3.9 Performance Evaluation

Here we present an evaluation of how effectively NICE copes with the large state space in OpenFlow.

3.9.1 Experimental setup.

We run the experiments on the simple topology of Figure 3.1, where the two end hosts behave as follows: host *A* sends a "*layer-2 ping*" packet to host *B*

Pings	Transitions	Unique states	ρ	CPU time
2	530 vs. 706	315 vs. 432	0.73	1.1 vs. 1.72 [s]
3	14,762 vs. 31,345	6,317 vs. 13,940	0.45	37.2 vs. 92.9 [s]
4	356,469 vs. 1,134,515	121,320 vs. 399,919	0.30	17 vs. 59 [m]
5	7,816,517 vs. 33,134,573	2,245,345 vs. 9,799,538	0.27	8 vs. 57 [h]

Table 3.1: Effectivity of the canonical switch state representation in NICE. We compare the state-space NICE needed to search without vs. with the canonical representation of the switch state on a MAC-learning switch application with ping-exchanging end hosts. Canonical representation enables recognizing equivalent switch states. Value ρ represents the ratio of the number of unique states.

which replies with a packet to A. The controller runs the MAC-learning switch program of Figure 3.3. We report the numbers of transitions and unique states, and the execution time as we increase the number of concurrent pings (a pair of packets). We run all our experiments on a machine with Linux 3.8.0 x86_64 that has 64 GB of RAM and a clock speed of 2.6 GHz. Our prototype does not yet make use of multiple cores.

3.9.2 Benefits of simplified switch model.

We first perform a full search of the state space using NICE as a depth-first search model checker (without symbolic execution) and compare it to doing model-checking without a canonical representation of the switch state. Effectively, this prevents the model checker from recognizing that it is exploring semantically equivalent states. These results, shown in Table 3.1, are obtained without using any of our search strategies. We compute ρ, a metric of state-space reduction due to using the simplified switch model, as

$$\rho = \frac{UniqueStates(\text{WITH-CANONICAL-REPRESENTATION})}{Unique(\text{NO-CANONICAL-REPRESENTATION})}$$

We observe the following:

• In both cases, the number of transitions and the number of unique states grow roughly exponentially (as expected). However, using the simplified switch model, the unique states explored with the switch canonical representation grow with a rate that is lower than the one observed without the canonical representation.

• The ratio ρ goes down with the problem size (number of pings) and is substantial (reduction of 3.7-times for four pings).

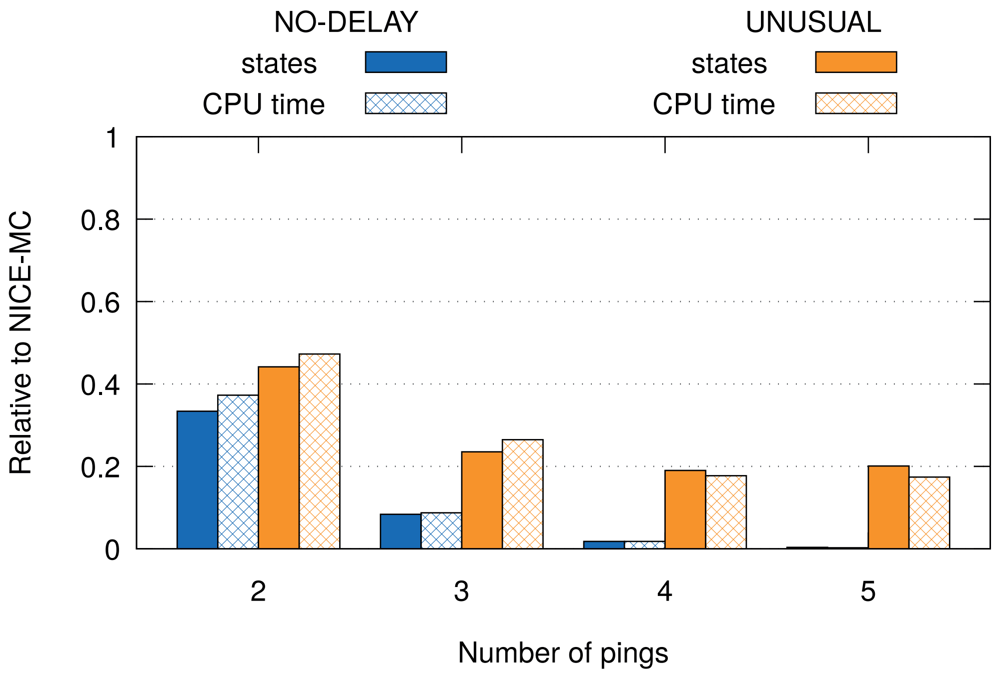


Figure 3.8: Running time and the number of states different heuristics (NO-DELAY, UNUSUAL) take relative to full search (NICE-MC).

• The time taken to complete a full state-space search in this small-scale example grows exponentially with the number of packets. However, thanks to symbolic execution, we expect NICE to explore most of the system states using a modest number of symbolic packets and even small network models.

3.9.3 Heuristic-based search strategies.

Figure 3.8 considers the same scenario as in the previous section (*i.e.*, host A pinging host B with multiple concurrent packets) and illustrates the contribution of NO-DELAY and UNUSUAL heuristics in reducing the search space relative to the metrics reported for the full search NICE-MC (we use the simplified switch model in all cases). The state space reduction is again significant; about factor of five and factor of ten for over two pings with UNUSUAL and NO-DELAY respectively. In summary, our switch model and these heuristics result in over 20-fold state space reduction for four and more pings. Finally, we also test the FLOW-IR heuristic. Unfortunately, in this scenario the heuristic is unable to reduce the state space because of the way the PySwitch works — as PySwitch learns new MAC addresses from all packets it receives, the flows are not independent. However, in Table 3.4 we show the effectiveness of this strategy in other scenarios (even 10 times reduction).

3.9.4 Comparison to other model checkers.

Next, we contrast NICE-MC with two state-of-the-art model checkers, SPIN [40] and JPF [85]. We create system models in these tools that replicate as closely as possible the system tested in NICE, using the same experimental setup as with our heuristics.

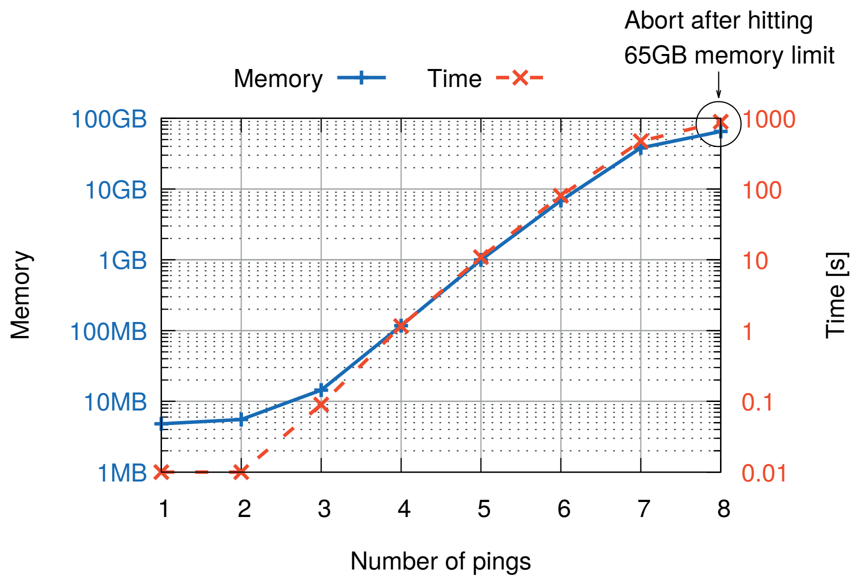
3.9.4.1 SPIN

SPIN is one of the most popular tools for verifying the correctness of software models, which are written in a high-level modeling language called PROMELA. This language exposes non-determinism as a first-class concept, making it easier to model the concurrency in OpenFlow. However, using this language proficiently is non-trivial and it took several person-days to implement the model of the simple OpenFlow system (Figure 3.1). To capture the system concurrency at the right level of granularity, we use the `atomic` language feature to model each transition as a single atomic computation that cannot be interleaved to any other transition. In practice, this behavior cannot be faithfully modeled due to the blocking nature of `channels` in PROMELA. To enable SPIN's POR to be most effective, we assign exclusive rights to the processes involved in each communication channel.

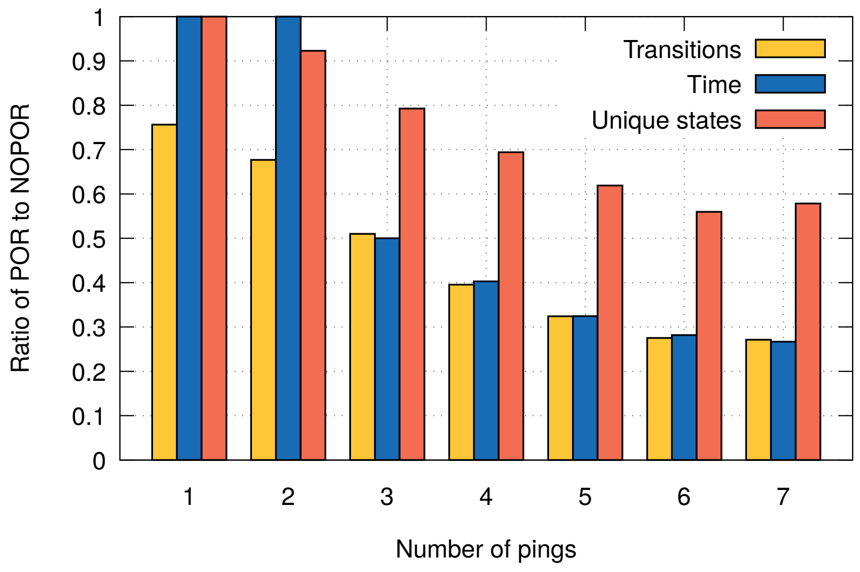
Figure 3.9a shows the memory usage and elapsed time for the exhaustive search with POR as we increase the number of packets sent by host A. As expected, we observe an exponential increase in computational resources until SPIN reaches the memory limit when checking the model with 8 pings (*i.e.*, 16 packets).

To see how effective POR is, we compare in Figure 3.9b the number of transitions explored with POR and without POR (NOPOR) while we vary the number of pings. In relative terms, POR's efficiency increases, although with diminishing returns, from 24% to 73% as we inject more packets that are identical to each other. The benefits due to POR on elapsed time follow a similar trend and POR can finish 6 pings in 28% of time used by NOPOR. However, NOPOR hits the memory limit at 7 pings, so POR only adds one extra ping.

Finally, we tried to see whether it would be possible to reduce SPIN's search space by taking advantage of the simple independence property as in FLOW-IR. Unfortunately, this is not possible as SPIN uses the accesses to communi-



(a) Memory usage and elapsed time with POR enabled.



(b) POR can reduce the number of explored transitions by up to 73%.

Figure 3.9: Results of the SPIN model checker on a network model representing the same scenario as Table 3.1. Exponential increase in computational resources partially mitigated by POR.

cation channels to derive the independence of events. Our DPOR algorithm instead considers a more fine-grained domain-specific definition of shared objects and thus could achieve additional state space reduction.

Pings	Time [s]	Unique states	End states	Mem. [MB]
1	0	55	2	17
2	9	20638	134	140
3	13689	25470986	2094	1021

(a) Thread-based model

Pings	Time [s]	Unique states	End states	Mem. [MB]
1	0	1	1	17
2	1	691	194	33
3	16	29930	6066	108
4	11867	16392965	295756	576

(b) Choice-based model

Table 3.2: Results of JPF model checker on a network model representing the same scenario as Table 3.1. Even the better choice-based model undergoes a big explosion of the state space.

3.9.4.2 Java PathFinder

Java PathFinder (JPF) is one among the first modern model checkers which use the implementation in place of the model. We follow two approaches to model the system by porting our Python code to Java.

In the first approach, we naively use threads to capture nondeterminism, hoping that JPF's automatic state-space reduction techniques would cope with different thread creation orders of independent transitions. However, in our case, the built-in POR is not very efficient in removing unnecessary network event interleavings because thread interleaving happens at a finer granularity than event interleavings. To solve this problem, we tune this model by using the `beginAtomic()` and `endAtomic()` functions provided by JPF. As this still produces too many possible interleavings, we further introduced a global lock.

In a second approach to further refine the model, we capture nondeterminism via JPF's choice generator: `Verify.getInt()`. This gives a significant improvement over threads, mainly because we are able to specify precisely the granularity of interleavings. However, this second modeling effort is nontrivial since we are manually enumerating the state space and there are several caveats in this case too. For example, explicit choice values should not be saved on the stack as the choice value may become a part of the global state, thus preventing reduction. The vector of possible transitions must also be sorted[4].

[4] We order events by their states' hash values.

Table 3.2a illustrates the state space explosion when using the thread-based model. Unfortunately, as shown in Table 3.2b, the choice-based model improves only by 1 ping the size of the model that we can explore within a comparable time period ($\approx$ 4 hours).

These results suggest that NICE, in comparison with the other model-checkers, strikes a good balance between (*i*) capturing system concurrency at the right level of granularity, (*ii*) simplifying the state space, and (*iii*) allowing testing of unmodified controller programs.

3.10 Experiences With Real Applications

In this section, we report our experience with applying NICE to three real applications—a MAC-learning switch, a server load-balancer, and energy-aware traffic engineering—and uncovering *13* bugs. In all experiments, it was sufficient to use a network model with at most three switches.

3.10.1 MAC-Learning Switch (PySwitch)

Our first application is the `pyswitch` software included in the NOX distribution (98 LoC). The application implements MAC learning, coupled with flooding to unknown destinations, common in Ethernet switches. Realizing this functionality under a centralized programming model seems straightforward (*e.g.*, the pseudo-code in Figure 3.3), yet NICE automatically detects three violations of correctness properties.

BUG-I: Host unreachable after moving. This fairly subtle bug is triggered when a host B moves from one location to another. Before B moves, host A starts streaming to B, which causes the controller to install a forwarding rule. When B moves, the rule stays in the switch as long as A keeps sending traffic, because the soft timeout does not expire. As such, the packets do not reach B's new location. This serious correctness bug violates the *NoBlackHoles* and *NoBlackHolesMobile* properties. If the rule had a *hard* timeout, the application would eventually flood packets and reach B at its new location; then, B would send return traffic that would trigger MAC learning, allowing future packets to follow a direct path to B. While this "bug fix" prevents persistent packet loss, the network still experiences *transient* loss until the hard timeout expires.

BUG-II: Unnecessary packets routed through the controller. The `pyswitch` also violates the *NoControllerInvolved* property, leading to suboptimal performance. The violation arises after a host A sends a packet to host B, and B sends a response packet to A. This is because `pyswitch` installs a forwarding rule in one direction — from the sender (B) to the destination (A), in line 13 of Figure 3.3. The controller does *not* install a forwarding rule for the other direction until seeing a subsequent packet from A to B. For a three-way packet exchange (*e.g.*, a TCP handshake), this performance bug directs 50% more traffic than necessary to the controller. Anecdotally, fixing this bug can easily introduce another one. The naïve fix is to add another `install_rule` call, with the addresses and ports reversed, after line 14, for forwarding packets from A to B. However, since the two rules are not installed atomically, installing the rules in this order can allow the packet from B to reach A before the switch installs the second rule. This can cause a subsequent packet from A to reach the controller unnecessarily. A correct fix installs the rule for traffic from A first, before allowing the packet from B to A to traverse the switch. With this fix, the program satisfies the *NoControllerInvolved* property.

BUG-III: Excess flooding. When we test `pyswitch` on a topology that contains a cycle, the program violates the *NoForwardingLoops* property. This is not surprising since `pyswitch` does not construct a spanning tree.

3.10.2 Web Server Load Balancer

Data centers rely on load balancers to spread incoming requests over service replicas. Previous work created a load-balancer application that uses wildcard rules to divide traffic based on the client IP addresses to achieve a target load distribution [88]. The application dynamically adjusts the load distribution by installing new wildcard rules; during the transition, old transfers complete at their existing servers while new requests are handled according to the new distribution. We test this application with one client and two servers connected to a single switch. The client opens a TCP connection to a virtual IP address corresponding to the two replicas. In addition to the default correctness properties, we create an application-specific property *FlowAffinity* that verifies that all packets of a single TCP connection go to the same replica. Here we report the bugs NICE found in the original code (1209 LoC), which had already been unit tested to some extent.

BUG-IV: ARP packets forgotten during address resolution. Having observed a violation of the *NoForgottenPackets* property for ARP packets, we identified two bugs. The controller program handles client ARP requests on behalf of the server replicas. Despite sending the correct reply, the program neglects to discard the ARP request packets from the switch buffer. This leads to the switch holding them in its buffers until they are deemed to expire.[5] A similar problem occurs for server-generated ARP messages.

BUG-V: TCP packets always dropped before the first reconfiguration. A violation of the *NoForgottenPackets* property for TCP packets allowed us to detect a problem where the controller ignores all `packet_in` messages for TCP packets caused by no matching rule at the switch. As before the first reconfiguration there are no rules installed, all flows that start during this period are ignored. Dropping such packets is understandable as the controller may have insufficient information about the replicas. However, the controller should drop them explicitly as ignoring them completely occupies space in switch buffers.

BUG-VI: Next TCP packet always dropped after reconfiguration. Having observed another violation of the *NoForgottenPackets* property, we identified a bug where the application neglects to handle the "next" packet of each flow—for both ongoing transfers and new requests—after any change in the load-balancing policy. Despite correctly installing the forwarding rule for each flow, the application does *not* instruct the switch to forward the packet that triggered the `packet_in` handler. Since the TCP sender ultimately retransmits the lost packet, the program does successfully handle each Web request, making it hard to notice this bug that degrades performance.

BUG-VII: Some TCP packets dropped during reconfiguration. After fixing previously described bugs, NICE detected another *NoForgottenPackets* violation due to a race condition. In switching from one load-balancing policy to another, the application sends multiple updates to the switch for each existing rule: (*i*) a command to remove the existing forwarding rule followed by (*ii*) commands to install one or more rules (one for each group of affected client IP addresses) that direct packets to the controller. Since these commands are not executed atomically, packets arriving between the first and second step do not match either rule. The OpenFlow 1.0 specification prescribes that packets

[5] The switch discards buffered packets after some time precisely to avoid bugs similar to this. However, until the packets are expired, they needlessly occupy precious buffer space.

that do not match any rule should go to the controller.[6] Although the packets go to the controller either way, these packets arrive with a different "reason code" (*i.e.*, `NO_MATCH`). As written, the `packet_in` handler ignores such (unexpected) packets, causing the switch to buffer them until they expire. This appears as packet loss to the end hosts[7]. To fix this bug, the program should reverse the two steps, installing the new rules (perhaps at a lower priority) before deleting the existing ones. Finding this bug poses another challenge: only a detailed analysis of event sequences allows us to distinguish it from **BUG-V**. Moreover, a proper fix of **BUG-V** hides all symptoms of **BUG-VII**.

BUG-VIII: Incomplete packets encapsulated in `packet_in` messages. The final *NoForgottenPackets* property violation was caused by a change in the OpenFlow specification before version 1.0. If a rule's action is to send packets to the controller, the action needs to define the maximum number of packet bytes that should be encapsulated in a `packet_in` message. The controller uses value 0, which in the initial versions of the specification means "encapsulate the entire packet". However, as of OpenFlow 1.0, value 0 is no longer special. As a result, the controller does not receive any bytes of the packet header and is unable to analyze it.

BUG-IX: Duplicate SYN packets during transitions. A *FlowAffinity* violation detected a subtle bug that arises only when a connection experiences a duplicate (*e.g.*, retransmitted) SYN packet while the controller changes from one load-balancing policy to another. During the transition, the controller inspects the "next" packet of each flow, and assumes a SYN packet implies the flow is new and should follow the new load-balancing policy. Under duplicate SYN packets, some packets of a connection (arriving before the duplicate SYN) may go to one server, and the remaining packets to another, leading to a broken connection. We are not the first to report this problem — the authors of [88] already acknowledged this possibility in Footnote 2 in their paper. However, they needed a careful consideration to discover this problem.

[6] In later versions of the specification, this behavior is actually configurable and the available options are: (*i*) drop, (*ii*) send to the controller, (*iii*) continue with the next flow table. In either case, the controller should explicitly configure this behavior by installing "table-miss" flow entry.

[7] To understand the impact, consider a switch with 1 Gb/s links, 850-byte frames, and a flow table update rate of 250 rules/s (as widely reported for the HP 5406zl [55, 70]). That would lead to 150 dropped packets per switch port.

3.10.3 Energy-Efficient Traffic Engineering

OpenFlow enables a network to reduce energy consumption [39, 83] by selectively powering down links and redirecting traffic to alternate paths during periods of lighter load. REsPoNse [83] pre-computes several routing tables (the default is two), and makes an online selection for each flow. The NOX implementation (374 LoC) has an *always-on* routing table (that can carry all traffic under low demand) and an *on-demand* table (that serves additional traffic under higher demand). Under high load, the flows should probabilistically split evenly over the two classes of paths. The application learns the link utilizations by querying the switches for port statistics. Upon receiving a packet of a new flow, the `packet_in` handler chooses the routing table, looks up the list of switches in the path, and installs a rule at each hop.

For testing with NICE, we install a network topology with three switches in a triangle, one sender host at one switch and two receivers at another switch. The third switch lies on the on-demand path. We define the following application-specific correctness property:

• *UseCorrectRoutingTable*: This property checks that the controller program, upon receiving a packet from an ingress switch, issues the installation of rules to all and just the switches on the appropriate path for that packet, as determined by the network load. Enforcing this property is important, because if it is violated, the network might be configured to carry more traffic than it physically can, degrading the performance of end-host applications running on top of the network.

NICE found several bugs in this application:

BUG-X: The first packet of a new flow is dropped. A violation of *NoForgottenPackets* and *NoBlackHoles* revealed a bug that is almost identical to **BUG-VI**. The `packet_in` handler installed a rule but neglected to instruct the switch to forward the packet that triggered the event.

BUG-XI: The first few packets of a new flow can be dropped. After fixing **BUG-X**, NICE detected another *NoForgottenPackets* violation at the second switch in the path. Since the `packet_in` handler installs an end-to-end path when the first packet of a flow enters the network, the program implicitly assumes that intermediate switches would never direct packets to the controller. However, with communication delays in installing the rules, the packet could reach the second switch before the rule is installed. Although these packets trigger `packet_in` events, the handler implicitly ignores them, causing the packets to buffer at the intermediate switch. This bug is hard to detect because the problem only arises under certain event orderings. Simply installing the rules in the reverse order, from the last switch to the first, is not sufficient — differences in the delays for installing the rules could still cause a packet to encounter a switch that has not (yet) installed the rule. A correct fix should either handle packets arriving at intermediate switches, or use barriers (where available) to ensure that rule installation completes at all intermediate hops before allowing the packet to depart the ingress switch.

BUG-XII: Only on-demand routes used under high load. NICE detects a *CorrectRoutingTableUsed* violation that prevents on-demand routes from being used properly. The program updates an extra routing table in the port-statistic handler (when the network's perceived energy state changes) to either always-on or on-demand, in an effort to let the remainder of the code simply reference this extra table when deciding where to route a flow. Unfortunately, this made it impossible to split flows equally between always-on and on-demand routes, and the code directed all new flows over on-demand routes under high load. A fix was to abandon the extra table and choose the routing table on a per-flow basis.

