Measuring and Managing Switch Diversity in Software Defined Networks

THÈSE N^{O} 7074 (2016)

PRÉSENTÉE LE 7 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

Maciej Leszek KUŹNIAR

acceptée sur proposition du jury:

Prof. A. Ailamaki, présidente du jury
Prof. W. Zwaenepoel, Prof. D. Kostić, directeurs de thèse
Prof. M. Canini, rapporteur
Prof. L. Vanbever, rapporteur
Prof. K. Argyraki, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2016

Sommario

L'approccio Software Defined Networking (SDN) è una novità nel campo delle reti informatiche. La migliore flessibilità, la gestione semplificata e la riduzione dei costi promesse dal paradigma SDN fanno pensare a molti che questo sia il futuro delle reti. L'intuizione principale di SDN sta nella separazione tra la configurazione, il controllo della rete e l'instradamento dei pacchetti a basso livello. In questo modo, lo sviluppo del complesso software di controllo diventa indipendente dai cambiamenti di instradamento e switching a livello hardware. Facendo affidamento su un'interfaccia ben definita, un programma di controllo pu supervisionare l'intera rete composta da parecchi switch, di diversi produttori. Nella mia tesi argomento, invece, che questa visione idealizzata difficile da ottenere in pratica e che un controllore SDN non può trattare tutti gli switch in modo eguale. Ci sono parecchie ragioni per le quali switch che implementano la stessa specifica rimanono eterogenei: specifiche non chiare, difficili implementazioni, costi ed errori umani.

In questa tesi descrivo un approccio a due fasi per gestire l'eterogeneit degli switch SDN. In primo luogo presento delle tecniche sistematiche per rilevare differenze tra componenti diversi. SOFT è un metodo, che ho definito ed implementato, che mette allo scoperto differenze funzionali tra diversi software di switching. Si basa su ben note tecniche di analisi software ed un nuovo modo di applicare un risolutore di constraint per trovare input che causano un diverso comportamento degli switch. Inoltre, definisco una metodologia sistematica per effettuare misure di prestazioni sulle richieste di aggiornamento degli switch. Sviluppo anche un tool di misura basato su questa tecnica che è in grado di computare le caratteristiche del tasso di aggiornamento degli switch hardware. Questa metodologia, in aggiunta,

tracciando le interazioni tra lo strato di controllo e di gestione dati durante gli aggiornamenti, è in grado di rivelare le inconsistenze che mostrano quando gli switch non seguono la specifica. Infine risolvo il più grosso problema rilevato. RUM, uno strato software che si pone tra il controllore e gli switch, maschera e aggiusta le notifiche di aggiornamento scorrette che arrivano da switch problematici.

Dimostro l'utilit della soluzione descritta analizzando degli switch esistenti. SOFT individua diverse inconsistenze tra due switch open source. Il benchmark di prestazioni rivela degli errori che compromettono la sicurezza di rete. Fornisce anche delle caratteristiche dettagliate sugli switch che dovrebbero essere prese in considerazione dagli sviluppatori di applicazioni lato controllore per migliore le prestazioni di rete. RUM previene perdite di pacchetti con un aggiornamento dell'instradamento sicuro senza che alcuna modifica sia necessaria agli switch problematici.

Parole Chiave:

Software Defined Network, switch, affidabilità, interoperabilità, prestazioni, flow table, aggiornamenti, OpenFlow

Abstract

Software Defined Networking (SDN) is a novel approach to building computer networks. Improved flexibility, simplified management and cost reduction promised by SDN makes many see it as the future of networking. The main insight of SDN is the separation of network control and configuration decisions from packet forwarding devices. This way, complex control software development becomes independent of changes in hardware traffic forwarding switches. Relying on a well-defined interface, a controller program can supervise the whole network built of many switches, produced by multiple vendors. I argue however, that this idealized vision is difficult to achieve in practice and that an SDN controller cannot treat all switches equally. There are multiple reasons why switches following the same specification are heterogeneous: unclear specification, implementation difficulties, cost, and human errors.