BUG-XIII: Packets can be dropped when the load reduces. After fixing **BUG-XI**, NICE detected another violation of the *NoForgottenPackets*. When the load reduces, the program recomputes the list of switches in each always-on path. Under delays in installing rules, a switch not on these paths may send a packet to the controller, which ignores the packet because it fails to find this switch in any of those lists.

3.10.4 Overhead of Running NICE

In Table 3.3, we summarize how many seconds NICE took (and how many state transitions were explored) to discover the *first property violation* that uncovered each bug, under four different search strategies with and without DPOR. Note the numbers are generally small because NICE quickly produces simple test cases that trigger the bugs. One exception, **BUG-IX**, is found in 1 hour by doing a PKT-SEQ-only search but NO-DELAY and UNUSUAL heuristics can detect it in just 3-8 minutes. Our search strategies are also generally faster than PKT-SEQ-only to trigger property violations, except in one case (**BUG-VI**).

Bug	only PKT-SEQ		NO-DELAY		FLOW-IR		UNUSUAL	
	no DPOR	DPOR	no DPOR	DPOR	no DPOR	DPOR	no DPOR	DPOR
BUG-I	2149 / 8.3	1631 / 7.14	418 / 1.6	355 / 1.34	2149 / 8.55	1631 / 7.61	994 / 3.88	819 / 3.17
BUG-II	540 / 1.65	415 / 1.18	187 / 0.62	156 / 0.4	540 / 1.70	415 / 1.21	241 / 0.8	179 / 0.45
BUG-III	23 / 0.22	23 / 0.16	17 / 0.2	17 / 0.16	23 / 0.21	23 / 0.16	24 / 0.2	24 / 0.18
BUG-IV	49 / 0.61	49 / 0.36	49 / 0.58	49 / 0.32	49 / 0.53	49 / 0.47	30 / 0.36	30 / 0.26
BUG-V	52 / 0.59	52 / 0.33	52 / 0.53	52 / 0.37	52 / 0.59	52 / 0.38	33 / 0.5	33 / 0.33
BUG-VI	738 / 6.65	312 / 2.25	1977 / 15.35	361 / 1.8	778 / 10.0	306 / 2.31	139 / 1.47	208 / 1.24
BUG-VII	12k / 93.14	1782 / 13.26	Missed	Missed	481 / 4.0	385 / 2.28	112 / 1.5	49 / 0.38
BUG-VIII	1274 / 12.12	245 / 3.74	1237 / 10.31	265 / 1.33	1134 / 11.22	278 / 2.15	709 / 6.81	200 / 1.54
BUG-IX	432.7k / 66m	33.7k / 324	33.5k / 213	15.6k / 111	Missed	Missed	53.5k / 478	13.4k / 138
BUG-X	22 / 0.31	22 / 0.19	22 / 0.24	22 / 0.21	22 / 0.24	22 / 0.21	22 / 0.24	22 / 0.19
BUG-XI	97 / 0.97	53 / 0.63	Missed	Missed	97 / 0.98	53 / 0.61	24 / 0.25	24 / 0.2
BUG-XII	19 / 0.27	19 / 0.21	17 / 0.22	18 / 0.17	19 / 0.23	19 / 0.21	19 / 0.25	19 / 0.19
BUG-XIII	3126 / 24.86	697 / 6.59	Missed	Missed	2287 / 18.54	617 / 5.59	1225 / 11.24	589 / 4.86

Table 3.3: Comparison of the number of transitions and running time until the first violation that uncovered each bug. Time is in seconds unless otherwise noted. Note that we use the same version of Load Balancer (containing a fix of BUG-IV) for bugs BUG-VII and BUG-V but we force NICE to explore the transitions in a different, namely reverse, order.

NO-DELAY takes longer for **BUG-VI** because the latter is faster to explore a sequence of transitions where the network reconfiguration event happens at the right time for experiencing a *NoForgottenPackets* violation. FLOW-IR does not produce benefits for several bugs — as we already explained, FLOW-IR does not work on PySwitch because the flows are not independent. FLOW-IR also does not help with the first three bugs in EATE because these are uncovered by test cases that do not involve using multiple flows. Adding DPOR improves all strategies unless the bug is found on one of the first explored paths. Also, note that there are no false positives in our case studies—every property violation is due to the manifestation of a bug[8]—and only in few cases (**BUG-VII**, **BUG-IX**, **BUG-XI** and **BUG-XIII**) the heuristic-based strategies experience false negatives. Expectedly, NO-DELAY, which does not consider rule installation delays, misses race condition bugs (3 missed bugs out of 13). **BUG-IX** is missed by FLOW-IR because the duplicate SYN is treated as a new independent flow (1 missed bug).

Finally, the reader may find that some of the bugs we found — like persistently leaving some packets in the switch buffer — are relatively simple and their manifestations could be detected with run-time checks performed by the controller platform. However, the programmer would not know what caused them. For example, a run-time check that flags a "no forgotten packets" error due to **BUG-VI** or **BUG-VII** would not tell the programmer what was special about this particular execution that triggered the error. Subtle race conditions are hard to diagnose, so having a (preferably small) example trace—like NICE produces—is crucial.

3.10.5 Effectiveness of Optimizations

Until now we reported only times to find the first invariant violation, which is critical when looking for bugs. However, to fully evaluate various optimizations described earlier, we disable all invariants and in Table 3.4 present a total number of transitions and running time for three configurations: MAC-learning controller like in Section 3.9 with 4 pings, Load Balancer with one connection, and Energy-Efficient Traffic Engineering with two connections (like for **BUG-XIII**).

[8] There is a *NoForgottenPackets* violation in the switch application for which it is difficult to discern if it is really a bug or not: the code explicitly discards LLDP packets and assumes they will be liberated from the switch buffer by other means. However, that might not happen so we still think the programmer should be aware of the consequences of his code. Therefore, we choose to report such case as a violation.

Strategy	**PySwitch**		**Load Balancer**		**REsPoNse**	
	Transitions	**Time [s]**	**Transitions**	**Time [s]**	**Transitions**	**Time [s]**
PKT-SEQ	356,469	1,739.42	84,831	486.88	62,152	541.32
PKT-SEQ + serialize	356,469	1,051.89	84,831	428.86	62,152	316.38
PKT-SEQ + DPOR	253,511	1,604.62	9,206	47.59	14,549	116.70
NO-DELAY	6,385	19.87	7,663	22.76	2,012	12.17
NO-DELAY + serialize	6,385	15.31	7,663	34.53	2,012	8.37
NO-DELAY + DPOR	4,962	20.22	2,841	9.15	834	5.78
FLOW-IR	356,469	1,587.34	8,636	41.02	32,352	251.76
FLOW-IR + serialize	356,469	1,058.00	8,636	40.35	32,352	167.88
FLOW-IR + DPOR	253,511	1,468.23	1,420	6.24	8,447	62.28
UNUSUAL	67,816	258.83	4,716	21.62	3,544	21.79
UNUSUAL + serialize	67,816	186.69	4,716	22.20	3,544	17.22
UNUSUAL + DPOR	21,659	111.93	1,406	8.11	1,072	6.12

Table 3.4: Comparison of the number of transitions and running time when using basic NICE, NICE with state serialization and NICE with DPOR. Except for the Load Balancer application for which serializing the controller state takes a lot of time, state serialization significantly reduces running time. Using DPOR results in even greater reductions except for the PySwitch application where many transitions are dependent and a small reduction in explored states is insufficient to balance the advantage of state serialization.

First, state serialization improves the performance and the improvement depends on the complexity of serializing the controller state. Load Balancer has a more complex state than the other two. For the controllers with a simpler state, the state serialization allows to finish the state space exploration in up to 40% less time.

DPOR reduces the number of transitions and states that the model checker needs to explore, however, it comes with two sources of overhead: (*i*) it performs additional computations, and (*ii*) in our implementation DPOR works does not work with state serialization. For this reason, in a network where many transitions are dependent and where serializing the controller is simple (learning switch) the benefits of using DPOR are smaller than the costs. On the other hand, with the Load Balancer, DPOR reduces the number of explored transitions up to 9 times which leads up to 10 times shorter exploration time. For REsPoNse the difference is smaller, but still meaningful: over 4 times.

3.11 Coverage vs. Overhead Trade-Offs

Testing is inherently incomplete, walking a fine line between good coverage and low overhead. Here we discuss some limitations of our approach.

Concrete execution on the switch: In identifying the equivalence classes of packets, the algorithm in Figure 3.5 implicitly assumes that the packets reach the controller. However, depending on the rules already installed in the switch, some packets in a class may reach the controller while others may not. This leads to two limitations. First, if *no* packets in an equivalence class go to the controller, generating a representative packet from this class was unnecessary. This leads to some loss in efficiency. Second, if *some* (but not all) packets go to the controller, we may miss an opportunity to test a code path through the handler by inadvertently generating a packet that stays in the "fast path" through the switches. This causes some loss in both efficiency and coverage. We could overcome these limitations by extending symbolic execution to include our simplified switch model and performing "symbolic packet forwarding" across multiple switches. We chose not to pursue this approach because (*i*) symbolic execution of the flow-table code would lead to a path-explosion problem, (*ii*) including these variables would increase the overhead of the constraint solver, and (*iii*) rules that modify packet headers would further complicate the symbolic analysis.

Concrete global controller variables: In symbolically executing each event handler, NICE could miss complex dependencies *between* handler invocations. This is a byproduct of our decision to represent controller variables in a concrete form. In some cases, one call to a handler could update the variables in a way that affects the symbolic execution of a second call (to the same handler, or a different one). Symbolic execution of the second handler would start from the *concrete* global variables, and may miss an opportunity to recognize additional constraints on packet header fields. We could overcome this limitation by running symbolic execution across multiple handler invocations, at the expense of a significant explosion in the number of code paths. Or, we could revisit our decision to represent controller variables in a concrete form.

Infinite execution trees in symbolic execution: Despite its many advantages, symbolic execution can lead to infinite execution trees [49]. In our context, an infinite state space arises if each state has at least one input that modifies the controller state. This is an inherent limitation of symbolic execution, whether applied independently or in conjunction with model checking. To address this limitation, we explicitly bound the state space by limiting the size of the input (*e.g.*, a limit on the number of packets) and devise OpenFlow-specific search strategies that explore the system state space efficiently. These heuristics offer a tremendous improvement in efficiency, at the expense of some loss in coverage.

Finally, there are two main sources of coverage incompleteness: (*i*) heuristic-driven and bounded-depth model checking, and (*ii*) incomplete symbolic execution of the controller code. We showed in Section 3.10 that at least one heuristic was always able to detect each bug. We do not report the code coverage of the controller because symbolic execution applies to event handlers that are a subset of the actual application logic, making it is difficult to distinguish between this logic and the rest of the system. Second, there are many hidden branches (*e.g.*, in dictionaries) that are not visible with code coverage statistics.

3.12 Chapter Summary

In this chapter we introduced NICE, a tool focused on finding bugs in SDN controllers. NICE, in its essence, works by combining two systematic software verification techniques — model checking and symbolic execution — into a single system. This, together with domain-specific heuristics, allows NICE to automatically exercise the controller-network interaction across different event orderings while at the same time picking only events that have a potential to uncover new system states.

Our evaluation shows that NICE scales better than traditional model checkers. Perhaps more importantly, when we put three different SDN controller applications to the test, NICE managed to find a total of 13 bugs. This is an indication that NICE can be useful to developers for uncovering problems in their controllers.

Chapter 4

Monitoring SDN Switches With Monocle

In the previous Chapter we described a systematic method to check whether an SDN controller program reacts correctly to network events and whether it installs the correct network policies. However, we based our testing on one major assumption, namely, that the SDN switches themselves behave correctly. Unfortunately, there are a plenty of reasons why this might not be the case; the potential switch errors may range from firmware implementation bugs and wrong/ambiguous interpretation of the specification to random hardware failures such as bit flips in TCAM[1] memory. Our goal in this Chapter is, therefore, to develop a new tool (called Monocle) that monitors the switches and checks whether they behave correctly.

4.1 Data Plane Correspondence Problem

While the notion of SDN correctness and reliability typically revolves around the controller responsible for enforcing the network policy configured by the network operator, it is the SDN switches that ultimately move (and modify) the packets in the network. It is, therefore, important to ensure that SDN switches perform their tasks (*i.e.*, data plane forwarding) exactly as instructed by the controller. We refer to this problem as *data plane correspondence*.

Unfortunately, guaranteeing data plane correspondence is difficult or downright impossible "by construction" or by pre-deployment testing. This is a

[1] Ternary content-addressable memory. Switches use this memory to quickly match packets against multiple forwarding rules.

consequence of the possibility of various software and hardware failures ranging from transient inconsistencies (*e.g.*, switch reporting a rule was updated sooner than it happens in data plane [55]), through systematic problems (switches incorrectly implementing the OpenFlow specification, *e.g.*, ignoring the priority field [55]), to hardware failures (*e.g.*, soft errors such as bit flips, line cards not responding, *etc.*), and switch software bugs [94], neither of which can be reliably detected in the control plane only.

Because a-priori guaranteeing data plane correspondence is impossible, it has to be done during run-time, *i.e.*, by actively monitoring the switches. Moreover, as already mentioned, querying only the switch's control plane (*e.g.*, asking the switch about what it is doing) cannot fully guarantee the data plane correspondence. Instead, the only way to reliably verify forwarding correctness, the switch must be exercised in the data plane. However, the typical choice of data plane monitoring tools is limited – operators can use end-to-end tools (*e.g.*, ping, traceroute, ATPG [94], *etc.*), or periodically collect switch forwarding statistics. Unfortunately, these methods are insufficient — ping/traceroute and other similar tools do not determine what packet header values can test for data plane correspondence. And while ATPG provides a comprehensive end-to-end data plane monitoring and can quickly localize problems, it is designed to batch-generate probes for all network rules at the same time. As a consequence, ATPG requires substantial time (*e.g.*, minutes to hours [94], depending on coverage) to pre-compute its probes after each network change. This delay is too long for modern SDNs where the ever-increasing amount and rate of change demand a quick, dynamic monitoring tool.

4.2 Verifying Data Plane Correspondence with Monocle

To address limitations of existing tools, we built Monocle — a system that allows network operators to simplify their network troubleshooting by providing automatic data plane correspondence monitoring capable of tracking all network changes. Monocle transparently operates as a proxy between an SDN controller and network switches, verifying that the network view configured by the controller (for example using OpenFlow) corresponds to the actual hardware behavior.

To ensure data plane correspondence, Monocle relies on a switch failure model in which each rule on the switch is either installed in the data plane or it is missing from it.[2] To ensure that a rule is correctly functioning, Monocle injects a monitoring packet (also referred to as a *probe*) into the switch, and examines the switch behavior. Monocle monitors multiple network switches in parallel and continuously, *i.e.*, both during *reconfiguration* (while the data plane is undergoing change during rule installation) and in *steady-state*.

During reconfiguration, Monocle closely monitors the updated rule(s) and provides a service to the controller that informs it when the rule updates sent to the switch finish being installed in hardware. This information could be used by a controller to enforce consistent updates [68].

In steady-state, Monocle periodically checks all installed rules and reports rules that are misbehaving in the data plane. This localization of misbehaving rules can then be used to build a higher level troubleshooting tool. For example, link failures manifest themselves as multiple simultaneously failed rules.

A major challenge in building Monocle is to correctly generate *probe* packets. This is difficult for a number of reasons. First, probe generation needs to be quick and efficient – the monitoring tool needs to be capable of quickly reacting to network reconfigurations, especially if the controller acts on its output. Moreover, the problem is computationally intractable (NP-hard, see Appendix A.1). The reason for this level of hardness is because the monitoring packets need to match the installed rule while avoiding certain other rules present in the switch. This case routinely occurs with Access Control rules, for which the common action is to drop packets. Instead of dealing with this complex problem on our own, we encode the probe generation as a SAT problem and leverage the existing tools to efficiently find the solution.

Second, a big challenge is dealing with the multitude of rules: drop rules, multicasting, equal-cost multi-path routing (ECMP), *etc.*, that all have to be carefully taken into account and dealt with. We address this challenge by developing a systematic framework with which we formulate constraints that the probes need to satisfy. Indeed, a big part of this Chapter is focused on describing what constraints the probes should satisfy and how this changes with varying types of rules.

[2] This is a reasonable assumption already adopted by the previous work [94]. Section 4.4.7 discusses what happens if we relax this property.

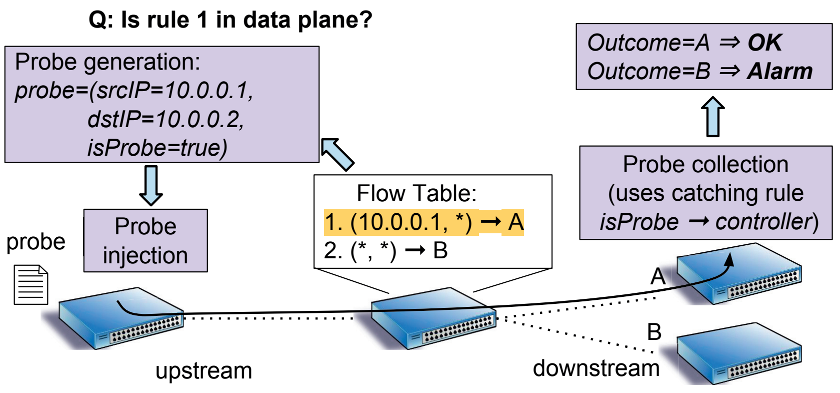

Q: Is rule 1 in data plane?
Probe generation:
probe=(srcIP=10.0.0.1, dstIP=10.0.0.2, isProbe=true)
Probe injection
probe
Flow Table:
1. (10.0.0.1, *) → A
2. (*, *) → B
Outcome=A ⇒ OK
Outcome=B ⇒ Alarm
Probe collection (uses catching rule isProbe → controller)
A
B
upstream
downstream

Figure 4.1: Overview of data-plane rule checking

4.3 Monocle Design

Monocle is positioned as a layer (proxy) between the OpenFlow controller and the network routers/switches. Such design allows Monocle to be transparent to the OpenFlow controller and thus it can be easily deployed in a network regardless of the controller itself. Being a proxy allows Monocle to intercept all rule modifications issued to switches and maintain the (expected) contents of flow tables in each switch. After determining the expected state of a switch, Monocle can compute packet headers that exercise the rules on that switch.

Figure 4.1 shows the core mechanism that the system uses to monitor a rule. Monocle uses data plane probing as the ultimate test for a rule's presence in the switch forwarding table. Probing involves instructing an "upstream" switch to inject a packet toward the switch that is being probed. The "downstream" switch has a special catching rule installed that forwards the probe packet back to Monocle. Upon the receipt of the correctly modified probe packet coming from the appropriate switch, Monocle can confirm that the tested rule behaves correctly in the data plane and can move to monitoring other rules. To ensure that probing does not affect the controller-generated network state, Monocle filters out all probes before they reach the controller.

Before Monocle starts monitoring the network, it computes and installs the catching rules. To reliably separate production and probing traffic, a catching rule needs to match on a particular value of a header field that is otherwise unused by rules in the network; moreover, this value cannot be used by the production traffic. In a network that requires monitoring rules at multiple switches, several such catching rules are needed. It is, therefore, important to minimize the number of extra catching rules that have to be installed. We

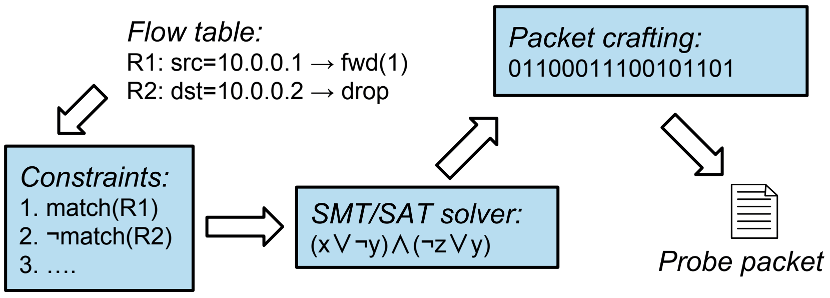


Figure 4.2: Steps involved in probe packet generation. Probes for different rules can be generated in parallel.

formulate this problem as a graph vertex coloring problem and solve it.

As Monocle relies on catching rules for its proper functioning, we need to verify that these rules work correctly. Fortunately, it is easy to check if the catching rules are installed by injecting packets that match them directly. Additionally, a broken catching rule appears as a correlated failure of all rules checked using this rule.

Figure 4.2 outlines how the probe packets are created. Monocle leverages its knowledge of the flow table at the switch to create a set of constraints that a probe packet should satisfy. Next, our system converts the constraints into a form that is understood by an off-the-shelf satisfiability (SMT/SAT) solver. Keeping constraint complexity low is important for the solving step. For this reason, Monocle formulates constraints over an abstract packet view [48, 94], structured as a collection of header fields. As the final step, Monocle needs to convert the SAT solution, represented in an abstract view, into a real probe packet. Monocle leverages an existing packet generation libraries to perform this task.

While we use OpenFlow 1.0 as a reference when describing and evaluating the system, its usefulness is not limited to this protocol. Presented techniques are more general and apply to other types of matches and actions (*e.g.*, multiple tables, action groups, ECMP).

4.4 Steady-State Monitoring

During steady-state monitoring, Monocle tests whether the control plane view of the switch forwarding state (constructed by observing proxied controller commands) corresponds to the data plane forwarding behavior. To ascertain

Hit	$Matches(probe, R_{probed}) \quad \wedge$ $(\forall R \in Rules : R.priority > R_{probed}.priority \Rightarrow \neg Matches(probe, R))$
Distinguish	Let $LowPrioRules := \{R \in Rules : \ R.priority < R_{probed}.priority\}$ and $IsHighestMatch(pkt, R, Rules) := Matches(pkt, R) \quad \wedge$ $(\forall R' \in Rules : R'.priority > R.priority \Rightarrow \neg Matches(pkt, R'))$ Then $\forall R \in LowPrioRules :$ $IsHighestMatch(probe, R, LowPrioRules) \Rightarrow$ $DiffOutcome(probe, R_{probed}, R)$
Collect	$Matches(probe, R_{catch})$

Table 4.1: Summary of constraints that probe packets need to satisfy when probing for rule R_{probed}.

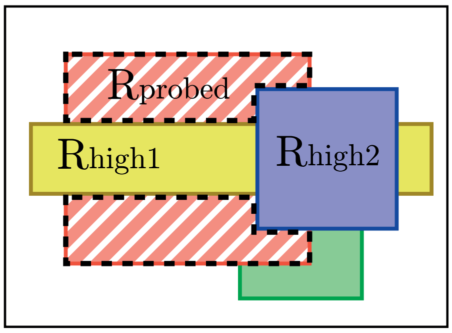
the correspondence, Monocle actively cycles through all installed rules and for each rule it (*i*) generates a data plane packet confirming the presence of the rule in data plane, (*ii*) injects this packet into the network, and (*iii*) moves on to testing the next rule as soon as the packet travels through the switch and it is successfully received by Monocle. In this section, we explain the creation of monitoring packets by gradually looking at increasingly complex forwarding rules.

4.4.1 Basic Unicast Rules

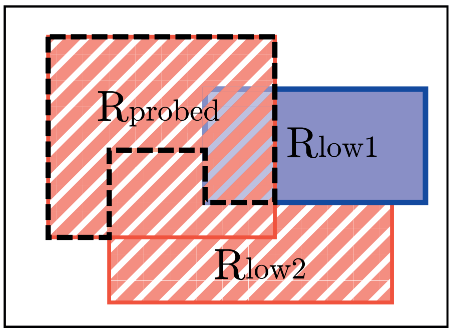
The presence of a given rule on a switch can be reliably determined if and only if there exists a packet that gets processed by a switch differently depending on whether the monitored rule is installed and working correctly. Therefore, the probe packet for monitoring the rule has to: (*i*) *hit* the given rule, (*ii*) *distinguish* the absence of the rule, and (*iii*) be *collected* by Monocle at the downstream switch. We formulate these conditions as formal constraints and summarize them in Table 4.1.

Hitting a rule: Only packets that *match* a given rule can be affected by this rule. Therefore, the header of any potential probe packet P must be matching the R_{probed} rule. Additionally, R_{probed} is seldom the only rule on the switch and different rules can overlap (*i.e.*, a packet can match multiple rules; switch resolves such a situation by taking rule priorities into account[3]). As such, for a probe P to be really *processed* according to R_{probed}, P cannot match any rule with a priority higher than the priority of R_{probed}. This is illustrated in Figure 4.3a.

[3] According to the OpenFlow specification, the behavior when overlapping rules have the same priority is undefined. Therefore, we do not consider such a situation.



(a) Hit: A probe for the striped rule needs to avoid any higher-priority rule(s).



(b) Distinguish: A probe needs to distinguish the rule from lower-priority rule with the same outcome.

Figure 4.3: Illustration of probe generation intricacies. A valid probe must belong to a region(s) within the dashed outline(s).

Distinguishing the absence of a monitored rule: Even the rules with priority lower than the probed rule R_{probed} affect the probe generation (Figure 4.3b). For example, if the probe matches a low priority rule R_{low2} that forwards packets to the same port as R_{probed}, there is no way to determine if R_{probed} is installed or not. Thus the probe has to avoid any such rule. Again, there is an intricate difference between a packet matching a rule R and being processed by R. Notably, if we just prevent P from matching all lower-priority rules with the same outcome, we may fail to generate a probe despite the fact that a valid probe exists. Consider the following set of rules ordered from lowest to highest priority (and unrelated to Figure 4.3b):

- $R_{lowest} := match(srcIP{=}*, dstIP{=}*) \rightarrow fwd(1)$, *i.e.*, default forwarding rule
- $R_{lower} := match(srcIP{=}10.0.0.1, dstIP{=}*) \rightarrow fwd(2)$, *i.e.*, traffic engineering diverts some flows
- $R_{probed} := match(srcIP{=}10.0.0.1, dstIP{=}10.0.0.2) \rightarrow fwd(1)$, *i.e.*, override specific flow, *e.g.*, for low latency

If the constraint prevented P from matching R_{lowest} (the same output port as R_{probed}), we would be unable to find any probe that matches R_{probed}. However, there exists a valid probe $P := (srcIP{=}10.0.0.1, dstIP{=}10.0.0.2)$ as the behavior of the switch with and without R_{probed} is different (R_{lower} overrides R_{lowest} for such a probe).

The provided example demonstrates that special care should be taken to properly formulate the *Distinguish* constraint listed in Table 4.1: When R_{probed} is potentially missing from the data plane, probe P cannot distinguish R_{probed} from an arbitrary lower priority rule R_{LP} having the same outcome as R_{probed} if probe P matches both R_{probed} and R_{LP} and at the same time P does not match any rule with a priority higher than R_{LP} (except R_{probed} itself). To formalize this statement, we define predicate $IsHighestMatch(P, R, OtherRules)$ that indicates whether packet P is processed according to rule R even if it matches some other rules on the switch. Using $IsHighestMatch$ we can assert that the probed rule R_{probed} must be distinguishable (*e.g.*, have a *different outcome*) from the rule which would process probe P if R_{probed} was not installed. For simplicity one may think about $DiffOutcome(P, Rule_1, Rule_2)$ as a test $Rule_1.outport \neq Rule_2.outport$, but we later expand this definition to accommodate rewrite and multicast.

Collecting probes: Monocle decides if a rule is present in the data plane based on what happens (referred to as probe *outcome*) to the probe packet. To gather this information but not affect the production traffic, we need to reserve a set of values of some header field exclusively for probes and ensure that a production traffic will not use these reserved values. We then pre-install a special "probe-catch" rule on each neighboring switch; this catching rule redirects probe packets to the controller and needs to have the highest priority among all rules. Naturally, as the last constraint, the probe P has to match the probe-catch rule R_{catch} of the expected next-hop switch.

4.4.2 Unicast Rules With Rewrites

On top of forwarding, certain rules in the network may rewrite portions of the header before outputting the packet. Accounting for header rewrites affects the feasibility of probe generation for certain rules. Consider a simple example containing two rules:

- $R_{low} := match(srcIP{=}*) \rightarrow fwd(1)$ and
- $R_{high} := match(srcIP{=}10.0.0.1) \rightarrow fwd(1)$.

It is impossible to create a probe for the high-priority rule R_{high} because it forwards packets to the same port as the underlying low-priority rule. However, if instead of R_{high} there was a different rule $R'_{high} := match(srcIP{=}10.0.0.1) \rightarrow rewrite(ToS \leftarrow voice), fwd(1)$ that marks certain traffic with a special type of service, we could distinguish it from R_{low} based on the rewriting action. The outcome of the switch processing a probe $P := (srcIP{=}10.0.0.1, ToS \neq voice)$ unambiguously determines if R'_{high} is installed.

In general, we can distinguish probes either based on ports they appear on, or by observing modifications done by the rewrites. Therefore, we define

$$DiffOutcome(P, R_1, R_2) := DiffPorts(R_1, R_2) \vee DiffRewrite(P, R_1, R_2).$$

However, checking if two rewrites are different requires more care than checking for different output ports. A strawman solution that checks if rewrite actions defined in two rules modify the same header fields to the same values does not work. Consider again rules R_{low} and R'_{high}. While the rewrites are structurally different (*e.g.*, $rewrite(None) \neq rewrite(ToS \leftarrow voice)$), they produce the same outcome if the probe packet happens to have $ToS = voice$. Therefore, to compare the outcome of rewrite actions, we need to take into account not only the rewrites themselves but also the header of the probe packet P and how it is transformed by the rules in question. Formally, we say that the rewrites of two rules are different for a given packet if and only if they rewrite differently at least one bit of the packet, *i.e.*, $DiffRewrite(P, R_1, R_2) := \exists i \in 1 \ldots headerlen : \Big(BitRewrite(P[i], R_1) \neq BitRewrite(P[i], R_2)\Big)$ where $BitRewrite(P[i], R)$ is either 0, 1, or $P[i]$ depending if rule R rewrites the bit to a fixed value or leaves it unchanged.

Finally, the rules in the network must not rewrite the header field reserved for probing. This assumption is required for two reasons: (*i*) if the probed rule rewrites the probe tag value, the downstream switch will be unable to distinguish and catch the probes; and additionally (*ii*) the headers of ordinary (non-probing) packets could be rewritten as well and afterward treated as probes; this would break the data plane forwarding.

4.4.3 Drop Rules

Drop rules can be easily distinguished from unicast rules based on output ports — the downstream switch either receives the probe or not. However, verifying that probes are dropped (a situation we call *negative probing*) brings in a risk of false positives: If the rule is not installed but monitoring packets get lost or delayed for other reasons (*e.g.*, overloaded link, packets damaged during transmission, etc.), Monocle is unable to determine the difference and assumes the rule itself drops the packets and thus is correctly installed in the data plane.

While false positives should be tolerable in most cases (*e.g.*, the production traffic is likely to share the same destiny as the probes and therefore the end-to-end invariant – traffic should be dropped – is maintained), we present a fully reliable method useful mainly for network update monitoring in Section 4.5.3.

Finally, since drop rules do not output any packets, header rewrites performed by them are meaningless and do not distinguish rules. As such we define $DiffRewrite(P, R_{drop}, R') := False$ to fit our theory.

4.4.4 Multicast and ECMP Rules

After discussing the rules that modify header fields and send packets to a single port or drop them, the only remaining rules are those that forward packets to several ports (*e.g.*, multicast/broadcast and ECMP). Such rules can be easily incorporated into our formal framework just by modifying the definition of $DiffOutcome$.

Both ECMP and multicast rules define a *forwarding set* of ports and send a packet to all ports in this set (multicast/broadcast) or a different port from this set at different times (ECMP). Moreover, note that drop and unicast rules are just special cases of multicast with zero and one element in their forwarding sets, respectively. Therefore, we only need to define $DiffOutcome$ for the following three combinations of rule types: (i) multicast + multicast, (ii) ECMP + ECMP, and (iii) multicast + ECMP. In all of these cases, we can distinguish rules again based on either their ports (forwarding sets) or based on their header rewrites, *e.g.*, $DiffOutcome(P, R_1, R_2) := DiffPorts(R_1, R_2) \vee DiffRewrite(P, R_1, R_2)$. We start by describing the case of distinguishing by different ports.

If both rules are multicast, a packet will appear on all ports from one of the forwarding sets. Therefore, if any port distinguishes these forwarding sets, we can use it to confirm a rule. As such, $DiffPorts(R_1, R_2) := (F_1 \neq F_2)$ where F_1 and F_2 denote forwarding sets of R_1 and R_2 respectively.

If both rules are ECMP, since each rule can send a packet to any port in its forwarding set, we can reliably distinguish them only if the forwarding sets do not intersect[4] (a probe appearing at a port in the intersection will not distinguish the rules as both rules can send a packet there). Thus, in this case $DiffPorts(R_1, R_2) := \big((F_1 \cap F_2) = \emptyset\big)$.

If only one of the rules (assume R_1) is multicast, we are sure that a packet will either appear on all ports in F_1, or on only one (unknown) port in F_2. We can simply capture the probe on any port that does not belong to F_2. Therefore, $DiffPorts(R_1, R_2) := \big((F_1 \setminus F_2) \neq \emptyset\big)$. (An equivalent definition is $F_1 \nsubseteq F_2$.)

Finally, there is an additional way to distinguish an ECMP rule from a multicast rule that is not unicast (*i.e.*,$|F_1| \neq 1$). We can differentiate them by counting received probes (an ECMP rule always sends a single probe). This way of counting the expected number of probes on the output is applicable in general and can extend the definitions of $DiffOutcome$, but since it is practically useful only in the presented scenario, we treat it as an exception rather than a regular constraint.

Now we analyze a situation when a rule may apply (possibly different) rewrite actions to the packets sent to different ports. We again need to consider the three types of combinations of rules R_1, R_2 with forwarding sets F_1, F_2 and adjust the definition of $DiffRewrite$ for each of them. When considering $DiffRewrite$, we take into account only actions that precede sending a packet to a port that belongs to $F_1 \cap F_2$ since if a packet appears at any other port, the location is sufficient to distinguish the rules. Additionally, we will need a new predicate: $DiffRewriteOnPort(P, R_1, R_2, port)$ which is true if the rule R_1 rewrites packet P differently than rule R_2 on port $port$. With the aforementioned observations, we consider the possible cases.

If both rules are multicast, there is going to be a probe packet at each output port in one of the forwarding sets. Thus, it is sufficient if there is a single port in the $F_1 \cap F_2$ on which the outputted packet is different depending which rule processed it. Therefore we have $DiffRewrite(P, R_1, R_2) := \exists port \in F_1 \cap F_2 : DiffRewriteOnPort(P, R_1, R_2, port)$ where F_1 and F_2 are forwarding sets of R_1 and R_2.

If both rules are ECMP, we need to be able to distinguish them regard-

[4] There is an alternative, probabilistic approach. Monocle could generate enough probes so that statistically at least one will differentiate two distinct forwarding sets. We decided not to use this method because it heavily relies on the switch hashing function and may fail if the switch hashes only on a small subset of header fields.

less of which output port each of them chooses. In particular, we need to be able to differentiate (by rewrite) on all common ports common to both rules. We therefore define $DiffRewrite(P, R_1, R_2) := \forall port \in F_1 \cap F_2 : DiffRewriteOnPort(P, R_1, R_2, port)$.

Finally, if only one of the rules (assume R_1) is multicast, we still do not know which port will be selected by R_2. Thus, for the same reason as in the previous case, we define $DiffRewrite(P, R_1, R_2) := \forall port \in F_1 \cap F_2 : DiffRewriteOnPort(P, R_1, R_2, port)$.

4.4.5 Chained Tables

Monocle as described so far assumes that each packet enters a switch on one of its ports, gets matched against a forwarding table once, and leaves the switch on one of the ports. In practice, switches may contain a pipeline of tables that each packet traverses (*e.g.*, chaining tables abstraction in OpenFlow 1.1). In such a case, Monocle would require the first table to have additional "fast forward" rules that redirect the probes to the desired tables. Similarly, each table needs to have a catching rule that intercepts the probe. Such design still requires only one probe per rule and in a sense treats chained tables in a single switch as a chain of switches, albeit with a more complicated probe injection mechanism.

4.4.6 Unmonitorable Rules

For some combinations of rules it is impossible to find a probe packet that satisfies all the aforementioned constraints, as can be seen in the following examples.

First, a rule cannot be monitored if it is completely hidden by higher-priority rules. For example, one cannot verify the presence of a backup rule if the primary rule is actively forwarding packets. Similarly, a rule is impossible to monitor if it overrides lower priority rules but it does not change the forwarding behavior, *e.g.*, a high-priority exact match rule cannot be distinguished from default forwarding if the output port is the same. If distinguishing such rules is absolutely necessary and the network operator is allowed to modify an unused header field of the production traffic[5], one may imagine Monocle modifying

[5] While allowing to modify header fields of a production traffic is not possible for general Internet connection providers, it might be feasible in a private datacenter setting.

rules by adding header rewrites to force different outcomes of the rules in question. We leave this as a potential future work.

Finally, it is impossible to monitor rules that send packets to the network edge as the probes would simply exit the network. While it is impossible to monitor such egress rules, many deployments (*e.g.*, typically in a datacenter) use hardware switches only in the network core and use software switches at the edge (*e.g.*, at the VM hypervisor). This lessens the importance of egress-monitoring — the software switches tend to update their data plane quickly and correctly acknowledge the update. Moreover, hardware failures are likely to manifest in the unavailability of the whole machine; this would be promptly diagnosed by existing server monitoring solutions.

4.4.7 Partial Failures

While Monocle is designed under the assumption that a rule is either correctly installed in the data plane or it is missing from it, there are types of failures which happen to lie in between these two cases.

In the first case, a rule might be matching correctly but have the rule *actions* damaged (*i.e.*, the switch is matching correct packets but performing a wrong action on them). This case is easy to spot for Monocle as the injected data plane probe follows an unexpected fate. Thus, as soon as Monocle receives the probe, either from a wrong neighboring switch or with unexpected header modifications, it can notify the operator about a rule failure. Similarly, if the probe was supposed to be forwarded but in reality it is dropped due to damaged actions, Monocle will observe that the probes are getting lost and alert the operator as well.

On the other hand, a rule might have a damaged *match*, *i.e.*, it matches a different part of header space than it is supposed to do. This type of a failure is very difficult to detect in practice due to the fact that some of the generated probes will behave correctly. Thus, it is necessary to use multiple probes to verify the rule — as an extreme example, if the rule matched wrongly only a single packet header, one would need to send all possible packet headers to be sure about the rule being correct. Because of this, neither Monocle nor any other system can guarantee to detect such failures. However, not all is lost, especially if the part of the match that is failing is big enough. Here, Monocle takes a pragmatic and probabilistic approach — each time we probe for a rule, we generate a *new* probe packet randomly (*e.g.*, by randomly seeding the SAT solver). This means that over time, Monocle verifies different parts of the rule match and the time it takes to detect a failure is inversely proportional to the size of the failure. In fact, if we define p to be the fraction of a match that is failing (*e.g.*, $p = 0.25$ if a quarter of the rule is failing), we expect to detect the problem on average in $1/p$ probing cycles with a heavy-tail distribution (the number of cycles is distributed according to Poisson distribution with $\lambda = 1/p$). Finally, if the partial match failures are more likely only in some region of the match (an excellent example is priority failures, as discussed in RuleScope [19]), Monocle could be guided to check these regions with higher probability. We however leave such implementation and evaluation for future work.

4.5 Update Monitoring

While monitoring networks in a steady state is important, network configuration is most fragile during policy updates. Monocle treats such periods with special caution and switches to a dynamic monitoring mode. In this mode, our system focuses only on rules that change, which allows it to generate probes quickly enough to confirm data plane updates almost in real time. Such knowledge is important for controllers trying to enforce consistent network updates [46], as the controller cannot update the "upstream" switch sooner than the "downstream" switch finished updating its data plane. In this section, we describe aspects of dynamic monitoring that differ from its static counterpart.

4.5.1 Rule Additions, Modifications, and Deletions

Generating probes for monitoring rule updates is similar to monitoring a static flow table. In particular, a probe for rule addition is constructed the same way as a steady-state probe assuming that the rule was already installed. The only difference is that for some switches, Monocle should tolerate transient inconsistencies (*e.g.*, the monitored rule missing from the data plane) and should not raise an alarm instantly. Instead, Monocle signals to the controller that the rule is safely in the data plane once the transient inconsistency disappears.

Similarly, rule deletion is treated as the opposite of installation. We look for a probe that satisfies the same conditions. However, rule deletion is successful only when the probe starts hitting actions of an underlying lower-priority rule. Next, rule modifications keep the match and priority unchanged. This means that the probe will always hit the original or the new version of the rule, regardless of other lower priority rules in the flow table. As such, we simply make a copy of the (expected) content of the flow table, adjust it by removing all lower-priority rules, and decrease the priority of the original rule. Afterward, we can use the standard probe generation technique on this altered version of the flow table to probe for the new rule version.

Finally, a single OpenFlow command can modify or delete multiple rules. Probing in such a case is similar to probing for concurrent modification of multiple overlapping rules at the same time. We describe the complications of concurrent probing in the next section, and leave reliable probe generation in the general case for future work. However, by knowing the content of the switch flow table, it is possible (at a performance cost) to translate a single command that changes many rules to a set of commands changing these rules one by one, and confirm them separately.

4.5.2 Monitoring Multiple Rules and Updates Simultaneously

In steady-state, generating a probe for a given rule does not affect other probes. Therefore, Monocle generates and then uses the probes for multiple rules in parallel. However, after catching the probe Monocle still needs to match it to the monitored rule. To solve this problem, we include in the probe packet payload, which cannot be touched by the switches, necessary metadata such as the rule under test and the expected result. This allows us to pinpoint which rule was supposed to be probed by the received probe packet. We use this technique in both steady-state and dynamic monitoring modes.

When monitoring simultaneous rule updates, Monocle must generate probes that work correctly for all already confirmed rules and at the same time for all subsets[6] of unconfirmed rules sent to the switch. This is required because the probe must work correctly even when the switch updates its data plane while other probes are still traveling through the network. As long as the unconfirmed updates are non-overlapping, the updates do not interfere with each other (see Section 4.6.4) and we can generate probes and monitor the

[6] According to the OpenFlow specification, a switch can reorder flow installation commands if they are not separated by a barrier message. Additionally, some switches do reorder even barrier-separated commands [55].

updates separately. Unfortunately, in a general case the problem is more challenging. Our current implementation handles unconfirmed overlapping rules by queuing rules that overlap with any yet unconfirmed rule until it is confirmed. We leave probe generation under several unconfirmed overlapping rules as a potential future work.

To illustrate why probe generation for multiple overlapping updates is challenging in a general case, consider the controller issuing three rules (in this order):

- low priority $R_1 := match(srcIP = 10.0.0.1, dstIP = *) \rightarrow fwd(1)$
- high priority $R_2 := match(srcIP = *, dstIP = 10.0.0.2) \rightarrow fwd(2)$
- middle priority $R_3 := match(srcIP = 10.0.0.0/24, dstIP = 10.0.0.0/24) \rightarrow drop$

After Monocle receives rule R_1, it has to send it to the switch, generate a valid probe (*e.g.*, $P_1 := (10.0.0.1, 10.0.0.2)$) and start injecting it. Assume the controller would then install rule R_2. On top of generating probe P_2, Monocle also needs to re-generate P_1 as it is no longer a valid probe for R_1 (if the switch installs R_2 before R_1, P_1 will always be forwarded by R_2, and therefore become unable to confirm R_1). Additionally, Monocle has to invalidate all in-flight probes P_1. And even if Monocle now receives rule R_3, probing for R_3 is impossible until rule R_1 is confirmed (if the default switch behavior is to drop). Similarly, until rule R_2 is confirmed, probe for R_3 needs to take into account two scenarios – either R_2 has been installed or not. The number of such combinations could rise exponentially, *e.g.*, 5 rules may require considering up to 2^5 outcomes.

4.5.3 Drop-Postponing

The final improvement is a way to reliably monitor installation of drop rules (rather than relying on negative probing or correctness of the control plane). The method is presented in Figure 4.4 and relies on modifying rules without affecting the end-to-end forwarding invariants. Specifically, instead of installing a drop rule on a switch, we can install a modified version of the rule which matches the same packets but instead of dropping, it rewrites the packet to a special header and forwards it to one of the switch's neighbors. The aforementioned neighboring switch must have in its flow table an appropriate, pre-installed rule which matches this special header and drops all matching traffic. Moreover, this drop rule must have a priority lower than the priority

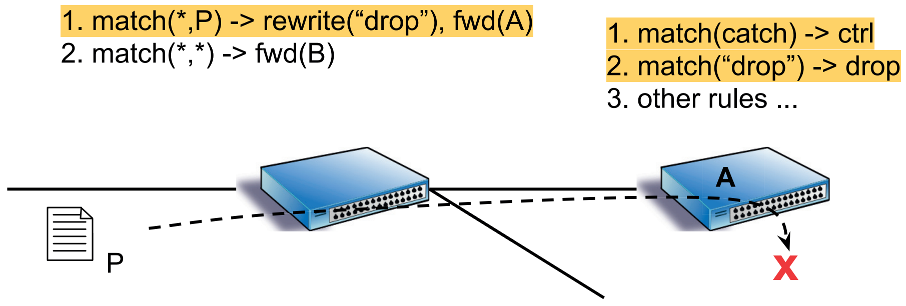


Figure 4.4: Illustration of how the drop-postponing method enables reliable probing for drop rules.

of probe-catching rule but sufficiently high that it dominates other rules. This way, all non-probe traffic is dropped one hop later while probe packets are still forwarded to Monocle (but with a modified header); this allows Monocle to realize when the drop rule is installed. Finally, after successfully acknowledging the "drop" rule, Monocle can update the rule to be a real drop rule as probing is no longer necessary; this change does not modify the end-to-end network behavior for production traffic.

While this method allows for the most precise monitoring of drop rule installation, it has the following drawbacks: First, it (temporarily) increases the utilization of a link to the neighboring switch because it forwards all to-be-dropped traffic there for some time. Second, it adds an additional rule modification to really drop packets after acknowledging the temporary "drop" rule.

4.6 Solving Constraints and Packet Crafting

As discussed in Section 4.4, probe generation involves creating a probe packet that satisfies a given set of constraints. Here we describe how to perform this task by leveraging the existing work on SMT/SAT solvers.