In this dissertation, I describe a two-phase approach to handle switch diversity in SDN. First, I present systematic techniques to detect various differences between devices. SOFT is a method and a tool that uncovers functional differences in switch software. It relies on established software analysis techniques and a novel application of a constraint solver to find inputs that result in distinct behavior of two switches. Further, I design a systematic methodology for switch update performance measurements. A benchmarking tool based on this technique computes update rate characteristics of hardware switches. By additionally tracking interactions between control and data planes during the update, this methodology is capable of revealing inconsistencies showing that the switches do not follow the specification. Finally, I address the most severe issue detected. RUM, a software

layer between the controller and the switches masks and fixes incorrect rule update notifications coming from faulty switches.

I demonstrate the usefulness of the described solutions by analyzing existing switches. SOFT detected several inconsistencies between two open source software switches. The performance benchmark revealed errors that compromise network security. It also provided detailed switch characteristics that should be taken into account by controller developers to improve network performance. RUM prevents packet drops in a safe network update without requiring any changes to faulty switches.

Keywords:

Software Defined Networks, switches, reliability, interoperability, performance, flow table updates, OpenFlow

Acknowledgements

This dissertation would not be possible without help of many people.

First, my greatest gratitude goes to my advisors Prof. Dejan Kostić and Prof. Willy Zwaenepoel. Dejan invited me to EPFL and gave me freedom to explore topics that excited me, while offering his excellent advice and guidance. He taught me how to successfully identify, produce and present scientific contributions. Willy offered his continuous support and made sure that I stayed on the right track during my studies.

I would like to thank the Jury members, Prof. Anastasia Ailamaki, Prof. Katerina Argyraki, Prof. Marco Canini and Prof. Laurent Vanbever for the time they took to read and evaluate this thesis and my research.

I am especially grateful to Marco Canini for the support and guidance I received from him early in my PhD journey. Later on, Marco was always willing to share his opinions and ideas on projects I was taking on.

Peter Perešíni and Daniele Venzano have been my great officemates and collaborators on multiple projects. I enjoyed many stimulating discussions with them, and learned a lot working with them.

Finally, I would like to thank all my friends and colleagues in Lausanne for making my stay here pleasant and entertaining.

Last but not least, I am grateful to my parents for their encouragements and unconditional support over the years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Computer Networks

Computer networks are essential components of modern life. Whether to work, get entertained, look for information, do shopping, or contact others, people use the Internet and cloud services every day. As part of the critical infrastructure, computer networks need to be reliable and keep working all the time, even when facing network element failures. Moreover, the ever growing need for connectivity poses strong extensibility and maintainability requirements for the networks.

Years of research have led to a slowly evolving network architecture characterized by a set of best practices. To avoid single points of failure, networks are composed of multiple independent devices (*e.g.*, switches, routers, middleboxes) that communicate using well-defined protocols. This way, after an addition or a failure of one entity, the remaining nodes can follow standardized procedures to reconverge their packet forwarding states. Further, by relying on common protocols, the devices produced by different vendors and managed by different operators can work together in one global network.

However, traditional computer networks are difficult to manage and leave little room for innovation. Introducing new functionalities or modifying the existing ones requires support at many levels. First, the standardizing body needs to define a new protocol. This process often takes years and is followed by detailed interoperability testing, as the new protocol cannot be in conflict with existing ones. Since switches and routers run proprietary software to

implement the standardized protocols, any implementation change requires vendors' intervention. If the feature is in low demand, switch and router developers may take a long time to provide it. Further, administrators configure each device individually at the protocol level, using its configuration interface. The configuration interfaces and commands differ across vendors, which leads to vendor-specific courses and certificates for network operators. As a result, network administrators have to be familiar with many protocols and many configuration interfaces. Adapting new protocols and devices in a network induces an additional cost of staff training. A recent legal conflict between Cisco and Arista about using similar command line interfaces shows that networking gear vendors are not interested in changing this situation [18]. All the above results in vendor lock-ins, where the network owners are forced to heavily depend on a single switch/router provider. Finally, since the network is configured at a protocol level with many protocols running concurrently, it is difficult to reason about a global network state.