4.6.1 Abstracting Packets

While constraints from Table 4.1 are relatively simple, their complexity is hidden behind predicates such as $Matches(P, R)$ or $DiffRewrite(P, R_1, R_2)$. In particular, when dealing with real hardware, the implementation of packet matching is performing more than a simple per-field comparison. Instead, a switch needs to parse the respective header fields and validate them before proceeding further. For example, a switch may drop packets with a zero TTL or an invalid checksum even before they reach the flow table matching step. As such, it is important to generate only valid probe packets.

While the "wire-format" packet correctness can be achieved by enforcing packet validity constraints, doing so is undesirable as such constraints are needlessly complex (*e.g.*, checksums, variable field start positions depending on other fields such as VLAN encapsulation, *etc.*) to be efficiently solved by off-the-shelf solutions. Similarly to other work in this field (*e.g.*, [46,48,94]), we use an abstract view of the packet, *i.e.*, instead of representing a packet as a stream of bits with complex dependencies, we abstract out the dependencies and treat the packet as a series of (abstract) fields where each field corresponds to a well-defined protocol field (similarly to the definition of OpenFlow rules).

By introducing abstract fields, we can solve the probe generation problem without dealing with the packet wire-format details. As the final step, we need to "translate" the abstracted view into a real packet. As we show in the rest of this section, this process involves some technicalities. While previous work (*e.g.*, ATPG [94]) uses a similar translation, its authors do not go into the details of how to deal with this task.

4.6.2 Creating Raw Packets

The process of creating a raw probe packet given an abstracted header can be handled by the existing packet crafting libraries. The library can handle all relevant assembly steps (computing protocol headers, lengths, checksums, *etc.*). The only remaining task is providing consistent data to the library. In particular, there are two requirements on the abstract data that we provide to the library: (*i*) limited domains of some fields and (*ii*) conditionally present fields.

4.6.2.1 Limited Domain of Possible Field Values

Some (abstract) packet header fields cannot have arbitrary values because the packet would be deemed invalid by the switch (*e.g.*, DL_TYPE or NW_TOS fields in OpenFlow). Therefore, we need to make sure that our abstract probe contains only valid values of such fields. A basic solution is to add an additional "must be one of the following values" constraint on the abstract field. This solution is preferred for small domains (*e.g.*, input port). For domains that are big, we have an alternative solution: Assume that an (abstract) header field H can be only fully wildcarded or fully specified (*i.e.*, field H cannot have a partial wildcard). Moreover, assume that the domain of field H contains at least one spare value, *i.e.*, a valid value that is currently not used by any rule in the flow table. Then, we can run the probe generation step without any additional constraints and look at the resulting probe *probe*. If $probe[H]$ contains a valid value for the domain, we leave it as is. However, if $probe[H]$ contains an invalid value, we replace it by the spare value.

Lemma: Replacement of an invalid value of field H by a spare value does not affect the validity of *probe*.

Informal proof: Assume $probe[H]$ contains an invalid (*e.g.*, out-of-domain) value. As all rules in the flow table contain only valid values from the domain, it is clear that for each rule R in the flow table either $R.match[H] \neq probe[H]$ or $R.match[H] = *$. Setting $probe[H] := spare$ does not change inequalities to equalities and vice versa as we assume *spare* is a value not used by any rule. Thus, the substitution does not affect the $Matches(probe, R)$ test and therefore it preserves the validity of the solution with respect to the given constraints.

4.6.2.2 Some (Abstract) Packet Header Fields Are Included Only Conditionally

For example, one cannot include TCP source/destination port unless IP protocol is `0x06`. We use a term *conditionally-included* to denote a header field that is present in the header only when another field is present and has a particular value (*e.g.*, TCP source port is present only if the transport protocol is TCP). Similarly, a field that cannot be in the header because of the value of another field (*e.g.*, UDP source port if transport protocol is TCP) is called *conditionally-excluded.* While it is easy to remove all conditionally-excluded fields from the probe solution (*e.g.*, by ignoring their values), we need to make sure that the solution remains valid. A particular concern is whether for any rule R the value of $Matches(probe, R)$ stays the same. We show that the statement holds if rules are well-formed (*i.e.*, they respect conditionally-included fields as required by the OpenFlow specification $\geq$ 1.0.1).

Lemma: Eliminating all conditionally-excluded fields from any valid solution does not change the validity of $Matches(probe, R)$ for any well-formed rule R.

Informal proof: We will eliminate all conditionally-excluded fields one by one. For a contradiction, assume that there exists a conditionally-excluded field H and a rule R such that during the elimination of H the validity of $Matches(probe, R)$ changes. Clearly, field H cannot be wildcarded in $R.match$ otherwise the validity of $Matches(probe, R)$ would not change. As a consequence, $R.match$ includes field H and as rule R is well-formed (an assumption), $R.match$ has to also include an exact match for the parent field H' of H, *i.e.*, the field which determines conditional inclusion of H. We can now finalize the contradiction: If $probe[H'] \neq R.match[H']$, value of $Matches(probe, R)$ is $False$ regardless of the value of $probe[H]$ which contradicts the assumption that leaving out H changes the value of $Matches(probe, R)$. Further, if $probe[H'] = R.match[H']$, field H is conditionally-included which also contradicts the assumptions. Finally, parent field H' itself might be conditionally-excluded in *probe*; in such case, we perform the same reasoning leading to contradiction on its parent recursively.

4.6.3 Solving Constraints

Next, we show how to solve the constraints (listed in Table 4.1) that the probe packet needs to satisfy. As it turns out (see Appendix A.1), the problem of probe generation is NP-hard. Therefore, our goal is to reuse the existing work on solving NP-hard problems, in particular work on SAT/SMT solvers. While this requires some work (*e.g.*, eliminating for-all quantifiers in the *Hit* and *Distinguish* constraints), our constraint formulation is very convenient for conversion into SAT/SMT. In particular, we convert the *Hit* constraint to a simple conjunction of several $\neg Matches$ terms and the *Distinguish* constraint to a chain of if-then-else expressions, *i.e.*, we represent it as $If(m_1, d_1, If(m_2, d_2, If(m_3, d_3, ...)))$ where m_i and d_i are in the form of $Matches(P, R)$ and $DiffOutcome(probe, R_{probed}, R)$ for some rule R; this effectively mimics priority-matching of a switch's TCAM. The only remaining task to discuss is the conversion of the $Matches$ and $DiffOutcome$ predicates. $DiffOutcome$ consists of $DiffRewrite$ and $DiffPorts$. Basic set operations allow us to evaluate $DiffPorts$ to either $True$ or $False$ before encoding to SAT. Both $DiffRewrite$ and $Matches$ are similar in nature. Therefore, here we present only the encoding of $Matches$ in the context of the first three constraints. (We provide more details in Appendix A.2). For example, assume that all header fields are 2-bit wide (including IP source and destination). The goal is then to generate a probe packet for a low-priority rule $R_{low} := match(srcIP{=}1, dstIP{=}*) \rightarrow fwd(1)$ while using probe-catching rule $R_{catch} := match(VLAN{=}3)$ and assuming a high-priority rule $R_{high} := match(srcIP{=}1,\ dstIP{=}2) \rightarrow fwd(2)$. We represent probe packet as a sequence of 6 bits $p_1 p_2 \ldots p_6$ where bits 1-2 correspond to IP source, bits 3-4 to IP destination and bits 5-6 to VLAN. Then, the *Catch* and *Hit* constraints together are $Matches(P, R_{catch}) \wedge Matches(P, R_{low}) \wedge \neg Matches(P, R_{high})$ and the *Distinguish* constraint is simply $True$ as R_{low} is distinguishable from the default drop-all rule. This field-wise corresponds to

$$(p_{5\text{-}6} = 0b11) \wedge (p_{1\text{-}2} = 0b01) \wedge \neg\,(p_{1\text{-}2} = 0b01 \wedge p_{3\text{-}4} = 0b10)$$

where prefix $0b$ means the binary representation. This can be further expanded to $(p_5 \wedge p_6) \wedge (\neg p_1 \wedge p_2) \wedge (p_1 \vee \neg p_2 \vee \neg p_3 \vee p_4)$, which is a SAT instance.

4.6.4 Considering Only Overlapping Rules

Probe packet generation involves generating a long list of constraints that need to be satisfied. To increase solving speed, we strive to simplify the constraints based on the following observation:

Lemma: Let R be a rule that does not overlap with R_{probed}. Then the presence/absence of R in a switch flow table does not affect results of probe generation.

Proof: By definition, rules R_{probed} and R overlap if and only if there exists a packet x that matches both. The negation (*i.e.*, non-overlapping condition) is therefore $\forall x : \neg Matches(x, R_{probed}) \vee \neg Matches(x, R)$. As the expression holds for all packets, it must hold for probe P as well, *i.e.*, at least one of $\neg Matches(P, R_{probed})$ and $\neg Matches(P, R)$ holds. Combined with the assumption $Matches(P, R_{probed})$, it implies $\neg Matches(P, R)$. Therefore, parts of the *Hit* and *Distinguish* constraints related to rule R are trivially satisfied for any probe that matches R_{probed}. As a corollary, all rules that do not overlap with R_{probed} can be filtered out before building constraints. This is a powerful optimization, as typically rules only overlap with a handful of other rules.

4.7 Network-Wide Monitoring

Monocle design allows it to monitor and generate probes for each switch in the network separately. However, care must be taken to avoid interference among catching rules of different Monocle instances. In particular, each monitored switch S_i could be a downstream switch for multiple other switches, each of them requiring a different catching rule at S_i. At the same time, these catching rules should not match the probes used to monitor S_i otherwise the catching rules at S_i would intercept all probes instead of letting them match the monitored rule.

To overcome this problem, a single reserved value of the probe-catching header field is no longer sufficient. Instead, we propose two possible solutions that offer a trade-off between the number of header fields that need to be reserved for monitoring and the additional load imposed on the control channel.

The first solution is similar to [13]. The solution reserves a single header field H for monitoring and uses a set $Reserved$ of reserved values of this field, $Reserved = \{x_i : i \ is \ a \ switch\}$. The assumptions are similar to a single-switch probing: (i) production traffic never uses these reserved values of H, and (ii) no rule can rewrite field H.

Then, switch i installs $|Reserved| - 1$ catching rules; a rule matching on $match(H = x_j)$ for each $x_j \in Reserved \backslash \{x_i\}$. According to *Hit* and *Collect* constraints in Table 4.1, the value of field H in a probing packet has to be equal x_i — it cannot match any catching rule at the probed switch, but must be intercepted by a catching rule at the downstream switch.

Unfortunately, during monitoring of flow table updates, this method causes all probes (except for the ones dropped at the probed switch) to return to the controller even if they were forwarded by rules other than the probed one. This potentially increases control channel load (which is undesirable during the update) and forces Monocle to analyze more returned probes.

To address this problem, we propose a second solution at the cost of reserving two header fields H_1 and H_2 with reserved values $Reserved_1 = \{x_i : i \ is \ a \ switch\}$ and $Reserved_2 = \{y_i : i \ is \ a \ switch\}$, respectively.. Switch i preinstalls two types of rules used during probing:

1. a (high priority) probe-catch rule $R_{catch} := match(H_1 = *, H_2 = y_i) \rightarrow fwd(controller)$, and
2. (slightly lower priority) rules $R_{filter(j)} := match(H_1 = x_j, H_2 = *) \rightarrow drop$ for all $x_j \in Reserved_1 \backslash x_i$.

The generated probe will have $H_1 = x_{probed}, H_2 = y_{next}$ where *probed* and *next* are identifiers of the probed and the desired downstream switch, respectively. Such a probe is not affected by any catching rule on the probed switch but gets sent to the controller only if it reaches the correct downstream switch. The probe gets dropped by other neighbors of the probed switch so the Monocle receives it back only after the update happened in the data plane.[7]

Both presented solutions have a potential downside: they require as many reserved values of field(s) H and as many catching rules in each switch as there are switches in the network. However, what matters for the first method is that any two connected switches have different identifiers. Finding an assignment of labels to nodes of a graph such that no two connected nodes have the same label, and the total number of distinct labels is minimum is a well-known vertex coloring problem [62]. While finding an exact solution is NP-hard, doing so (as our evaluation in Section 4.9.3.2 suggests) is feasible for real-world topologies. Our study of publicly available network topologies [51, 79] shows that at most 9 distinct values are required in network topologies consisting of up to 11800 switches. Moreover, the time required is not crucial as it is a rare effort. Network topology changes, such as addition of new switches or links, trigger catching rule recomputation. Network failures do not require recomputation; the setup may simply no longer be optimal but it is still working.

The number of identifiers used by the second method can also be reduced in a similar fashion. In this case, however, it is not enough to ensure that two directly connected switches have distinct numbers assigned. Indeed, any switches that have a common neighbor must also have different identifiers otherwise this method loses the guarantee that the controller does not receive a probe until the probed rule is modified. As such, the second method works best on topologies which do not contain "central" switches with a high number of peers. From the algorithmic perspective, we can use the vertex coloring problem solver and just modify the underlying graph; we take the original graph and for each switch, we add fake edges between all pairs of its peers, essentially adding a clique to the graph.

4.8 Implementation

We design Monocle as a combination of C++ and Python proxies. Such proxy-based design enables chaining many proxies to simplify the system and provide various functionalities (*e.g.*, improving switch behavior by providing update acknowledgments). Moreover, it makes the system inherently scalable — each

[7] Unless the modification affects only rewrite actions, not the output port.

Monocle proxy is responsible for intercepting only a single switch-controller connection and can be run on a separate machine if needed.

Monocle mainly consists of two proxies — Multiplexer and Monitor. Multiplexer connects to Monitors of all monitored switches and is responsible for forwarding their PacketOut/In messages to/from the switch. Monitor is the main proxy and is responsible for tracking the switch flow table, generating the necessary probes, and sending update acknowledgments to the controller. To reduce latency on the critical path, Monitor forwards the FlowMod messages from the controller as soon as it receives them, and delegates the probe computation to one of its worker processes.

Monocle can use conventional SMT solvers for the probe generation. In particular, we implement conversion for Z3 [27] and STP [32] solvers. However, our measurements indicate that these solvers are not fast enough for our purposes (they are 3-5 times slower compared to our custom-built conversion to SAT when run on the experiments presented in Section 4.9.2). While we do not know the exact cause, it is likely that (*i*) Python version of bindings is slow, and (*ii*) these SMT solvers try too hard to reduce the problem size before passing it to SAT (*e.g.*, by using optimizations such as bit-blasting [32]). While such optimizations pay off well for large and complex SAT problems, they might be an overkill and a bottleneck for the probe generation task. Thus, we wrote our own, optimized, conversion to plain SAT (we use PicoSAT [7] as a SAT solver). The conversion and PicoSAT binding is written in Cython[8] to be on par with plain C code speed and we use the DIMACS format [25] to represent the CNF formulas as one-dimensional vectors of integers. We use such a single-dimensional representation instead of a more intuitive two-dimensional one (vector of vectors of integers, inner vectors representing disjunctions) because such representation resulted in poor performance – in particular, it necessitated *malloc()*-ing of too many small objects, which was the major bottleneck for the conversion.

Finally, since we do not have access to a real PICA8 switch for our evaluation[9], we create and use an additional proxy placed in front of an OpenVSwitch in one of the experiments. This proxy intercepts, delays and modifies the control plane communication to mimic the behavior (rule reordering and premature barrier responses) and update speeds of the PICA8 switch as described in [55].

[8] Do not confuse with CPython, the standard Python interpreter.
[9] We already returned a borrowed model used in [55].

4.9 Evaluation

In our evaluation, we explore the practicality of Monocle as well as its limits. We focus on and answer the following questions:

1. How quickly can Monocle detect failed rules and links? (*in a matter of seconds*)
2. Is steady-state monitoring useful? (*by using Monocle we detected issues with two hardware switches*)
3. How quick and effective is Monocle in helping controllers deal with transient discrepancies between control and data planes? (*it enables correct execution of consistent network updates [68] by providing accurate feedback on rule installation with only several milliseconds of delay*)
4. How long does Monocle take to generate probing packets? (*a few milliseconds*)
5. How big is the overhead in terms of additional rules and additional packets being sent/received? (*typically small*)
6. Does Monocle work with larger networks? (*it does and delays an installation of 2000 paths for only 350 milliseconds*)

4.9.1 Monocle Use Cases

We start by showcasing Monocle's capabilities in both steady-state and dynamic monitoring modes.

4.9.1.1 Detecting Rule and Link Failures in Steady-State

To demonstrate Monocle's failure detection abilities, we conduct an experiment where we monitor the data plane of an HP ProCurve 5406zl switch. We connect this switch with 4 links to 4 different OpenVSwitch instances mimicking a star topology with the hardware switch in the middle. We run OpenVSwitches and Monocle on a single 48-core machine based on the AMD Opteron 8431 Processor. To detect failures, we configure Monocle to monitor the switch with a conservative rate of 500 probes/second (Section 4.9.3), re-try sending a probe if there is no response for more than 50ms, and raise an alarm if a given probe is not received after 3 retries. In our first experiment, we install 1000 layer-3 forwarding rules on the HP switch, and let Monocle monitor the switch. Afterward, we "fail" a random rule (by removing it from the data plane without telling Monocle) and we measure the time it takes for Monocle

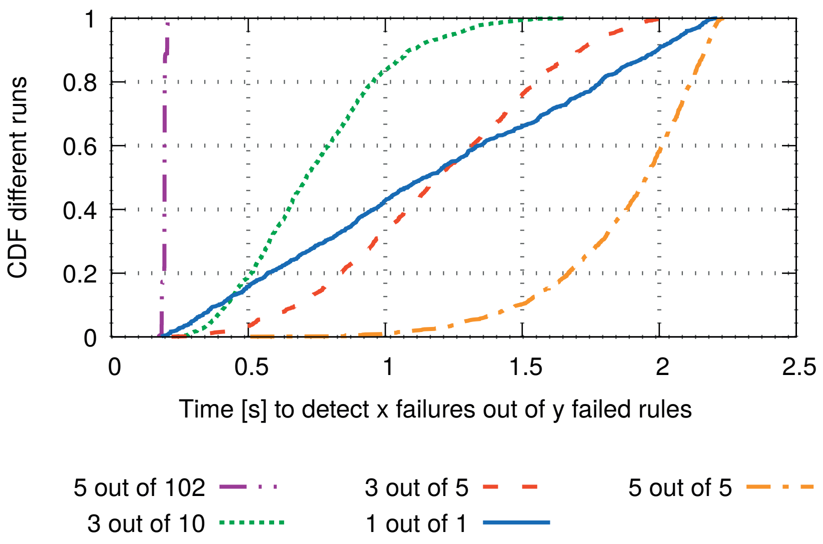


Figure 4.5: Time to detect a configured threshold of failures after a rule/link failure with a probing rate of 500 probes/second and 1000 rules in the switch flow table.

to detect the failure. We repeat the experiment 1000 times and plot the CDF of the resulting distribution. The results (solid line in Figure 4.5) suggest that, depending on where the failed rule happens to be with respect to the monitoring cycle (Monocle repeatedly goes through all the monitored rules), Monocle can detect the failure in 150 to 3000 ms.

Next, we study how fast a system built on top of Monocle could detect correlated failures, *e.g.*, failures that affect multiple rules simultaneously. In this experiment, we configure Monocle to raise an alarm only after detecting a given threshold (number) of individual rule failures. During the experiment, we fail multiple rules simultaneously, or, in one case, fail a whole link to which 102 of the installed rules forward to. We again repeat the experiment 1000 times and plot the CDF. As the violet (dash-dot-dot) line in Figure 4.5 shows, identifying big correlated failures (*e.g.*, a link failure) can be done quickly (on average in 200 ms, out of which 150 ms is the detection timeout). For a smaller number of failures and higher thresholds, Monocle requires more time as it is unlikely that many (or, in the extreme case, all) of the failed rules would be covered early on in the monitored cycle.

We finish the evaluation of Monocle's steady-state detection capabilities by experimenting with partial failures. With the exception of the switch itself (we did not manage to run the experiment on HP 5406zl, see Section 4.9.1.2),

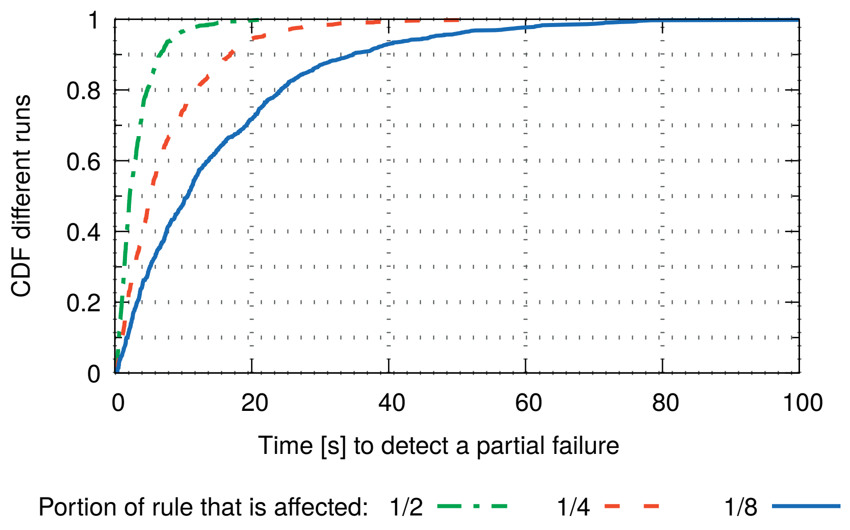


Figure 4.6: Time to detect a partial failure with a probing rate of 500 probes/second and 1000 rules in the switch flow table. Monocle might need multiple probing cycles to find a probe that falls within the affected part of the failed rule.

we use the same experimental setup (*e.g.*, 1000 layer-3 rules) as in the previous experiments. However, when we need to fail a rule, instead of removing it from the data plane, we replace it by several rules with a more specific match. In particular, we replace a single "/24" rule with $2^k - 1$ rules having netmask $/(24+k)$, effectively emulating a failure of a single $/(24+k)$ subregion. We run the experiment with $k = 1$, 2, and 3 (*i.e.*, failing 1/2, 1/4, and 1/8 of the rule). Note that in order to not introduce any false alarms due to the non-atomic rule updates on the switch, we first install the new rules (with slightly higher priority) and only then delete the original rule; only the deletion triggers the failure.

The results in Figure 4.6 show that partial rule failures take substantially longer to detect than full rule failures. In particular, detecting a partial failure takes on average 3.11 seconds for $k = 1$, 7.35 seconds for $k = 2$ and 15 seconds for $k = 3$ in our experiments. Unfortunately, as Figure 4.6 shows, the detection times are heavy-tailed (*e.g.*, it took slightly more than 100 seconds to detect the failure in the worst experimental run of the $k = 3$ experiment). This is no surprise as partial failures rely on a probabilistic detection approach and while not particularly likely, Monocle might get repeatedly unlucky with the probe selection.

4.9.1.2 Monocle Detects Previously Unknown Switch Problems

While running regular experiments to evaluate our system, Monocle surprised us by detecting unknown problems with the two switches we used.

A previously unknown firmware bug in the HP 5406zl switch: Surprisingly, when running the partial failure experiment (with $k = 1$) on our HP 5406zl switch, Monocle quickly and repeatedly detected a failure even after we restored the failed rule R to the original state.

By looking at the issue, we verified that the reported error is a real problem — the switch would drop packets destined to the restored rule R even while it reported in CLI that the rule is installed. Our suspicion is that the switch firmware contains a bug related to moving overlapping rules between software and hardware flow tables (when configured to use only hardware flow tables, the switch rejects installing rule $R_{partial}$ covering half of the rule R. Moreover, with the software flow table enabled, the switch reports that rule R is in hardware before we install rule $R_{partial}$, and gets moved to software later).

Rare non-deterministic rule failures on an undisclosed switch: When we run the basic steady-state experiments on a switch from a vendor that wanted to be anonymized, Monocle detected a non-deterministic failure with a rate of one in around 500-1500 experimental runs. Surprisingly, when we looked at the switch behavior, we observed that the problem affects rules which were not “failed” in the current run of the experiment but several runs ago. Importantly, the switch believed that the problematic rule was installed (according to the `ovs-ofctl dump-flows` command run on the switch). However, the packets destined to this rule were hitting an underlying drop-all rule (verified through the rule packet counters and an absence of packets on the output port).

Unfortunately, we did not manage to find out what exact conditions cause the problem to appear. Due to the rare occurrence, we managed to trigger this problem only a dozen times and the problem disappeared once we modified a few unrelated rules on the switch. Our best guess is that the problem happens because of TCAM memory corruption and the issue disappears as soon as the switch is forced to update the affected TCAM entry when the switch flow table is modified. This has been confirmed with the vendor, which stated that the problem was due to direct IO memory access to the TCAM and that they fixed it in a newer version of the chip.

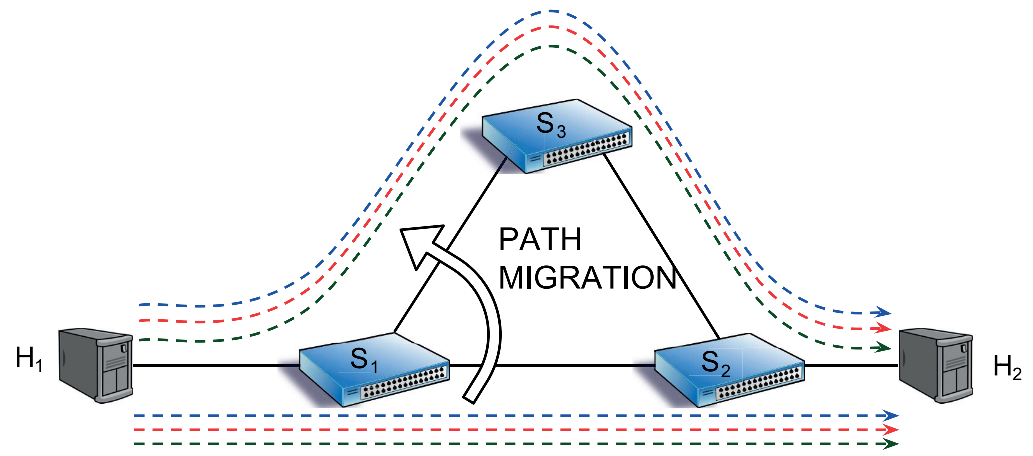
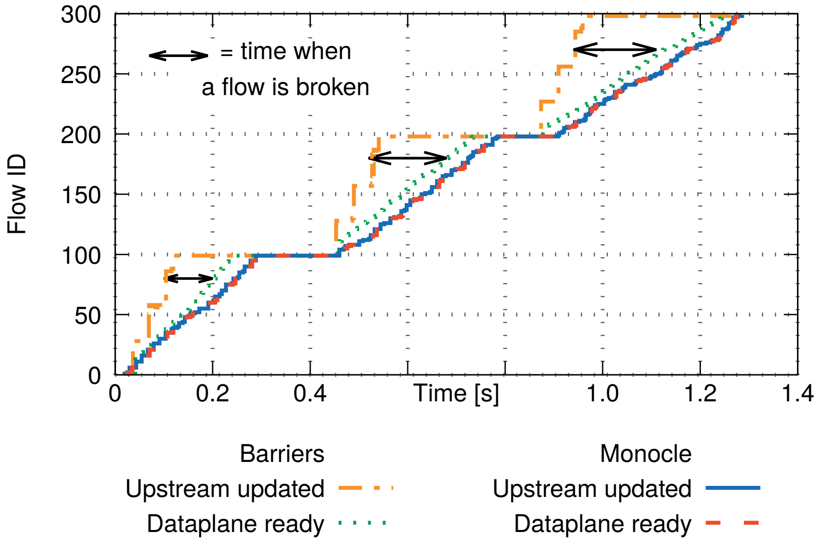


Figure 4.7: Experimental setup for testing transient inconsistencies during a path migration. The upstream switch S_1 needs to be updated only after switch S_3 installed forwarding rules in the data plane.

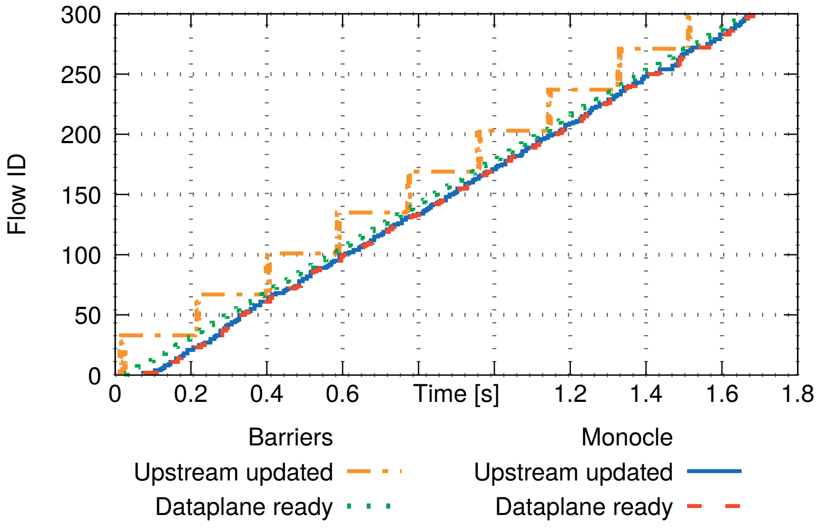
4.9.1.3 Helping Controller Deal With Transient Inconsistencies

Some OpenFlow switches prematurely acknowledge rule installation [54, 55]. As Monocle closely monitors flow table updates, it can help the controller to determine the actual time when the rules are active in the data plane. This, in turn, allows the controller to perform network updates without any transient inconsistencies. We demonstrate this by using Monocle in a scenario involving an end-to-end network update.

We configure a testbed (Figure 4.7) consisting of three switches S_1, S_2 and S_3 connected in a triangle, and two end hosts – H_1 connected to S_1, and H_2 connected to S_2. Switch S_3 is the monitored switch exhibiting transient inconsistencies between control and data planes. Initially, we install 300 paths that are forwarding packets belonging to 300 IP flows from H_1 to H_2 through switches S_1 and S_2. We send traffic that belongs to these flows at a rate of 300 packets/s per flow. Then, we start a consistent network update [68] of these 300 paths, with the goal of rerouting traffic to follow the path S_1-S_3-S_2. For each flow, we install a forwarding rule at S_3 and when it is confirmed, we modify the corresponding rule at S_1. We repeat the experiments using two different switches in the role of a probed switch (S_3): HP ProCurve 5406zl, and an OpenVSwitch with a proxy that modifies its behavior to mimic the Pica8 switch described in [55]. We always use OpenVSwitch as S_1 and S_2.



(a) HP 5406zl



(b) PICA8 emulation

Figure 4.8: Time when flows move to an alternate path in an end-to-end experiment. For both switches, Monocle prevents packet drops by ensuring that the controller continues the consistent update only once the rules are provably in the data plane.

Because both HP 5406zl and Pica8 report rule installations (by replying to a subsequent barrier command) before the installation actually happens in the data plane, a rule at the upstream switch S_1 gets updated in the vanilla experiment too soon and traffic gets forwarded to a temporary black hole. Figures 4.8a and 4.8b show when the packets for a particular flow stop following the old path, and when they start following the new path. The gap between

Data set	avg [ms]	max [ms]	probes found
Campus	4.03	5.29	10642 / 10958
Stanford	1.48	3.85	2442 / 2755

Table 4.2: Time Monocle takes to generate a probe.

the two lines shows the periods when packets end in a black hole. In the experiment, a theoretically consistent network update led to 8297 and 4857 dropped packets at HP and Pica8 respectively. In contrast, Monocle ensures reliable rule installation acknowledgments so both lines are almost overlapping and there are no packet drops. The total update time is comparable to the elapsed time without Monocle.

4.9.2 Monocle Performance

Here, we evaluate Monocle's performance. First, we answer the question whether Monocle can generate probes fast enough to be usable in practice.

Having access to a dataset containing rules from an actual OpenFlow deployment is hard. We observe that rules in Access Control Lists (ACL) are those most similar to OpenFlow rules, since they match on various combinations of header fields. Hence we report the times Monocle takes to generate probes for the rules from two publicly available data sets with ACLs: Stanford backbone router "yoza" configuration [46] containing 2755 rules (referred to as "Stanford"), and ACL configurations from a large-scale campus network [80] with a total of 10958 ACL rules (referred to as "Campus"; we aggregate ACL rules from 300 routers into a single virtual flow table).

For each dataset we construct a full flow table and then ask Monocle to generate a probe for each rule. In Table 4.2 we report average and maximum per-rule probe generation time. On average, Monocle needs between 1.44 and 4.13 milliseconds to generate a probe on a single core of a 2.93-GHz Intel Xeon X5647. This time depends mostly on the number of rules, and not on the rule composition and header fields used for matching. This is the case because the SAT solver is very efficient and the most time-consuming part is to check for the rule overlaps and to send all constraints to the solver. Further, our solution can be easily parallelized both across the switches (separate proxy and probe generator for each switch) and across the rules on a particular switch (probe generation is independent in steady-state).

Finally, we also show the number of generated probes compared to the number of rules Monocle is able to find (for reasons why Monocle may fail to find a probe see Section 4.4.6). On both datasets our system was able to generate probes for the majority of rules.

4.9.3 Overhead

Next, we show that the overhead of sending and receiving probes is modest when using hundreds or even thousands of probes per second, depending on the switch. We also show that the catching rules occupy a small amount of TCAM space in the switches.

4.9.3.1 PacketIn and PacketOut Processing Overhead

While it is possible to inject and collect probes via data plane tunnels (*e.g.*, VXLANs) to and from the desired switch, the approach we implemented relies on the control channel. Therefore, it is essential to make sure that the switch's control plane can handle the additional load imposed by the probes without negatively affecting other functionality. We start by estimating what are the maximum PacketIn and PacketOut rates that switches can handle when otherwise idle.

To measure the maximum PacketIn rate, we install a rule forwarding all traffic to the controller, send traffic to the switch, and observe the message rate at the controller. To measure the maximum switch PacketOut rate, we issue 20000 PacketOut messages and record the time when the last injected packet arrived at the destination. We repeat both experiments 5 times and for each experiment we report the average values over the runs; in all cases the standard deviation is lower than 3%.

The observed throughputs are 5531 PacketIn/s and 7006 PacketOut/s on an older HP ProCurve 5406zl switch, 401 PacketIn/s and 850 PacketOut/s on a modern, production grade, Dell S4810 switch, and 1105 PacketIn/s and 9128 PacketOut/s on Dell 8132F with experimental OpenFlow support. If the packet arrival rate is higher than maximum PacketIn rate available at a given switch, the switches start dropping PacketIns. These values assume no other load on the switch.

In the next experiment, we measure the effect PacketIns have on the switch control plane performance. To do so, we perform an update while injecting

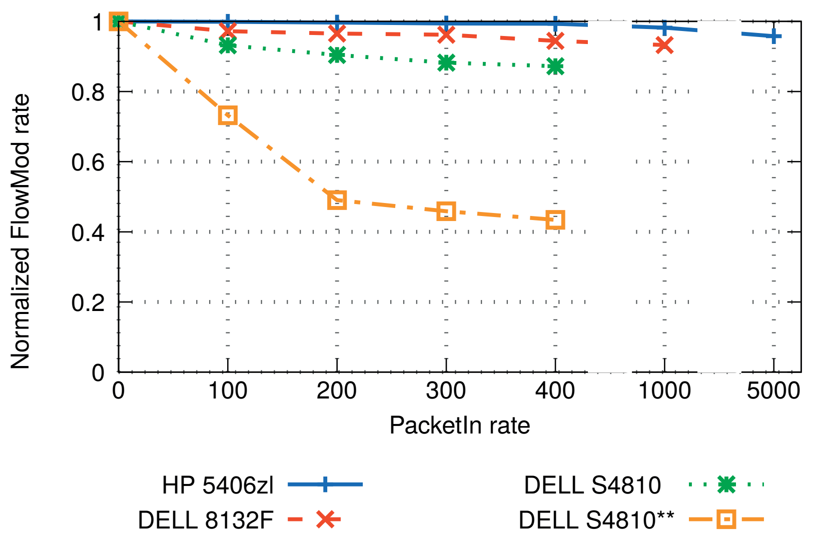


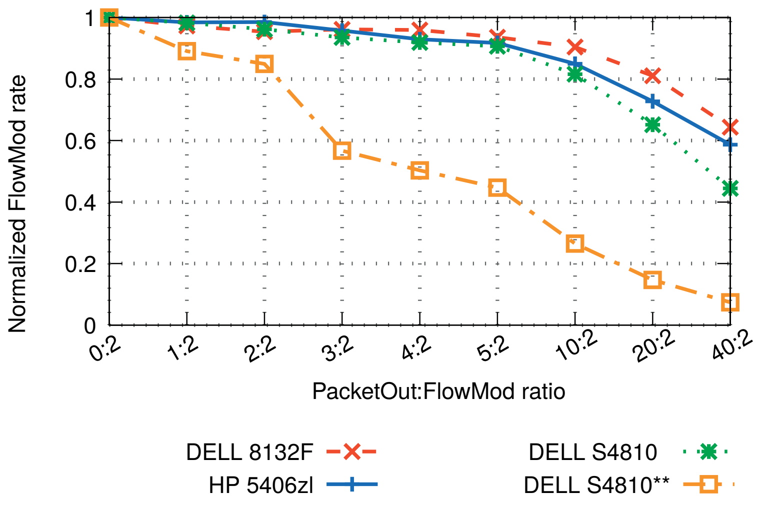
Figure 4.9: Impact of PacketIns on rule modification rate normalized to the rate with no PacketIns. Except for Dell S4810 with all rules having equal priority, PacketIns have a negligible impact on switches.

data plane packets at a fixed rate of r packets/s causing r PacketIn messages/s and observe how they affect the rule update rate. Figure 4.9 shows that all switches apart from one exceptional case are almost unaffected by the additional load caused by PacketIn messages. Dell S4810 with all rules having the same priority (marked with ** in Figure 4.9) is more easily affected by PacketIns because its baseline rule modification rate is much higher in such a configuration.

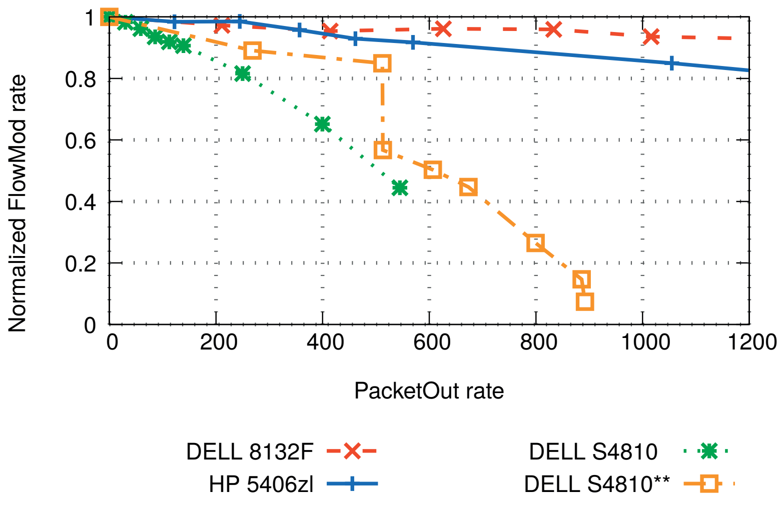
Next, we measure the overhead of PacketOut messages on the performance of flow table updates. We emulate in-progress network updates by mixing PacketOut and FlowMod messages using the $k : 2$ ratio (to keep the total number of rules stable, the 2 FlowMod messages are: delete an existing rule and add a new one). We vary k and observe how it affects the flow modification rate.

The results presented in Figure 4.10a show that the performance of all switches is only marginally affected by the additional PacketOut messages as long as these messages are not too frequent. Apart from one exceptional case, the measured switches maintain 85% of their original performance even if each flow modification command is accompanied by up to five PacketOut messages. Again, Dell S4810 performance drops when the baseline modification rate is high (all rules have the same priority, marked with **).

We also reuse the same measurement to estimate performance curves for



(a) Monocle can send 1 to 5 probes per each FlowMod without heavily impacting switch performance.



(b) Switches can handle hundreds to thousands of PacketOuts per second.

Figure 4.10: Impact of PacketOut messages on the rule modification rate normalized to a baseline rate for each switch. Note that Figure 4.10b is based on the data from Figure 4.10a instead of a separate measurement of the switch performance.

a varying PacketOut rate. This can be done by connecting scatter points $(x, y) = (FmodRate * ratio, FmodRate)$ of different ratios for each switch as plotted in Figure 4.10b. Switches can handle, depending on the model, hundreds to thousands of PacketOuts per second without excessively impacting the rule installation speed.

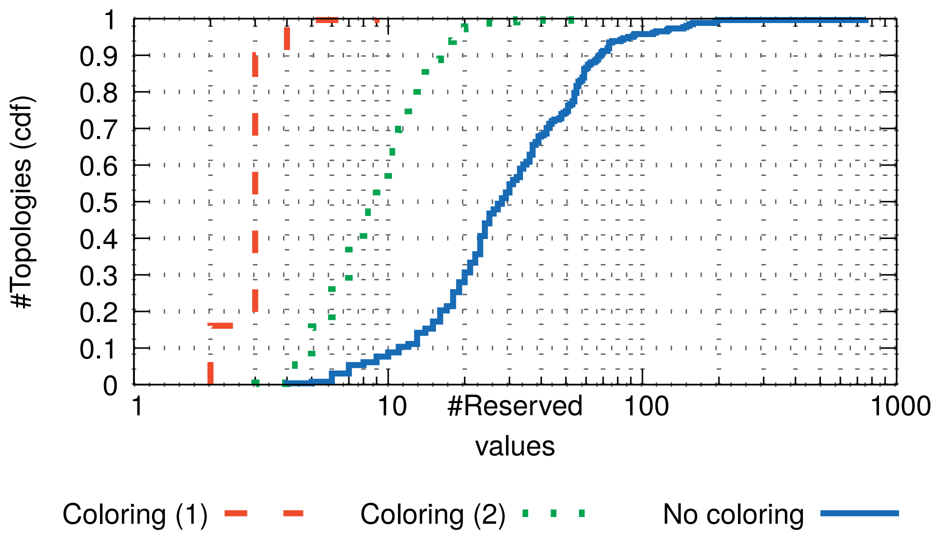


Figure 4.11: Number of reserved values in the probing field (also equal to a number of catching rules) for topologies from Topology Zoo [51]. Coloring 1 and 2 correspond to vertex coloring optimization for catching rules with 1 and 2 reserved fields, respectively.

4.9.3.2 Number of Catching Rules Required

Recall that our approach for multi-switch monitoring requires multiple probe-catching rules, and these effectively introduce rule overhead. To quantify this overhead, we compute the number of catching rules required for monitoring the network topologies from the Internet Topology Zoo [51] and Rocketfuel [79] datasets. To assign probe-catching rules to different switches, we use an optimal vertex coloring solution computed using an integer linear program formulation; solving takes only a couple of minutes to compute the results for all 261+10 topologies. We start by counting the number of topologies from Topology Zoo that require at least a given number of reserved values of the probe-catching field(s)[10] in the basic version where each switch has a distinct ID, as well as using vertex coloring optimizations for both of the previously explained strategies. Figure 4.11 presents a couple of interesting observations. First, both vertex coloring optimizations significantly decrease the number of the required values. Moreover, the strategy using just a single reserved field works with a very low number of IDs in practice — up to 9 values are sufficient for networks as big as 754 switches. The final, somewhat unexpected, conclusion is another tradeoff introduced by the technique with two reserved

[10]Which is the same as the number of probe-catching rules that must be installed on a switch, see Section 4.7

fields. Since the number of IDs it requires is at least as large as the largest node-degree in the network, the number is sometimes high (the maximum is 59).

Rocketfuel topologies confirm these observations — for networks of up to 11800 switches, the technique with a single reserved field requires at most 8 values while the second technique needs to use up to 258 values (note that we use greedy coloring heuristic for the second technique as our ILP formulation runs out-of-memory on our machine). Taking these observations into account, the most practical solution is the one that requires a single reserved field for probing.

4.9.4 Larger Networks

Finally, we show that Monocle can work in larger networks without prohibitive overheads. We do not have access to a large network, therefore, we set up an experiment that consists of a FatTree network built of 20 OpenVSwitches. As before, we add a proxy emulating Pica8 behavior to each of these switches. Further, each ToR switch has a single emulated host underneath, running a hypervisor switch that implements reliable rule update acknowledgments (also implemented as a proxy on top of OpenVSwitch). For comparison, we construct the same FatTree, but consisting of 28 (ideal) switches with reliable acknowledgments. We ignore the data-plane traffic to avoid overloading the 48-core machine we use for the experiment. Monocle is realized as a chain of three proxies per switch. As already mentioned, the proxies are highly independent and the problem can be easily parallelized. Probe generation for each switch is done in two threads.

We carry out an experiment to show how Monocle copes with high load and what is its impact on update latency. In the experiment the controller performs an update installing 2000 random paths in the network, starting 40 new path updates (5-7 rule updates each) every 10 ms.[11] Installation of each path is done in two phases: (i) install all rules except for the ingress switch rule, and (ii) install the remaining rule.

Figure 4.12 shows that Monocle performs comparably to the network built with ideal switches. Even though the probes have to compete for the control plane bandwidth with rule modifications, the entire update takes only 350 ms longer.

[11]Sending all rules at once would cause head-of-line blocking effects on the update. [66]

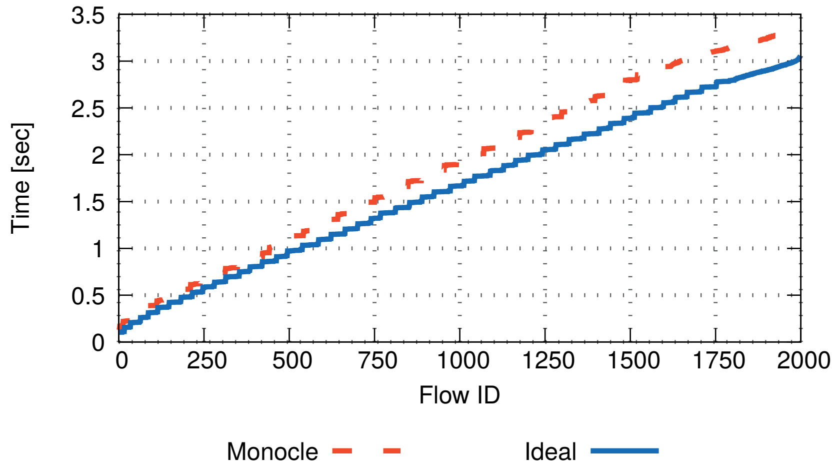


Figure 4.12: Batched update in a large network. Monocle provides rule modification throughput comparable to ideal switches.