While a majority of computer networks is still following the traditional approach, research efforts to simplify network management are quickly getting traction.

1.2 Software Defined Networks

Software Defined Networking (SDN) is a recent approach to build computer networks that promises to simplify network management and lower the barrier for deploying new functionalities. It is gradually replacing the traditional design in data center [78], campus [54], and wide area networks [39, 43, 45].

The core concept in the SDN paradigm is physical separation of the data and control planes in the network [64]. The control plane decides how to configure the network and the data plane forwards packets according to this

configuration. Unlike in the traditional case, where both planes were located inside the same device (*e.g.*, switch, router), in SDN the control plane of multiple switches is realized in a single software program running on external computers. One control program supervises multiple data plane devices via a general configuration protocol.

This novel design solves many problems present in the traditional approach to networking. First, introducing a new functionality requires a network operator to prepare a software program instead of making individual changes to hardware devices. Moreover, the control program is written in a high level programming language. Since the switches are configured externally, they become simpler and cheaper. Complexity gets moved to a software program running on a server machine. Further, as vendors are no longer involved in the process, network innovation accelerates and becomes more dynamic. Network operators get flexibility — they choose what functions they need and want to provide and can do it quickly.

1.2.1 OpenFlow

OpenFlow [64] is the most popular open protocol used to realize SDN concepts in practice. The protocol specification defines an interface and messages exchanged between the control program (often called controller) and data plane packet forwarding devices called OpenFlow switches. Figure 1.1 shows an overview of the OpenFlow-based design.

1.2.1.1 OpenFlow Controller

A controller serves a crucial role in a software defined network. It collects statistics and other information from the network and makes appropriate reconfiguration decisions. Additionally, since the controller runs on a reg-

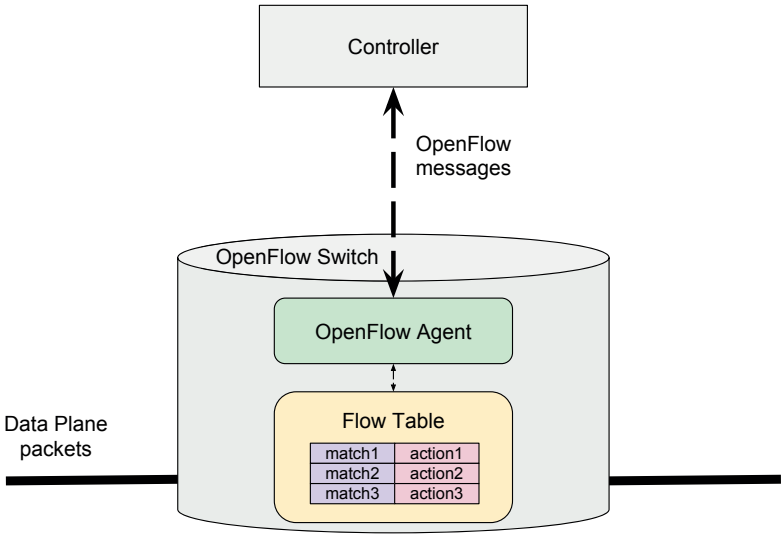


Figure 1.1: An overview of SDN realized with OpenFlow. The controller configures the OpenFlow switch using OpenFlow messages sent over a secure channel. OpenFlow agent running at the switch parses the messages and updates rules in the flow table accordingly.

ular computer, it can also pull data from other sources than the switches. Consider, for example, a control program that adjusts routing based on a machine reservation system in a cloud or expected popular social events in a mobile network.

An OpenFlow controller is usually constructed as a layered, modular system. On top of a controller platform, developers create so called controller applications that realize particular functionalities such as topology discovery, routing, load balancing. While controller platforms are often compared to operating systems [38, 75], basic platforms such as NOX [6] provide only message handling and translation between data structures and the wire format. More advanced, distributed platforms such as Onix [56] and ONOS [19] spread state across multiple machine to improve fault tolerance. Contrary to a common misconception, an SDN controller does not have to be centralized. Commercial solutions rely on distribution and replication to improve perfor-

mance and reliability. The controller platforms often come with a set of basic and frequently used applications such as shortest path routing and link state detection. This way other, problem-specific applications reuse the common code. In the end, a fully functional controller contains a mesh of interdependent applications. Since the distinction between the controller platform and the applications is often blurry, in this dissertation the term controller refers to the entire modular control program.

Controller platforms try to balance programming simplicity and performance by choosing different programming languages such as C++ (NOX [6]), Java (Floodlight [4], Beacon [32]), Python (POX [12], Ryu [14]). Finally, as is the case for any modern programming technique, various supporting tools and abstractions make programming networks simpler. The control software developer can use one of the high level network programming abstractions to reason about the network at a high level in an intuitive manner [35, 47].

1.2.1.2 OpenFlow Switches

An OpenFlow switch is a generic packet forwarding device that replaces switches, routers, and middleboxes known from traditional networks. There are two main logical components of an OpenFlow switch: a flow table and an OpenFlow agent.

OpenFlow agent. An OpenFlow agent is a software component running at a switch. Its role is to expose a standardized programmatic interface to the switch forwarding tables and to handle the communication with the controller. The agent also performs initial message validation and error checking.

Flow table. The controller directly configures the switches by inserting low-level forwarding rules in their flow tables. The flow table is a set of match-action rules that define how the switch should handle packets. Each

forwarding rule consists of a general match pattern that matches packet headers and optionally a port at which the packet arrived. The matches span fields belonging to multiple encapsulation layers and traditional networking protocols (Ethernet, IP, TCP/UDP, and others). Moreover, each match can either define the expected value of each field, or mark a field as unimportant by wildcarding it. Actions include modifying headers, dropping and forwarding packets to one or many ports, or encapsulating them in an OpenFlow message and sending to the controller for further analysis. For each data plane packet, the switch applies instructions or actions defined by the highest priority rule matching this packet. Additionally, each rule has a set of counters assigned. These counters collect statistics about the number of packets and bytes forwarded by a given rule. Newer OpenFlow specifications introduce a notion of multiple flow tables that get chained and form a packet processing pipeline.

In practice switches realize the abstract flow table as two separate tables. First, an OpenFlow agent parses an incoming rule modification message, checks for errors and places the desired rule in the software flow table. Afterwards, the updates are propagated to the hardware flow table managed by an application-specific integrated circuit (ASIC). The main reason for this split is limited bandwidth between the switch CPU and the ASIC. It can become a bottleneck for frequent rule updates [29]. Independently, in private communication switch vendors confirmed that to reduce the impact of hardware updates, they batch multiple rule modifications before sending them to the ASIC. Moreover, to provide quick matching for general and partially wildcarded matches required by SDN, rules in the hardware table are stored in Ternary Content-Addressable Memory (TCAM). This type of memory is expensive and has high power requirements [17], which limits its

available capacity. Therefore, switches may choose not to move some rules to hardware [21, 48]. On the other hand, only rules placed in the hardware flow table can guarantee high packet forwarding rates. Header matching and packet forwarding in software is slower and overloads the switch CPU.

1.2.1.3 OpenFlow Commands

While this work considers all OpenFlow commands, these four are most relevant to network reconfiguration: rule modifications (`Flow Mod`), `Barrier`, `Packet Out`, and `Packet In`.

Rule modifications (`Flow Mod`). The controller instructs the switch how to configure its flow table using flow table modification messages (`Flow Mod`, sometimes called rule modification message in this thesis). There are three subtypes of the rule modification message: addition of a new rule, deletion of an existing rule, and change of an existing rule.

Barrier. Barrier is a multipurpose message used to order other messages and to synchronize the switch and controller states. After receiving a `Barrier` request, the switch has to finish processing all previously received messages before executing any messages after the `Barrier` request. When the processing is complete, the switch must send a `Barrier` reply message [11]. All older specifications do not clarify what processing means exactly. Since OpenFlow version 1.5, it is defined that a switch can send the `Barrier` reply only when all earlier messages are processed such that their effects are visible in the data plane.