4.10 Chapter Summary

This chapter introduced Monocle — a system that uses data plane packets to monitor whether SDN switches behave as configured by the controller. The main insight to data plane monitoring is generating probe packets while accounting for the various types of rules (*e.g.*, unicast, multicast, ECMP, header rewriting, *etc.*) and rule overlaps. To do this systematically, we devoted a big part of the chapter to build up the theory behind the constraints that the probe packets need to satisfy to reliably determine if the rule is working properly in the data plane. Monocle then converts the given constraints into a SAT/SMT instance that is solved by existing solvers. Afterward, Monocle translates the SAT/SMT solution back into actual packets and injects them into the network to observe their fate.

Evaluation of Monocle shows its high practicality. Monocle is capable of generating probes for the majority of rules in the switch flow table, and it can detect rule failures within a few seconds from the occurrence. When Monocle focuses on a single rule, it is fast enough to actually help controllers determine the exact time when a rule modification lands in the data plane. Finally, Monocle uncovered two previously unknown problems with the switches we used for the evaluation which further attests to its usefulness.

Chapter 5

Optimizing SDN Updates With ESPRES

In Chapters 3 and 4 we discussed how to improve SDN correctness and reliability. This chapter moves on and discusses another important topic — performance. In order to fully unleash its potential, SDN needs to offer an unprecedented speed of reconfiguration. This is driven by the fact that as network operators transition to SDN infrastructure, network changes that took days of planning and execution are now automated by the controller. As a consequence, network operators will push for ever higher rate of change, for example by providing self-provisioning portals to the customers or maybe running high-frequency traffic engineering to optimize the network load.

In this chapter, we examine the problem of installing bulk network updates. These updates, which may arise from traffic engineering, VM migration or failure recovery, are touching many different flows in the network. It is, therefore, essential to optimize the update in a way that minimizes the individual completion times, reduces the disruption time, or reduces the switch rule overhead.

Towards the goal of optimizing network updates, we introduce a runtime mechanism named ESPRES. ESPRES is based on the observation that a large update typically consists of a set of independent sub-updates, and hence sub-updates can be installed in parallel, in any order. ESPRES carefully plans the installation of individual sub-updates and actively manages switch message queues, while striving to fully utilize message processing capacities of switches without overloading them. By doing so, ESPRES optimizes network updates for a variety of goals without increasing the total update time.

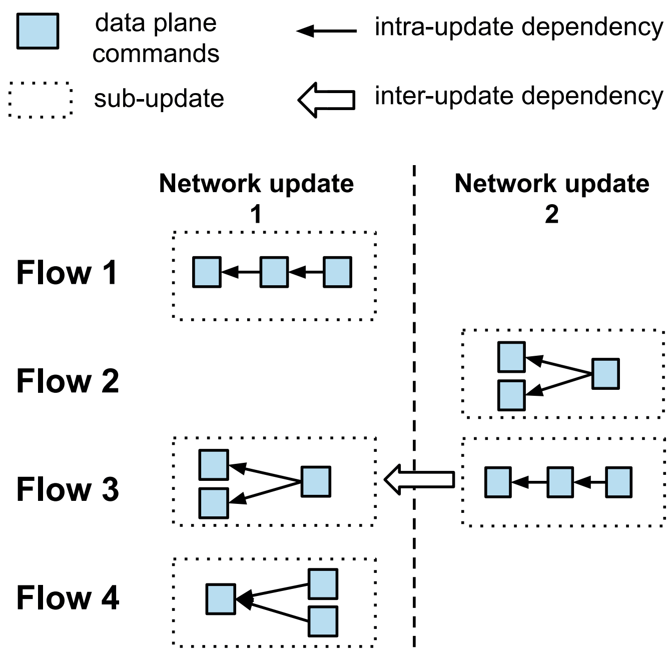


Figure 5.1: A key observation: *network updates* can be broken down into independent sub-updates.

5.1 Bulk Network Updates

This chapter is based on the key observation that an SDN update is typically induced by one or more high-level events, such as traffic engineering recomputations, VM migrations, and topology or policy changes. Typically, these events result in a batch-style update of forwarding state spanning multiple switches. Importantly, such a network update consists of a set of sub-updates that are independent of one another, that is, there are no rule installation dependencies between rule operations corresponding to these sub-updates (Figure 5.1). For example, each flow affected by traffic engineering can typically be updated independently of any other flow.[1]

Independence between sub-updates plays an important role because any combination of independent sub-updates can be applied in an arbitrary order, or even interleaved in parallel. Importantly, such interleaving would not introduce data plane inconsistencies (*e.g.*, would not cause a forwarding loop or a black hole, impose mutually exclusive forwarding actions, or violate other safety conditions [59]). Such independence is, therefore, a great source of flexibility for choosing an order in which rule operations and whole sub-updates

[1] Even with congestion free-induced limitations, there are solutions generating sets of independent sub-updates [58].

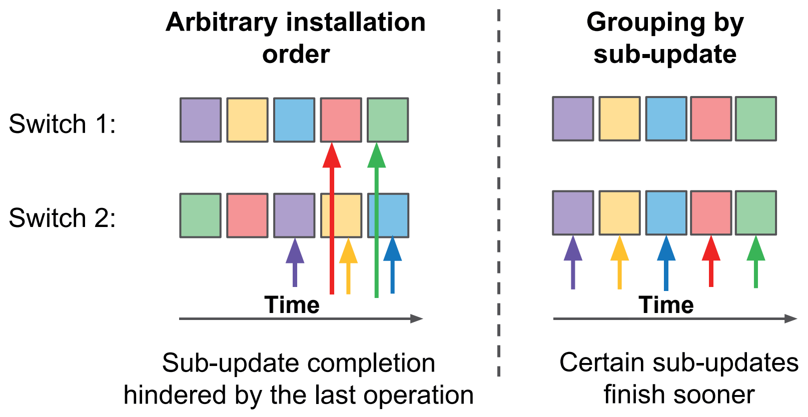

Arbitrary installation order
Grouping by sub-update
Switch 1:
Switch 2:
Time
Time
Sub-update completion hindered by the last operation
Certain sub-updates finish sooner

Figure 5.2: Grouping rules by sub-update helps to finish part of a *network update* sooner. A box corresponds to a data plane command executed on a switch and different colors correspond to different sub-updates.

are applied. By leveraging such flexibility, we can optimize the *network update* installation for a variety of goals. As an example, consider Figure 5.2 — by grouping individual switch commands by a sub-update and ordering them, parts of the *network update* could be finished sooner.

5.1.1 ESPRES Overview

Figure 5.3 illustrates the position of ESPRES in an SDN controller architecture introduced by Mahajan *et al.* [59]. The architecture consists of two layers: the *Network / Policy Database* and the *Update Plan Generator*. To a good extent, this architecture is a simplification from existing SDN control plane proposals such as Onix [52] and ONOS [12]. SDN applications interact with the controller through an interface that allows them to programmatically alter the network state via modifications to the Network / Policy Database. Changes to the database are propagated to the *Update Plan Generator*, a component that translates these changes into actual commands affecting switch rules (hereafter *rule operations*) as well as a *dependency graph* (Figure 5.1) that describes the intra-update dependencies between operations [59]. Additionally, inter-update dependencies between different *network updates* may exist. In a typical SDN stack, an OpenFlow driver or similar component would then send rule operations to switches based on the computed update plan. As shown in Figure 5.3, ESPRES operates below the *Update Plan Generator* and subsumes this latter component. ESPRES receives a stream of *network updates* and optimizes the

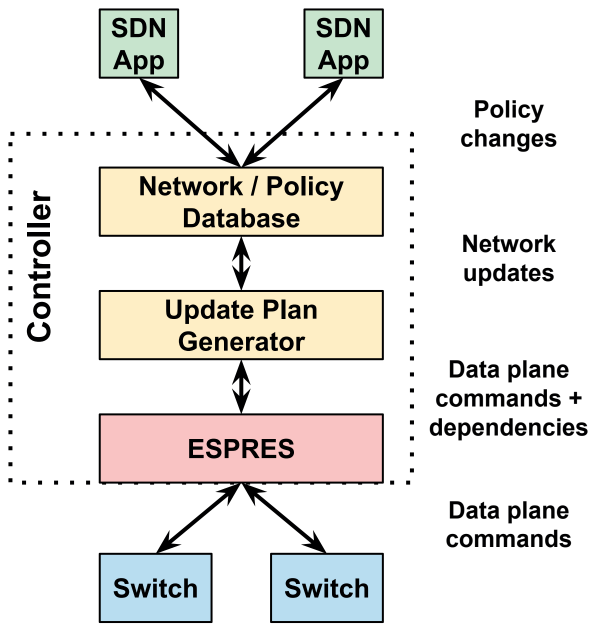


Figure 5.3: Position of ESPRES in an SDN architecture.

efficiency of rule installation by scheduling *rule operations*. Finally, ESPRES acknowledges all finished *network updates* back to the *Update Plan Generator*.

ESPRES introduces a *per-switch virtual message queue* that is realized as the combination of the message queue on the switch extended with a message queue maintained at the controller. Our key insight is that this queue extension enables ESPRES to continuously reassess and change the order in which messages should be sent to switches — virtual message queue is needed since once the messages are queued at the switches they can no longer be reordered with the current versions of OpenFlow or similar protocols. ESPRES' queue manager observes how long each switch takes to execute each message (data plane command), and carefully issues enough of them to keep the switch occupied without excessively queuing messages there.

At a high-level, ESPRES first groups all operations of an update into *sub-updates* that affect different logical traffic flows, and then schedules operations by ordering the *sub-updates*. ESPRES aims to execute as many *sub-updates* in parallel as possible, while offering the ability to optimize for different goals. For example, the goal of finishing flows sooner is accomplished by choosing shorter updates first. On the other hand, ESPRES can reduce rule overhead in the switches by preferring *sub-updates* that remove rules first.

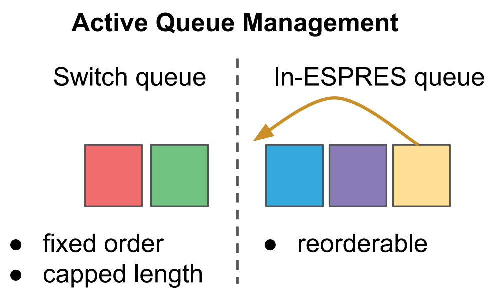


Figure 5.4: Active queue management helps with queue re-ordering. Each box represents a data plane command.

5.2 Managing Switch Command Queues

A key to exploiting all the available scheduling flexibility is maintaining good switch responsiveness by actively managing their command queues. That is, instead of sending all commands at once to a switch (and queuing them there with no possibility for future reordering or cancellation), ESPRES queues these commands at the controller and sends to the switch only a small subset of them (Figure 5.4). Naïvely sending all available commands to a switch fills up its queue, which delays installation of some rule dependencies. Instead, when the queue length is actively managed, ESPRES can decide which commands are to be sent next according to a particular scheduling discipline (*e.g.*, prefer rules from sub-updates that already started).

Because sending rule operations to a switch is not an instantaneous process, the *Queue Manager* needs to trade-off queue length (queuing adds an additional delay as well as limits the reordering possibilities) versus switch performance.

In particular, a very short queue length ensures low waiting latency but causes low rule modification throughput, whereas a very long queue provides full throughput at the expense of rule operations being stuck in the back of the queue for a long time. In our prototype, we use a simple heuristic for queue management: our algorithm limits the number of outstanding requests for each switch not to exceed a fixed threshold (5 in our experiments). Because OpenFlow lacks positive acknowledgments, we limit outstanding requests by using barriers and tracking the number of sent `BarrierRequest` messages and received `BarrierReply` messages.[2] We validate our decision for using just a simple threshold, as well as the trade-off between latency and performance in Section 5.4.

5.3 Scheduling Rule Operations

When the *Queue Manager* considers a particular switch command queue to be short, it notifies the *Scheduler*. Then, the *Scheduler* selects which rule operation should be next sent to that switch. A baseline solution is to disregard the active queue management and scheduling altogether and send rule operations as soon as all their dependencies are met. Such baseline solution has,however, a major drawback. Recall that each traffic flow starts following the new desired forwarding configuration only after all rules corresponding to the particular network flow are installed, *i.e.*, the sub-update for that flow completes. Therefore, if rule operations are ordered in an arbitrary (random) way, sub-update completion will be frequently hindered by the last rule operation. Instead, grouping rule operations by sub-update helps to finish parts of the *network update* sooner as illustrated in Figure 5.2.

5.3.1 ESPRES Schedulers

We base our schedulers on the observation that it is beneficial to install all rule operations of a given sub-update at roughly the same time. Thus, we design ESPRES schedulers as sub-update schedulers — they decide on the *preferred* order in which the sub-updates should be installed. Note that this is not a strict sequential ordering — in order to be work-conserving, ESPRES can send rule operations from any (even the last) sub-update as long as previous sub-updates do not have rule operations which are *ready* to be sent on a per-switch basis. A rule operation is ready if all of its dependencies are already installed.

Preferred order scheduler. The basic version of the scheduler always sends the first ready rule operation, according to the preferred order (we discuss how this order is derived in the next section). That is, each time the *Queue Manager* informs the *Preferred order scheduler* that (some) switches are ready,

[2] While barriers cannot be trusted to determine the data plane state of the switch, they can be trusted enough to identify which commands the switch processed in the control plane. As such, they can be used for our purpose of monitoring the switch command queue. Moreover, if the update contains dependencies between data plane commands, ESPRES could use Monocle to ensure that dependencies are ready in the data plane.

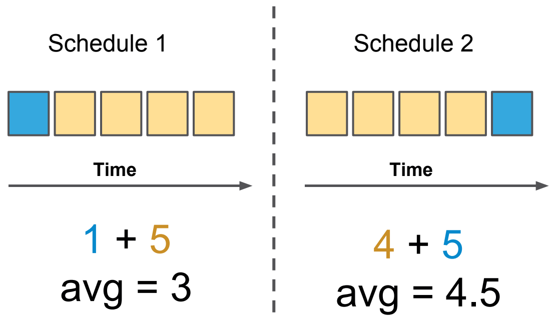


Figure 5.5: Ordering sub-updates by size can speed up average sub-update installation time.

the scheduler goes through the sub-updates in the current order, inspects each sub-update and sends out any ready rule operations. The scheduler ends iterating through sub-updates when the end of the sub-update list is reached or when there is no more switch queue space, whichever comes first.

Batch-ready scheduler. We observe that we can further improve the *Preferred order scheduler* performance by using the following heuristic. Instead of sending any ready rule operation across sub-updates as soon as a switch is available, we can synchronize ready operations within a sub-update and send them at the same time.[3] The *Batch-ready scheduler* thus iterates through the sub-update list similarly to *Preferred order scheduler* but sends sub-update ready rule operations as a batch and only when all corresponding switches are available. This provides an effect similar to gang scheduling [29], an operating systems scheduling concept that enhances the performance of multi-threaded programs by co-scheduling multiple threads of the same program at the same time.

5.3.2 Ordering Subupdates

The preferred ordering in which sub-updates should be installed plays an important role in the scheduling performance. For example, if the goal is to finish updating the majority of sub-updates in the network as soon as possible, we should order shorter sub-updates first similarly to the shortest job first scheduling in the context of operating systems (see Figure 5.5).

ESPRES supports a variety of preferred orderings. A baseline is an arbitrary fixed order of sub-updates (*e.g.*, sort by sub-update identifier). ESPRES can

[3] Assuming that the number of ready operations per switch is smaller than the limit on the outstanding number of requests.

also sort on the sub-update size, priority (if given by the controller), or a custom-defined order. Moreover, the preferred order is not necessarily fixed throughout the whole update.

For instance, to support the goal of minimizing mid-update rule overhead,[4] ESPRES uses an estimate of the current switch rule overhead to prefer updates that remove rules from the currently most overloaded switches. In this case, ESPRES periodically reorders all sub-updates according to a penalty function after potentially installing a sub-update. We define the penalty function of subupdate U as:

$$penalty(U) = \sum_{s \in switches} \mathrm{sqr}(\max(0,\ current\_rules_s + delta_s(U) - target_s))$$

where $current\_rules_s$ is the current number of rules installed at switch s, $delta_s(U)$ is the change in the number of rules on s if U was installed, and $target_s$ is the maximum of rules at s before the entire update starts (known by the controller) and the number of rules after it ends (can be computed by counting rule additions and deletions of the entire update), *i.e.*, $target_s = \max(initial_s,\ final_s + update\_delta_s)$

5.4 Evaluation

We implemented our ESPRES prototype as a Python program on top of the POX OpenFlow controller platform. We also developed a discrete event simulator to evaluate the system under more controlled conditions.

To validate the simulator, we run ESPRES in an emulated Mininet environment using the reference OpenFlow switches rate limited to 40 rule modifications/second. We observe that results of the simulation and emulation are comparable and therefore we use the simulator for the rest of our experiments. Further, unless specified otherwise, we set the rule modification rate at simulated switches to 1000 rules/second, which is more than the current generation of OpenFlow switches is capable of [41, 55]. Although fast switches are adversarial to ESPRES, we choose to use such value to confirm that ESPRES will be equally relevant in future deployments.

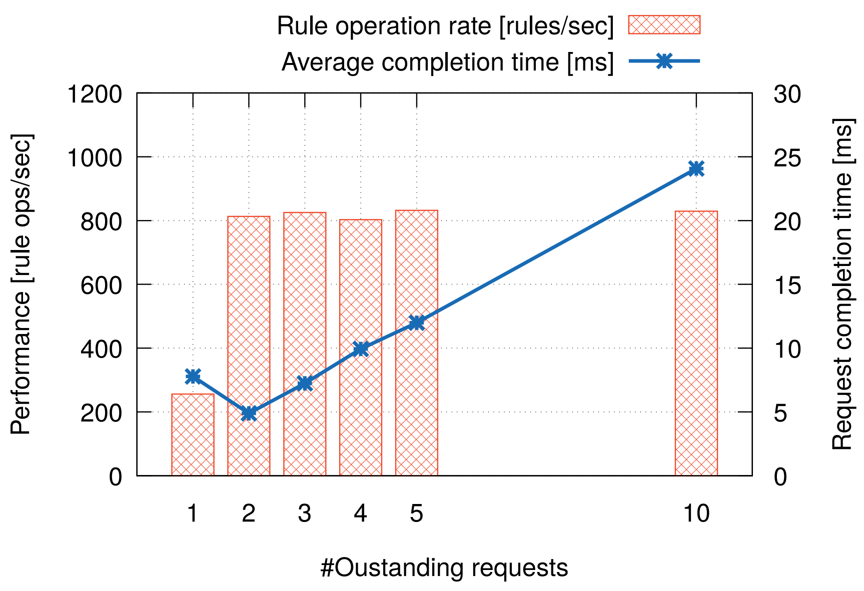


Figure 5.6: Performance of a limit-outstanding-requests queue management on a Pica8 switch. Rule modification throughput is constant for queue longer than 1.

5.4.1 Active Queue Management

First, we explore how short the switch queue can be without decreasing control plane performance. We design a benchmark where the controller maintains a fixed number of outstanding requests per switch and measures the switch performance. The controller pre-populates a switch flow table with 500 initial rules and then repeatedly removes a random rule and replaces it with a new one followed by a barrier request.

We run the benchmark on a Pica8 3290 switch (PicOS 2.0.4, OVS 1.10.0) and summarize the results in Figure 5.6. The main takeaway is that using a small number of outstanding requests (*e.g.*, two) does not decrease control plane performance. Running the benchmark on an HP ProCurve E5406zl, we observe that it requires a few more outstanding requests, and we thus set the number of outstanding requests in our experiments to 5 (to account for some variance in switch performance). Our measurements [55] confirm this choice for additional switch models.

[4] Controllers using the two-phase consistent update [68] require keeping both old and new rules at the same time.

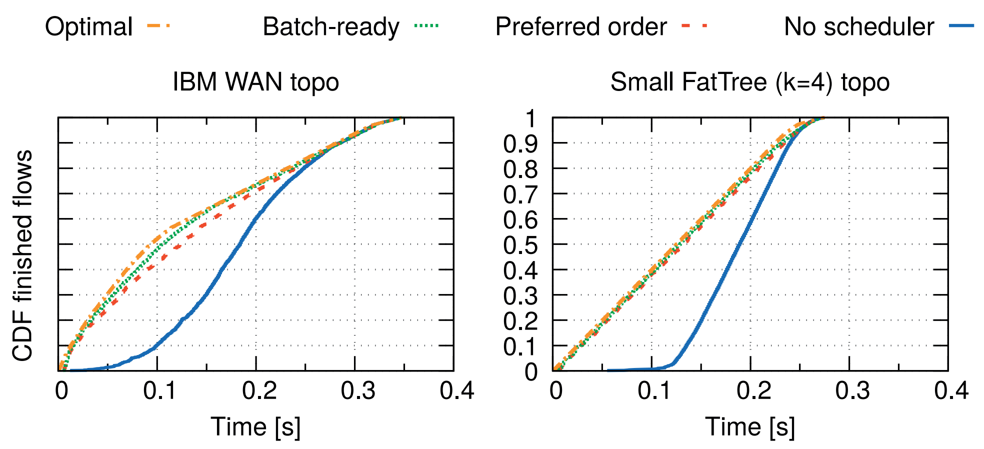


Figure 5.7: When installing a batch of new flows, scheduling reduces completion time for most of them.

5.4.2 Intra-Update Scheduling

To show the scheduling benefits of ESPRES, we evaluate it in three scenarios with different scheduling goals.

5.4.2.1 Improving Mean Time to Finish

We first focus on the mean time to finish a sub-update. This is an important metric when the controller is installing a batch of new flows (*e.g.*, spinning up a new VM) or repairing existing flows after a topology change. In this experiment, ESPRES coordinates installation of new sub-updates, each representing a flow on a randomly-selected shortest path between two edge switches chosen at random. Flows are installed in a per-packet consistent manner [68], *i.e.*, we use a two-phase update where the ingress rule is installed only after all other rules are in place. We run the experiment using various parameters, as discussed below. Each run was repeated 3 times and we observed comparable results across runs; therefore, due to space limit, we highlight a few major results.

Figure 5.7 shows the CDF of flow installation time for 1000 flows in an IBM topology [51] with 18 switches (all switches are edge switches) and a FatTree topology with 20 switches ($k = 4$, 8 ToR switches are edge switches). Table 5.1 summarizes results of experiments across a range of flows and switch performance parameters on the IBM topology.

Overall, the results show that significant benefits come even from our simple scheduling algorithms. The Batch-ready scheduler comes very close to an

Flows	Finish time improvement for the x-th percentile							
	relative		10 rules/s		100 rules/s		1000 rules/s	
	20th	50th	20th	50th	20th	50th	20th	50th
100	1.85x	1.46x	0.6 s	0.6 s	0.06 s	0.06 s	0.006 s	0.006 s
1000	4.06x	1.72x	10.1 s	7.7 s	1.01 s	0.77 s	0.10 s	0.08 s
5000	4.27x	1.80x	50.8 s	41.8 s	5.08 s	4.18 s	0.51 s	0.42 s

Table 5.1: Relative and absolute sub-update finish time improvement (at the 20th and the 50th percentile) of ESPRES's batch scheduler compared to no scheduler on IBM topology. ESPRES helps for all update sizes, but the improvement increases with the size. Note that switch rule installation speed has no effect on the relative improvement.

optimal schedule calculated by an integer linear program,[5] which has a high run-time overhead of 10 minutes. Further, our batch scheduler algorithm is up to 4.3 times better than not using a scheduler[6] for the 20th percentile of flows, up to 1.7 times better for the 50th percentile, and achieves equal total update time. Note that scheduling does not introduce performance benefits in the 9x-percentiles of flow installations, which correspond to the flows that are installed last. This is because these flows ultimately depend on which bottleneck switch is last to finish with rule installations. Nonetheless, scheduling does not worsen the installation time of these flows. Moreover, as shown in Table 5.1, increasing the switch rule installation speed does not affect the scheduling benefits. The overall update time decreases proportionally, but the benefits coming from ESPRES remain. Finally, the benefits of ESPRES are greater for bigger updates when the scheduler has a higher freedom to reorder sub-updates.

5.4.2.2 Lowering Mid-update Switch Rule Overhead

In these experiments, we assess rule overhead using a version of the scheduler that tries to minimize the mid-update switch rule overhead[7].

We design an experiment that resembles a traffic engineering recomputation

[5] For the ILP formulation we assume a priori known and constant switch performance.

[6] "No scheduler" is a scheduler that iterates over the flows in some predefined order and sends ready operations to the switch as soon as possible, ignoring any queue management. We try with different flow orderings but the ordering itself seems to have no major effect on the results.

[7] We calculate the per-switch overhead (in percent) as in [45]: $overhead = 100 * (worst/base - 1)$ where $worst$ is the maximum number of rules during the update and $base$ is the lower bound, $base = \max(\textit{pre-update rule count},\ \textit{post-update rule count})$.

Scheduler \ Sub-updates	352		1074	
per-switch overhead	max	avg	max	avg
No scheduler	55.4%	26.3%	62.3%	27.8%
Incremental consistent updates [45]	15.8%	15.0%	17.1%	8.7%
ESPRES	16.8%	5.7%	3.5%	1.3%

Table 5.2: Comparison of rule overhead when using two-phase update methodology [68] to achieve consistency.

in a FatTree topology with 20 switches. We first set up a number of flows over arbitrary shortest paths between pairs of ToR switches chosen at random. We consider both 500 and 1500 flows. Then, we compute new paths for all flows by randomly choosing a new shortest path for each flow. In about 25% of cases the same path for a flow is chosen and therefore the number of sub-updates is lower than the number of flows (*e.g.*, 352 and 1074 instead of 500 and 1500, respectively). Finally, we let ESPRES to update all flows in a consistent manner, *i.e.*, we provide dependencies between rule operations of the same flow as a three stage update where we first add new rules, then modify the "ingress" switch, *i.e.*, the switch where the new path diverges from the old one, and finally delete old rules.

Table 5.2 summarizes our results. We observe that a naïve update without a scheduler results in major maximum per-switch overhead (up to around 60%), while both incremental consistent updates [45] configured to use 4 rounds and ESPRES keep the overhead low (17% in the worst case). Further, ESPRES outperforms incremental consistent updates for larger updates because ESPRES reacts to current switch conditions at run-time and interleaves rule installations and rule deletions from different sub-updates to lower the overhead. We note that ESPRES is a best-effort service and does not guarantee low worst-case overhead like incremental consistent updates does. In the future, we plan to explore the benefits of running ESPRES on top of incremental consistent updates to bound the worst-case behavior while maintaining the benefits of run-time information.

5.4.2.3 Minimizing Subupdates Durations

We conclude the evaluation of scheduling goals with an experiment measuring the duration of individual sub-update installation times, *i.e.*, the time taken to install a sub-update from start to finish, as measured at the controller. We

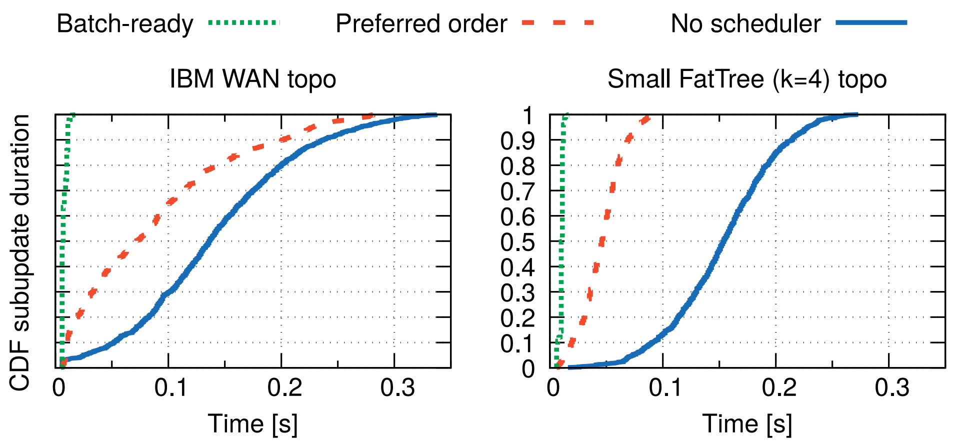


Figure 5.8: CDF of individual sub-update durations. Scheduling helps to minimize the possible disruption times.

use the same setup as for the mean time experiment. Results in Figure 5.8 suggest that our scheduler rapidly decreases the in-progress sub-update durations for most sub-updates, while finishing the update at the same time as the baseline case with no scheduling. This reduction can be attributed to grouping operations per sub-update and co-scheduling all operations at the same time.

5.4.3 ESPRES Sensitivity to Network Latency

To check how our greedy scheduler performs when working with delayed information, we vary the controller-switch round-trip-time (RTT) from 0 to 200 ms. To avoid switch underutilization, we adjust the number of outstanding requests to cover at least RTT-worth of rule operations (*e.g.*, 2, 105 and 205 outstanding requests for 0 ms, 100 ms, and 200 ms, respectively.)

Understandably, the update time increases with higher RTTs, as shown in Figure 5.9. However, we also observe that ESPRES performs worse than no scheduler for long RTTs (*e.g.*, the line marked with squares in the figure). An investigation revealed that our schedulers are too eager to finish the already-started sub-updates, which does not interact well with the long latencies during the final phase of an update as some sub-updates do not start until the very end and then they have to wait long for their dependencies. In particular, for each operation denote its *depth* as the maximum length of dependency chain from the operation to the update end (depth is 1,2 or 3 in our experiments).

Assuming the same RTT between the controller and all switches, if an update contains an operation at depth d, the update cannot finish sooner than in $d \times RTT$. To fix the scheduler, for each switch s and each depth d, we calculate

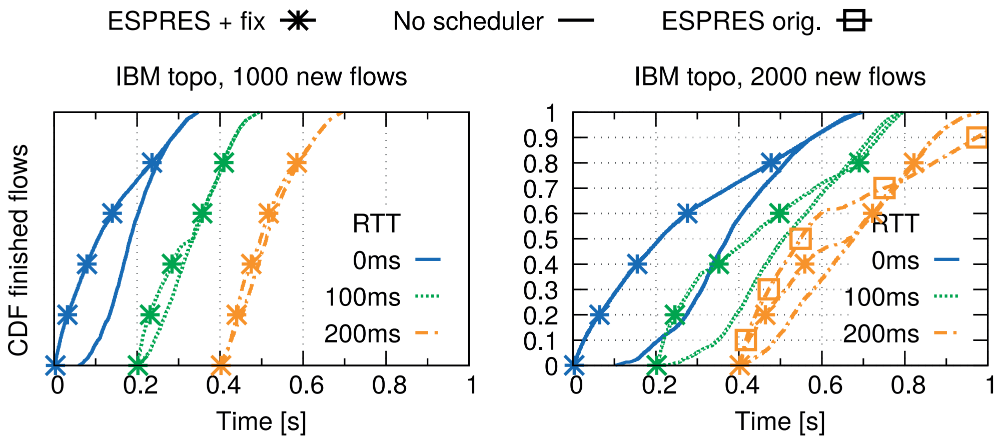


Figure 5.9: CDF of flow installation time as we vary the controller-switch RTT.

$pending(s, d)$ as the count of all operations at depth d still not sent. Then we calculate

$$T_{min}(s, d) = d \times RTT + pending(s, d)/rate(s)$$

and

$$T_{ETA} = \max_s \left(\sum_d pending(s, d)/rate(s) \right)$$

If $T_{min}(s, d) \times (1 + \varepsilon) > T_{ETA}$ for some depth d, there is a risk that an update would be delayed because of operations with depth d (we use $\varepsilon = 5\%$ as a safety margin). In this case, we force the scheduler to send to switch s only operations with depth $\geq d$, effectively starting new sub-updates instead of finishing already started ones.

After fixing the scheduler to start sub-updates earlier if some operations may wait too long because of dependencies, the results improve. There are two main conclusions coming from Figure 5.9. First, the fixed scheduler performs much better than baseline early during an update and stays no worse than the baseline near the end. Second, the time when the scheduler changes the strategy depends on the RTT, the switch rule modification rate and the update size. Thus, even if switches become faster, ESPRES will still be helpful in the future because the updates are likely to grow and the RTTs to decrease.

5.5 Chapter Summary

In this chapter we presented ESPRES, a runtime system that uses a moderate amount of available compute power to optimize bulk network updates. The main insight of ESPRES is that a bulk network update can be broken into many independent sub-updates, and the sub-updates can be re-ordered. ESPRES exploits the reordering potential and adapts to the switch command plane performance at runtime. By doing so it enables a vast majority of the flows to begin functioning correctly much quicker when compared to launching all commands on the switches at once as is typically the case today.

Our evaluation suggests that ESPRES is successful at lowering individual sub-update installation times for the majority of sub-updates, while not affecting the total update time. ESPRES is also capable of using its runtime knowledge to optimize mid-update overhead better than the existing static scheduling methods.

Chapter 6

Related Work

Over the years, the research on SDN produced hundreds of articles. Because it is impossible to list all the work here, in this chapter we give a just short overview of past research efforts closely related to this dissertation.

6.1 Model Checking

Model checking is an automatic and effective technique that has been successfully applied for finding bugs in concurrent programs (tools such as **VeriSoft** [34], **SPIN** [40] or **Java Path Finder** [85]), network protocol implementations (**CMC** [63], **KleeNet** [71]), distributed systems (**MaceMC** [50], **CrystallBall** [90], **MODIST** [91]), and file systems (**EXPLODE** [92]).

While some tools such as SPIN [40]) require writing models in a high-level modeling language (PROMELA in the case of SPIN), there has been an effort to use a real code as the base for model checking. VeriSoft [34] and Java PathFinder (JPF) [85] are among the first model checkers to directly operate on the source code instead of the model.

Unfortunately, even model checkers using real code instead of a model have a major drawback – model checking ordinarily requires a closed system, *i.e.*, a system (model) together with its environment. Moreover, the creation of such an environment is typically a manual process (*e.g.*, [71]). This is especially true for networks where modeling a universal real-world environment would require making an environment supporting many different protocols (and thus packet types).

NICE re-uses the idea of model checking—systematic state-space exploration—

and combines it with the idea of symbolic execution—exhaustive path coverage—to avoid pushing the burden of modeling the environment on the user. Also, NICE is the first to demonstrate the applicability of these techniques for testing the dynamic behavior of OpenFlow networks. Finally, NICE makes a contribution in managing state-space explosion for this specific domain.

6.2 Symbolic Execution

Symbolic execution exhaustively exercises all code paths through a program by utilizing an SMT solver to decide whether a given path is feasible. Symbolic execution has proven useful in automatically creating test cases that attempt to uncover implementation bugs in a given piece of code. Symbolic execution was pioneered by tools such as **DART** [35], **EXE** [22], **KLEE** [21], and **Cloud9** [20].

Khurshid *et al.* [49] present a generalization of symbolic execution that enables a model checker to perform symbolic execution. Their work effectively builds a symbolic execution engine for Java on top of an existing model-checker (JPF [85]): that is, it maps branching points in the code to possible transitions in the program state space. NICE shares this spirit of using symbolic variables to represent data from very large domains (*e.g.*, possible packets in the network). However, the difference is that NICE builds its model checker on top of symbolic execution. By doing so, NICE can use state matching as opposed to [49] since state matching is, in general, undecidable when states represent path conditions on unbounded data.

As a result, we (*i*) reduce state-space explosion due to feasible code paths because not all code is symbolically executed, and (*ii*) enable matching of concrete system states to further reduce the search of the state space.

6.3 Model Checking and Symbolic Execution in Computer Networks

Verifying behavior of computer networks is important for many reasons (*e.g.*, most importantly security and reliability). As model checking and symbolic execution have proven to be useful in general, researchers started to applying them to this domain as well. Here we list the most relevant work and its differences from NICE.

Kuai [61] is an SDN model checker that introduces a set of partial order reduction techniques to reduce the state space. However, unlike NICE, Kuai requires the application to be rewritten into a custom modeling language.

VeriCon [16] extends verification of SDN programs to check their correctness on all admissible topologies, and for all possible sequences of network events. These approaches need to manually port the controller application to a different programming language and do not use symbolic execution to reduce the space of input packets.

Kothari *et al.* [53] use symbolic execution and developer input to identify protocol manipulation attacks for network protocols. In contrast, NICE combines model checking with symbolic execution to identify relevant test inputs for injection into the model checker.

Bishop *et al.* [18] examine the problem of testing the specification of end host protocols. NICE tests the network itself, in a new domain of software defined networks. Moreover, NICE performs model checking of the implementation (*e.g.*, Python code that runs in the controller) directly, and thus avoids a lengthy step of deriving the specification.

Sethi *et al.* [76] present data- and network-state abstractions for model checking SDN controllers. Their approach extends verification to an arbitrary number of packets by considering only one concrete packet for the verification task. This reduces the state space but it also limits the invariants that can be checked to just per-packet safety properties.

SDNRacer [28] presents an alternative approach to NICE. Instead of exercising different event orderings in the network, SDNRacer simulates a network while collecting traces of network events including their happens-before relationship. Afterward, SDNRacer builds the “happens-before graph” and runs a concurrency analysis on it. NICE does not build a happens-before graph but its DPOR technique eliminates exploring different orderings of independent events.

6.4 Network Policy Verification

Anteater [60] uses static analysis of network devices’ forwarding information bases to uncover problems in the data plane. This approach takes user-specified network-wide invariants such as reachability and loop-free forwarding and maps them into instances of boolean satisfiability problem that are analyzed with a SAT solver.

SecGuru [43] is similar to Anteater. SecGuru uses a SAT solver to analyze network policies. Moreover, SecGuru can provide a semantic difference between two policies so that the operators can verify whether their changes have any unexpected consequences.

FlowChecker [14] applies symbolic model checking techniques on a manually-constructed network model based on binary decision diagrams to detect misconfigurations in OpenFlow forwarding tables.

Header Space Analysis (HSA) [46] is the first work to systematically examine the theory behind network forwarding. HSA models packets as points in a "header space" – a set of all viable headers, and models switches as functions which transform input packet into an output packet. More importantly, HSA allows for easy representation rules as wildcards can be represented as hypercubes in header space.

NetPlumber [47] is a follow-up work on HSA. NetPlumber allows for incremental recomputation of HSA if rules in the network change.

VeriFlow [48] is similar in nature to HSA. VeriFlow explicitly computes equivalence classes of packets that follow the same path in the network. In contrast, NICE computes equivalence classes of packets that trigger the same path in the network controller.

All these works as complementary to NICE — they deal with a snapshot of the network policy (*i.e.*, control plane configuration) rather than the dynamic interaction between the switches and the controller. NICE can, however, easily benefit from these tool by using their policy-checking abilities as safety invariants.

Similarly, we view all these works as orthogonal to Monocle because problems such as hardware failures, soft errors and switch implementation bugs can still manifest as an obscure and undetected data plane behavior that is not visible in the control plane snapshot. By systematically dissecting and solving the problem of probe packet generation, Monocle closes the gap and complements these other works. Monocle monitors the packet forwarding done at the hardware level and ensures that it corresponds to the control plane view that these tools verify.

6.5 Data Plane Monitoring

ATPG [94] is the first system to use data plane probes to automatically verify the switch forwarding behavior. However, there are some fundamental differences. To the best of our knowledge, ATPG (*i*) generates probes taking into the account only *Hit* and *Collect* constraints. It never checks whether the probes actually can *Distinguish* the rule from a lower priority one. (*ii*) More importantly, ATPG takes substantial time to generate the monitoring probes it needs. While this approach works well for static networks, it has serious limitations in highly dynamic SDN networks. In contrast, Monocle copes easily with this case, down to the level that it can observe the switch reconfiguring its data plane during a network update.

RuleScope [19] is a system similar to Monocle and ATPG. RuleScope's advantage over Monocle is its ability to systematically detect and troubleshoot rule priority inversions; Monocle could be adapted to look for such failures as well (by restricting the probe search only to overlaps with other rules, see section 4.4.7) but we leave such implementation and evaluation as future work.

On the other hand, similarly to ATPG, RuleScope does not consider *Distinguish*-ing rules from lower-priority ones based on rule actions and its design is more concerned with stable state monitoring/troubleshooting rather than quick dynamic inspection of the switch.

6.6 Network Debugging

SDN traceroute [13] focuses on a mechanism that allows tracing packets in an SDN network. Traceroute aims to observe the behavior for a particular packet. Our goal in Monocle is to observe switch behavior for a particular rule.

OFRewind [89] enables recording and replay of events for troubleshooting problems in production networks due to closed-source network devices. However, unlike NICE, it does not automate the testing of OpenFlow controller programs.

NDB [37] collects packet traces. Although both tools can uncover consistency problems (NICE in a model, NDB in production), they can only report the existence of a bug and do not fix it.

Heller *et al.* [38] describe a unified vision of SDN troubleshooting — they observe that an SDN network consists of multiple layers and categorize existing tools by the layers they are testing. This work thus shows deep insights about how exactly the different SDN troubleshooting tools are related.

6.7 Network Operating Systems and Platforms

NOX [36] is the first OpenFlow platform that was available. NOX is written in C++ but it also has Python API.

POX [8] started as an attempt to rewrite NOX into Python. Because of its ease of use and simple setup, POX became an attractive platform for research controllers despite its Python limitations (*e.g.*, slow speed and the absence of multithreading).

We built our three systems on top of these two platforms. Apart from NOX and POX, there are more controller platforms such as **Ryu** [10], **OpenDaylight** [4], and **ONOS** [2], *etc.*

6.8 SDN Abstractions and Programming Models

Providing abstractions is important for easying SDN development. Unfortunately, finding abstractions with the right balance of simplicity and flexibility is hard. The works in this short section show some very interesting points in the design space.

Frenetic [31] and **FlowLog** [64] are domain-specific languages for SDN. In contrast with NICE, which aims at finding programmer bugs, the higher-level abstractions in these works make it possible to eradicate certain classes of programming faults and to some extent enable controller verification. However, these languages have not yet seen wide adoption.

Maple [86] is an interesting compromise between higher-level abstractions and practicality. Maple provides an abstraction that the OpenFlow controller examines every packet at every switch in the network. In reality, Maple attempts to offload some of the work to the switches.

While these abstractions avoid some classes of bugs (*e.g.*, rules conflicting with each other or hiding visibility to the controller), we believe NICE would be still relevant in finding different classes of concurrency issues [67]. Moreover, we believe that these abstractions might simplify the task of network verification. It would be therefore interesting to see whether tools such as NICE could be modified to fully exploit the limits these abstractions impose on the network and controller behavior.

6.9 Switch Control Plane Behavior

Having a detailed knowledge about switch control plane behavior is important for SDN programmers because it can affect the design of SDN controllers. The most important questions that need to be answered are (*i*) the limits on the rate of change (affecting the practicality of big network updates), and (*ii*) any switch "quirks", *i.e.*, unexpected behaviors. Fortunately, multiple researchers already answered these questions and these results actually prompted us to work on both Monocle and ESPRES.

OFLOPS [70] is the first study of SDN switch performance. OFLOPS measures several OpenFlow-capable switches and shows that they are not very fast at updating their data plane configuration. Moreover, OFLOPS is the first work to notice the temporal discrepancy between the time when the switch reports a command to be done, versus when its data plane starts following it.

Huang *et al.* [41] measure switch performance (*e.g.*, rule installation rate) to build high-fidelity switch models.

Our own switch measurements in [55] confirm findings from these papers, namely that the switches (*i*) have unpredictable (and sometimes rather low) performance, and (*ii*) controller cannot trust barrier requests. As a response, because large updates can take substantial time to install, we build ESPRES to optimize such updates. Similarly, we built Monocle to closely monitor whether the switch data plane is doing what it should.

6.10 Consistency of Network Updates

There are many states a network can undergo while updating from one configuration to another. This is especially true when the update touches multiple switches, each of them updated at a different time. If not carefully controlled, these intermediate network states might lead to policy violations, induce network loops, or send traffic into temporary black holes.

Consistent updates. Reitblatt *et al.* [68] pioneered the notion of consistent updates, a class of network updates where each packet is processed according to either old or new configuration but never by a mix of them. This prevents transient inconsistencies such as loops, black holes or policy violations, and is in a stark contrast with traditional BGP updates where temporary inconsistencies can occur [65]. To prevent a mix of network policies, consistent updates use a variation of a two-phase commit where the new rules are first installed on all switches except ingress points and only then the ingress points are updated.

Incremental Consistent Updates. Consistent updates spawned a lot of interest from researchers and multiple follow-up works. As already stated, consistent updates use a two-phase commit to transition from the old configuration to the new one. Unfortunately, this requires having rules of both configurations at the same time, effectively reducing the switch TCAM size by two. This was improved by Katta *et al.* [45] who reduce switch rule overhead by splitting the update into several rounds. This, however, increases the overall update duration.

zUpdate [58] further improves the consistent update to take link capacities into the account – it performs congestion-free updates using a set of carefully computed steps.

These systems are important to ESPRES but at the same time somewhat orthogonal – the major difference between these systems and ESPRES is the fact that they are responsible for coming up with the update operations and their dependencies while ESPRES is responsible for taking these rules and installing them in the network. Moreover, both zUpdate and Incremental Consistent Updates use linear programming to come up with a plan optimizing for the worst-case behavior. ESPRES, on the other hand, is a best-effort system which cannot change already-existing (potentially suboptimal) update plans, it can only decrease transient problems while the network update is being installed. Thus, ESPRES can actually work on top of both Incremental Updates and zUpdate by trying to avoid the worst-case scenario if possible.

Finally, these systems rely on the assumption that switches report correctly when they installed rules in the data plane. Unfortunately, as the measurements [55, 70] suggest, this is not true for some switch models and thus these systems might lose consistency guarantees. Monocle can help these systems to monitor the switch data plane and proceed with the update only once the rules are indeed in the data plane. By doing so it restores the guarantees these systems provide.

6.11 Network Update Scheduling

Jive [57] measures the performance of OpenFlow switches according to predetermined patterns to derive switch capabilities; these capabilities could, in turn, be used to optimize network behavior. In contrast, ESPRES dynamically adapts to run-time switch performance while scheduling rule installation.