Packet Out. The controller uses the `Packet Out` command to inject packets in the network data plane directly from a particular switch. The command specifies the packet header and its content, as well as on which of the switch ports should the packet appear.

Packet In. The `Packet In` command is the opposite of `Packet Out`. This message encapsulates a data plane packet and sends it to the controller using the control channel. There are two scenarios when such an event occurs: (*i*) the packet matches a rule that specifies a special "send to the controller" action, or (*ii*) there is no matching rule and the switch sends packets to the controller by default. The message contains the encapsulated packet itself (or a part of its header) and the ingress port where the packet arrived.

1.2.2 Evolution of Network Devices

Years of steady development has lead to expensive and complex routers that implement hundreds of protocols. In 2011, Cisco routers supported 700 different standards [15] while Juniper supported over 250 standards related to routing alone [16]. Systems running on these machines contain over 20 millions lines of code and each device has to be separately updated.

The SDN revolution started in software switches first. Open vSwitch [69] providing OpenFlow 1.0 support enabled first local OpenFlow tests. While processing of packets in software is slower than in hardware, a modern evolution of the Open vSwitch is capable of connecting virtual machines and serves as the network edge in a virtualized environment [55]. This switch implementation has been constantly maintained and developed to keep up with the most recent protocol versions. It was also followed by other software switches with OpenFlow support [3, 5].

When established networking gear vendors realized that SDN is getting popular and there is demand for hardware OpenFlow switches, they started adding extensions to their existing devices. The OpenFlow agent in these switches is often based on Open vSwitch. Legacy devices with patched software, such as HP ProCurve 5400zl, Dell PowerConnect 8132F and NEC

IP8800 offer limited support for OpenFlow. Existing forwarding, routing, and filtering tables have insufficient flexibility to perform general matches. Their capacity is also insufficient for SDN requirements. Moreover, not all required actions can be applied in hardware, and complex rules get rejected or get placed in the software flow table. Since these devices were not designed for frequent communication between CPU and ASICs, rule update rate is low. Despite all the shortcomings, this first generation of switches allowed for initial prototype deployments and many of them still are the core of many academic networks [7].

Finally, new companies such as Pica8 and NoviFlow took advantage of the SDN popularity and entered the market with switches purposefully built for programmable networks. The new generation of devices claims full OpenFlow support in multiple versions and offers improved performance. Their hardware flow tables fit thousands of general rules with arbitrary matches and actions.

The most recent trend in switch design argues for making switch hardware interface fully open. Traditionally, switch hardware, an operating system and specific feature implementations running at the switch were proprietary and accessible only through a vendor-specific configuration interface. While early SDN efforts standardized the interface and simplified the switch software, the whitebox switches proposal goes a step further [76]. In a whitebox switch only bare hardware is provided by vendors. Both the operating system and applications are developed independently and users can get them from other sources. This design is similar to a model successfully applied to personal computers and servers for years.

1.3 Motivation and Goals

The main assumption of Software Defined Networking is that all data plane elements are configured using a common, well-defined programming interface (API). Open APIs such as OpenFlow hide technological and implementation details and give switch vendors freedom in choosing the best internal design.

However, because of the rapid development, devices that belong to different generations of SDN switch design exist in networks concurrently. This diversity is further increased by several issues that make it difficult to produce error-free and functionally equal switch software and hardware.

First, the specification is often ambiguous and can be interpreted in multiple, seemingly equally correct ways. Despite constant improvements and clarifications in new specification revisions, there is no guarantee that two independent teams implementing the same functionality will always understand all the details in the same way.

Second, switch vendors sometimes have explicit implementation freedom that can affect network behavior.

Finally, especially the newest specifications define a large set of desired functions some of which are difficult or expensive to realize. Thus, vendors may choose to simplify them or not to provide them at all, choosing higher performance and lower cost over correctness.