Mahajan and Wattenhofer [59] recently discuss consistent updates in SDN, and their *plan executor* subsystem is perhaps the closest in spirit to our work. However, ESPRES goes further in offering the initial design, implementation, and evaluation of a network-wide scheduler for rule installation.

Dionysus [44] is a system for fast, consistent update installation. Dionysus improves on the previous work by considering constraints such as network bandwidth and switch TCAM space on top of standard consistency guarantees. Similarly to ESPRES, Dionysus schedules updates at runtime depending on the progress. However, unlike Dionysus which strives to minimize the total length of the update, ESPRES observes that it is also worthwhile to optimize parts of the update. Ultimately, joining ESPRES and Dionysus would combine advantages of both systems.

Chapter 7

Conclusions and Future Work

It is likely that Software-Defined Networking is going to revolutionize the way network operators configure and manage their networks. However, it is still too early for SDN to be production-ready and the network operators do not want to give up the "five-nines" (99.999%) reliability of the old, traditional networks.

This thesis is a step towards bridging this gap — the underlying question we answer throughout the thesis is *"Is it possible to use the vast amounts of computing power available today to ease SDN programming and management?"* Fortunately, the answer to the question is *yes* and we illustrate it multiple times.

First, in Chapter 3 we describe how we can utilize compute power to improve SDN debugging. Our tool NICE thoroughly explores the interaction of an SDN controller with the rest of the network by systematically examining various event orderings. Doing so helps programmers uncover insidious bugs and race conditions that result in wrong network behavior.

Second, our tool Monocle described in Chapter 4 demonstrates that the extra CPU cycles can be put to the difficult task of overseeing SDN switch correctness. Monocle monitors the switches by carefully constructing data plane packets, which are then injected into the network and observed. By doing this, Monocle detects switch problems ranging from firmware bugs to random memory corruption.

Finally, Chapter 5 shows that even performance of SDN updates can be improved by taping into the computation power. Thanks to the careful scheduling and constant adjustments based on the current progress, our tool ESPRES is able to improve on certain aspects of SDN updates.

7.1 Future Work

There are many ways in which the work in this thesis could be improved. Here we list the natural follow-up work that would remove some of the current limitations, as well as more vision-like ideas that can spawn new research directions on their own.

NICE: As we mention in Section 3.11, NICE cannot model dependencies between two handler invocations. To lift this constraint, we could either run a series of handler calls each time we use the `discover_packet` transition, or use symbolic state. Only a careful evaluation would show which approach fares better in terms of scalability vs. bug-finding ability. Similarly, when computing packets to be injected, NICE currently does not consider rules installed on the switches. The potential solution here would be to either run symbolic execution also on the switches, or combine NICE with some specialized reachability analysis such as Header Space Analysis [46].

By being based on model checking and symbolic execution, NICE walks a tight line between the state-space explosion encountered by these techniques and its usefulness in terms of bug-finding abilities. A part of the future work would be pushing this line further. Specifically, could we find heuristics which work better? Can we increase the size of the network on which applications are tested without a big performance penalty? And more importantly, can we systematically reduce the state-space explosion by perhaps splitting the problem into several smaller subproblems?

Monocle: Monocle currently cannot cope with multiple overlapping rule updates at the same time and needs to serialize them (see Section 4.5.2). We envision that this problem could be solved by carefully reformulating the constraints and SAT/SMT conversion to accommodate for "maybe-installed" rules. The outstanding question is whether this can be encoded more efficiently than enumerating all 2^k possibilities where k is the number of maybe-installed rules.

Monocle, as it was designed, assumes that the only type of a failure on the switch is "a single rule missing from the data plane" (although Monocle can also probe for multiple non-overlapping failures as well as some partial failures described in Section 4.4.7). By carefully studying what other types of failures can happen in practice (*e.g.*, priority failures [19]) we could improve Monocle's monitoring capabilities even more. Finally, in Section 4.4.5 we discuss a way for Monocle to probe OpenFlow switches with a pipeline of tables. Unfortunately, the method we present there might not work with legacy switches not designed with OpenFlow in mind — these switches may have serious limitations on how the pipeline works. Yet, being able to work with legacy switches would certainly improve Monocle's usability.

ESPRES: One of the foundations of ESPRES is the assumption that a network update can be split into many sub-updates that are commutative, *i.e.*, that they could be installed in any order (or even interleaved). While this is true in many cases, Dionysus [44] shows that the matter is more complicated if one starts taking link capacity into the account. In such a case, dependencies between switch operations will change over time (*i.e.*, as the individual flows are moved). Here we, however, see a possibility for improvement - Dionysus tries to optimize the total update time and in this aspect it is very similar to not having a scheduler at all. It is, therefore, worth exploring the combination of Dionysus and ESPRES – Dionysus optimizing the total update time while informing ESPRES about the current set of commutative operations, and letting ESPRES decide on the order of operations that are not on the critical path.

Another possible avenue for ESPRES improvement is inter-update scheduling. Currently ESPRES schedules operations only within a single (big) network update. However, the controller might want to install multiple updates, sometimes with a different priority. For example, a controller might start installing a big traffic engineering update but before the installation finishes, a higher-priority, more urgent update might come (*e.g.*, moving flows off a failed link). Normally, it would be hard for the controller to cancel the traffic engineering update while it is running and perform the emergency re-routing (*e.g.*, what if the same flow was being moved by both the traffic engineering and failure recovery?). We envision that by declaring dependencies between updates, ESPRES would be able to: (*i*) prioritize the failure-affected part of the traffic engineering update, (*ii*) install the failure recovery update, and (*iii*) continue with the rest of the traffic engineering.

Chapter 8

Appendix

A.1 Probe Generation is NP-hard

We prove this by providing a polynomial reduction from SAT, *i.e.*, by producing an instance of the probe generation problem for a given instance of SAT. In particular, let I be an instance of SAT, *i.e.*, I is a formula in the conjunctive normal form (CNF). Let $x_1, x_2, ..., x_n$ be variables of I. Our reduction uses n header fields (or, alternatively, n bits of a single header field if it can be arbitrarily wildcarded). The reduction is best illustrated on an example. Consider $I = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_3$. We create several high-priority rules, one for each disjunction of I. In particular, we design the rules so that matching rule R_i logically corresponds to i-th disjunction being false, *i.e.*, the header fields of R_i must match bit 0 for each positive variable, bit 1 for each negative variable, and be wildcarded for each variable not present in the disjunction. In our case, $R_1 := (0, 0, *)$, $R_2 := (*, 1, 0)$, and $R_3 := (*, *, 1)$. Then, we ask for a probe packet matching a low-priority all-wildcard rule $R_{low} := (*, *, *)$ (note that the probe packet must exclude all higher-priority rules). We leave it as an exercise for the reader to prove that a given probe packet is a valid solution to the aforementioned probe generation problem if and only if the values of probe fields interpreted as values of variables are a valid solution to the original SAT instance I

A.2 Encoding Hit, Distinguish, and Collect Constraints as CNF Expressions

In this section we briefly describe how to encode the *Hit*, *Distinguish*, and *Collect* constraints into the conjunctive normal form.

Let $\varphi_1, \ldots, \varphi_n$ be formulas in CNF. Then, we can perform the following operations and obtain a CNF formula as a result

- Conjunction $\varphi := \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$: The formula is already in CNF.
- Disjunction $\varphi := \varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$: We can repeatedly apply the distribution theorem $(\psi_1 \wedge \psi_2) \vee \psi_3 \Leftrightarrow (\psi_1 \vee \psi_3) \wedge (\psi_2 \vee \psi_3)$ to expand the formula into CNF. However, in general, such an expansion may lead to a formula of an exponential size, making it impractical. A better approach is to create an *equisatisfiable* formula, *i.e.*, a formula which is satisfied under the given valuation of variables if and only if the original formula is satisfied. The idea is to create a new formula by introducing new fresh variables; this is usually referred to as Tseitin transform [81]. As an example, consider $\varphi := \varphi_1 \vee \varphi_2$ and a fresh new variable v. We can write $\varphi' := (v \vee \varphi_1) \wedge (\neg v \vee \varphi_2)$ and observe that it is satisfied if and only if at least one of φ_1 and φ_2 is satisfied. It should be mentioned that while it looks that we only swept the problem of disjunctions one level deeper, disjunctions of the form $v \vee \varphi_i$ with v being a literal can be expanded to CNF without an exponential explosion. For longer disjunctions $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$, we use an extended form $\varphi' := (v_1 \vee \varphi_1) \wedge (v_2 \vee \varphi_2) \wedge \cdots \wedge (v_n \vee \varphi_n) \wedge (\neg v_1 \vee \neg v_2 \vee \cdots \vee \neg v_n)$
- Implication $\varphi := \varphi_1 \rightarrow \varphi_2$ is equivalent to $\neg\varphi_1 \vee \varphi_2$
- Variable substitution $\varphi := x \leftrightarrow \varphi_1$ is simply $(x \rightarrow \varphi_1) \wedge (\varphi_1 \rightarrow x)$ or using previous point: $(\neg x \vee \varphi_1) \wedge (x \vee \neg\varphi_1)$
- Negation $\neg\varphi$: It turns out that we need to support only several special cases of the negation:
  - the negation of a literal: $\neg(v) = \neg v$, $\neg(\neg v) = v$
  - the negation of a CNF formula consisting only of single disjunction: $\varphi := \neg(l_1 \vee l_2 \vee \cdots \vee l_n)$ is equivalent to $\neg l_1 \wedge \neg l_2 \wedge \cdots \wedge \neg l_n$ where $l_1, \ldots, l_n$ are literals
  - the negation of a CNF formula where each disjunction is trivial: $\varphi := \neg(l_1 \wedge l_2 \wedge \cdots \wedge l_n)$ is equivalent to $(\neg l_1 \vee \neg l_2 \vee ... \vee \neg l_n)$

- If-then-else chain substitution

$$\varphi := \Big(s = if(i_1, t_1, if(i_2, t_2, if(\dots, if(i_n, t_n, else))\dots))\Big)$$

  First, we substitute all sub-expressions as new fresh variables. Then, we use the following construction from [84]:

$$\begin{aligned}\varphi = &\Big(\neg i_1 \lor \neg t_1 \lor s\Big) \bigwedge \\ &\Big(\neg i_1 \lor t_1 \lor \neg s\Big) \bigwedge \\ &\Big(i_1 \lor \neg i_2 \lor \neg t_2 \lor s\Big) \bigwedge \\ &\Big(i_1 \lor \neg i_2 \lor t_2 \lor \neg s\Big) \bigwedge \\ &\cdots \bigwedge \\ &\Big(i_1 \lor i_2 \lor \cdots \lor i_{n-1} \lor \neg i_n \lor \neg t_n \lor s\Big) \bigwedge \\ &\Big(i_1 \lor i_2 \lor \cdots \lor i_{n-1} \lor \neg i_n \lor t_n \lor \neg s\Big) \bigwedge \\ &\Big(i_1 \lor i_2 \lor \cdots \lor i_{n-1} \lor i_n \lor \neg else \lor s\Big) \bigwedge \\ &\Big(i_1 \lor i_2 \lor \cdots \lor i_{n-1} \lor i_n \lor else \lor \neg s\Big)\end{aligned}$$

  Note that the construction is quadratic in size and therefore very long if-then-else chains should be split by repeatedly substituting some postfix of the chain by a fresh variable.
- Predicate $Matches(P, R)$ is simply a conjunction of per-bit terms defined in Table A.1. When encoding into SAT, we perform trivial simplification by excluding all $True$ terms from the conjunction.
- Predicate $DiffOutcome$ is a disjunction of $DiffRewrite$ and $DiffPorts$. Note that truth value of $DiffPorts$ can be determined in a preprocessing step and as such we can simplify $DiffOutcome$ to either $True$ or $DiffRewrite$.
- Predicate $DiffRewrite(P, R_1, R_2)$ (which corresponds to the expression $rewrite(P, R_1) \neq rewrite(P, R_2)$) is a disjunction (over all bits of P) of expressions from Table A.2; here $P[i]$ represents the variable holding the value of i-th header bit (see $Matches()$ definition), and $R[i]$ is 0, 1 or * depending on whether the rule R rewrites bit to 0, 1 or it does not update the bit). Finally, we can perform trivial simplification on the returned disjunction — remove all $False$ sub-expressions as well as return simply $True$ if one of the sub-expressions is $True$.

$R[i]$	i-th bit of P matches R iff
0	$\neg P[i]$
1	$P[i]$
*	$True$

Table A.1: Converting $Matches(P, R)$ predicate to a CNF formula. The resulting formula is a conjunction of per-bit terms and is satisfied if and only if P matches R.

$R_1[i]$	$R_2[i]$	Bit rewrites are different iff
0	0	$False$
0	1	$True$
1	0	$True$
1	1	$False$
*	0	$P[i]$ (*e.g.*, bit needs to be set to 1)
*	1	$\neg P[i]$ (*e.g.*, bit needs to be set to 0)
0	*	$P[i]$
1	*	$\neg P[i]$
*	*	$False$

Table A.2: Converting $DiffRewrite(P, R_1, R_2)$ predicate to a CNF formula. The resulting formula is a disjunction of per-bit terms and is satisfied if and only if R_1 rewrites at least one bit of P differently than R_2.

Bibliography

[1] AfNOG Takes Byte Out of Internet. http://goo.gl/HVJw5.

[2] Open Network Operating System. http://onosproject.org/.

[3] Open vSwitch: An Open Virtual Switch. http://openvswitch.org.

[4] OpenDaylight. http://www.opendaylight.org/.

[5] OpenFlow Switch Specification. http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[6] OpenFlow Switch Specification v1.1. http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[7] PicoSAT. http://fmv.jku.at/picosat.

[8] POX Controller. http://noxrepo.org.

[9] Reckless Driving on the Internet. http://goo.gl/otilX.

[10] Ryu. https://osrg.github.io/ryu/.

[11] The international SAT Competitions web page. http://www.satcompetition.org.

[12] ONOS: Open Network Operating System, 2014. http://tools.onlab.us/onos-learn-more.html.

[13] Kanak Agarwal, Eric Rozner, Colin Dixon, and John Carter. SDN traceroute: Tracing SDN Forwarding without Changing Network Behavior. In *HotSDN*, 2014.

[14] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *SafeConfig*, 2010.

[15] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[16] Thomas Ball, Nikolaj Bjørner, Aaron Gember, et al. VeriCon: Towards Verifying Controller Programs in Software-Defined Networks. In *PLDI*, 2014.

[17] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[18] Steve Bishop, Matthew Fairbairn, Michael Norrish, et al. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *SIGCOMM*, 2005.

[19] Kai Bu, Xitao Wen, Bo Yang, et al. Is Every Flow on The Right Track?: Inspect SDN Forwarding with RuleScope. In *IEEE INFOCOM*, 2016.

[20] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[21] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[22] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS*, 2006.

[23] Marco Canini, Dejan Kostić, Jennifer Rexford, and Daniele Venzano. Automating the Testing of OpenFlow Applications. In *WRiPE*, 2011.

[24] Martin Casado, Michael J. Freedman, Justin Pettit, et al. Rethinking Enterprise Network Control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[25] DIMACS Challenge. Satisfiability: Suggested Format. *DIMACS Challenge. DIMACS*, 1993.

[26] Cisco. The Zettabyte Era-Trends and Analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni_hyperconnectivity_wp.html, Jun 2016. Online accessed April 14, 2016.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

[28] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency Analysis for Software-Defined Networks. In *PLDI*, 2016.

[29] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16, 1992.

[30] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[31] Nate Foster, Rob Harrison, Michael J. Freedman, et al. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[32] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, 2007.

[33] Gartner. Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015. http://www.gartner.com/newsroom/id/3165317. Online accessed April 14, 2016.

[34] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL*, 1997.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[36] Natasha Gude, Teemu Koponen, Justin Pettit, et al. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38:105–110, July 2008.

[37] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. Where is the debugger for my software-defined network? In *HotSDN*, 2012.

[38] Brandon Heller, Colin Scott, Nick McKeown, et al. Leveraging SDN Layering to Systematically Troubleshoot Networks. In *HotSDN*, 2014.

[39] Brandon Heller, Srini Seetharaman, Priya Mahadevan, et al. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[40] G.J. Holzmann. *The Spin Model Checker - Primer and Reference Manual.* Addison-Wesley, Reading Massachusetts, 2004.

[41] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[42] Sushant Jain, Alok Kumar, Subhasree Mandal, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[43] Karthick Jayaraman, Nikolaj Bjrner, Geoff Outhred, and Charlie Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, MSR, 2014.

[44] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, et al. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[45] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

[46] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[47] Peyman Kazemian, Michael Chang, Hongyi Zeng, et al. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

[48] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[49] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, 2003.

[50] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[51] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Journal on Selected Areas in Communications*, 29(9), 2011.

[52] Teemu Koponen, Martin Casado, Natasha Gude, et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[53] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

[54] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. Providing Reliable FIB Update Acknowledgments in SDN. In *CoNEXT*, 2014.

[55] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What You Need to Know About SDN Flow Tables. In *PAM*, 2015.

[56] Parantap Lahiri, George Chen, Petr Lapukhov, et al. Routing Design for Large Scale Data Centers: BGP is a better IGP! https://www.nanog.org/meetings/nanog55/presentations/Monday/Lapukhov.pdf.

[57] Aggelos Lazaris, Daniel Tahara, Xin Huang, et al. Jive: Performance Driven Abstraction and Optimization for SDN. In *ONS*, 2014.

[58] Hongqiang Harry Liu, Xin Wu, Ming Zhang, et al. zUpdate : Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[59] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *HotNets*, 2013.

[60] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, et al. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[61] Rupak Majumdar, SD Tetali, and Zilong Wang. Kuai: A Model Checker for Software-defined Networks. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2014.

[62] Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, 2010.

[63] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[64] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, 2014.

[65] Dan Pei, Xiaoliang Zhao, Dan Massey, and Lixia Zhang. A Study of BGP Path Vector Route Looping Behavior. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[66] Peter Perešíni, Maciej Kuźniar, Marco Canini, and Dejan Kostić. ESPRES: Transparent SDN Update Scheduling. In *HotSDN*, 2014.

[67] Peter Perešíni, Maciej Kuźniar, Nedeljko Vasić, Marco Canini, and Dejan Kostić. OF.CPP: Consistent Packet Processing for OpenFlow. In *HotSDN*, 2013.

[68] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[69] Yakov Rekhter and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 1654, RFC Editor, July 1995.

[70] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *PAM*, 2012.

[71] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, et al. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *IPSN*, 2010.

[72] SDxCentral. HPE Says Telefónica Just Wants a Multivendor Network. https://www.sdxcentral.com/articles/news/hpe-says-telefonica-just-wants-a-multivendor-network/2015/12/. Online accessed April 14, 2016.

[73] SDxCentral. Lifecycle Service Orchestration (LSO) Market Overview Report. https://www.sdxcentral.com/reports/nfv-lso-mef-report-2016/. Online accessed April 14,2016.

[74] SDxCentral. Top SDx Trends to Watch in 2015 - SDxCentral. https://www.sdxcentral.com/articles/summaries/sdx-trends-watch-2015/2014/12/. Online accessed April 14,2016.

[75] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[76] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for Model Checking SDN Controllers. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[77] Arjun Singh, Joon Ong, Amit Agarwal, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[78] Ahmed Sobeih, Marcelo D'Amorim, Mahesh Viswanathan, Darko Marinov, and Jennifer C. Hou. Assertion Checking in J-Sim Simulation Models of Network Protocols. *Simulation*, 86, 2010.

[79] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, 2002.

[80] Yu-Wei Eric Sung, Sanjay G Rao, Geoffrey G Xie, and David A Maltz. Towards Systematic Design of Enterprise Networks. In *CoNEXT*, 2008.

[81] G.S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In JörgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[82] U.S.News. Software Developer Salary. http://money.usnews.com/careers/best-jobs/software-developer/salary. Online accessed April 15,2016.

[83] Nedeljko Vasić, Dejan Novaković, Satyam Shekhar, et al. Identifying and using energy-critical paths. In *CoNEXT*, 2011.

[84] Miroslav N. Velev. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 310–315, Piscataway, NJ, USA, 2004. IEEE Press.

[85] Willem Visser, K. Havelund, G. Brat, S Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[86] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[87] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature News*, 530, February 2016.

[88] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[89] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[90] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[91] Junfeng Yang, Tisheng Chen, Ming Wu, et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[92] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*, 2006.

[93] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 288–305, Berlin, Heidelberg, 2008. Springer-Verlag.

[94] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.

Biography

Peter Perešíni is originally from Banská Bystrica, Slovakia. He studied at Comenius University in Bratislava, where he received his Bachelor degree in Computer Science in 2009 and his Master of Science degree with distinction in 2011. During his studies, Peter competed in numerous math, physics and informatics competitions, often achieving high positions in the ranking. He also worked multiple times as an intern at Google, working on different projects.

In 2011, Peter joined EPFL's EDIC doctoral program to pursue his Ph.D. under the supervision of Prof. Dejan Kostić. When his advisor moved, Peter stayed at EPFL under Prof. Willy Zwaenepoel. In July 2016 Peter obtained his Ph.D. from EPFL in Switzerland.

Peter's research interests include Software-Defined Networking (with the focus on correctness, testing, reliability, and performance), Distributed Systems, and Computer Networking in general. His publications (as of June 2016) include:

- P. Perešíni, M. Kuźniar, M. Canini, D. Venzano, D. Kostić and J. Rexford. *Systematically testing OpenFlow controller applications.* In Computer Networks, Elsevier, vol. 92, p. 270-286, 2015.
- P. Perešíni, M. Kuźniar and D. Kostić. *Monocle: Dynamic, Fine-Grained Data Plane Monitoring.* The 11th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), December 2015.
- M. Kuźniar, P. Perešíni and D. Kostić. *What You Need to Know About SDN Flow Tables.* Passive and Active Measurements Conference (PAM), 2015.
- M. Kuźniar, P. Perešíni and D. Kostić. *Providing Reliable FIB Update Acknowledgments in SDN.* The 10th International Conference on

emerging Networking EXperiments and Technologies (ACM CoNEXT), December 2014.

- P. Perešíni, M. Kuźniar, M. Canini and D. Kostić. *ESPRES: Transparent SDN Update Scheduling.* ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), 2014.
- P. Perešíni, M. Kuźniar, M. Canini and D. Kostić. *ESPRES: Easy Scheduling and Prioritization for SDN.* Open Networking Summit (ONS) Research Track, 2014.
- P. Perešíni and D. Kostić. *Is the Network Capable of Computation?* The 3rd International Workshop on Rigorous Protocol Engineering (WRiPE), 2013.
- P. Perešíni, M. Kuźniar and D. Kostić. *OpenFlow Needs You! A Call for a Discussion About a Cleaner OpenFlow API.* The 2nd European Workshop on Software Defined Networks (EWSDN), 2013.
- P. Perešíni, M. Kuźniar, N. Vasic, M. Canini and D. Kostić. *OF.CPP: Consistent Packet Processing for OpenFlow.* ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), 2013.
- M. Kuźniar, P. Perešíni, M. Canini, D. Venzano and D. Kostić. *A SOFT Way for OpenFlow Switch Interoperability Testing.* The 8th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), December 2012.
- M. Canini, D. Venzano, P. Perešíni, D. Kostić and J. Rexford. *A NICE Way to Test OpenFlow Applications.* The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2012.