Therefore, despite following the same specification and exposing a theoretically unified interface, switches are likely to behave differently in some corner case scenarios. If controller developers and network administrators are unaware of these differences, the network behavior can become suboptimal, unexpected or even incorrect. However, while switches are the main component of Software Defined Networks and their correct behavior is paramount

for network reliability, there is little interest in switch testing [8, 72]. As for any new technology, most of the SDN research is devoted toward developing new functionalities. Testing efforts that exist focus on the controllers and network policies as a whole. They assume correctly working switches and abstract them behind simplified models.

My goal in this dissertation is to start the process of *bridging the gap between the reality of heterogeneous switches and the theory of the unified network view promised by the SDN vision.* To this end, switches should be interoperable from the point of view of the controller, which means that switches supporting the same protocol version should be treated by the controller equally, regardless of their implementation.

Two phases are essential for reaching this goal.

- Well-established methodologies to systematically measure, analyze and understand switch heterogeneity are required to test existing and new switches. Such testing tools will detect problems with new devices. Depending on severity, the detected issues should be addressed by vendors, acknowledged by controller developers and network operators, or handled by runtime tools.

- A multilevel chain of runtime tools should mask the detected differences without requiring changes to the switches or to the controller. Applying modifications to hardware devices such as switches is troublesome and slow because it requires vendor efforts. On the other hand, it would be wasteful and repetitive for all controllers to handle switch heterogeneity on their own. Instead, I envision a chain of software-based proxies layered between switches and controllers that mask both functional and performance based differences to some extent.

1.4 Solution Overview

I first set to understand how big of an issue the switch heterogeneity is in practice. There are two sources and types of diversity.

First, differences and errors in OpenFlow agent source code could lead to inconsistent functional behavior of the switches. Since an agent is in essence a regular program, I apply an established systematic software analysis technique — symbolic execution. Symbolically executing a program generates its mathematical representation: each execution path is encoded as a set of constraints which must hold for the execution of that path. Additionally, the path execution produces a corresponding output. The constraints split the space of switch inputs into subspaces. All elements in a given subspace result in the same output. In the second testing phase, I check if there are any inconsistencies between the two switches by intersecting their input/output spaces. Such a two phase approach solves the main challenge of this problem — how to compare two switches without requiring simultaneous access to their source codes.

The second type of inconsistency cannot be detected using static source code analysis alone. It is related to interactions between the switch software and hardware. Here, I designed a switch flow table update measurement methodology that systematically covers the input space (rule modification request sequences). The main feature of this methodology is that it assumes not only precise observation of the control channel, but also requires constant data plane monitoring. A switch benchmark that implements the methodology also injects and captures packets matching newly installed rules. This approach revealed various timing related issues with existing OpenFlow switches.

Finally, I propose a software-based solution to one of the detected problems. Switches differ in terms of how precise they report rule modifications. The biggest observed divergence from the truth is in order of minutes — a delay that may even compromise network security. RUM is a software layer between faulty switches and the controller that relies on additional techniques to make sure that the switch really modifies a rule. The techniques vary from simply waiting for a safe time, through using artificial, probe rule modifications, up to full probing of the network data plane. They differ in terms of the provided guarantees, assumptions about the switches, and induced overhead.

1.5 Thesis Contributions

While working on achieving the thesis goals, I make the following contributions:

- I apply symbolic execution to systematically identify and compare code paths in OpenFlow agents to determine input subspaces that result in the same outputs. In the process, I identify what combinations of symbolic and concrete inputs guarantee satisfactory running time without sacrificing coverage. I also show how observing external actions of an agent leads to conclusions about its internal state.

- I demonstrate a novel use of a constraint solver to compute an intersection of input subspaces for different agent implementations. It quickly reveals inputs that cause different behavior (inconsistencies) in multiple agents. In addition, it works without any definition of correct behavior. This phase is separate from symbolic execution and does not require access to switch source code.

- I devise a systematic methodology for switch control plane performance testing along many different dimensions. The methodology focuses on measuring the interactions between the control plane and the data plane. I also create and publicly release a benchmarking tool that implements this methodology.

- I propose techniques that hide imprecise rule update notifications in various SDN switches. The techniques offer different precision and guarantees depending on the induced overhead and assumptions made about the switch. All assumptions are based on the characteristics observed in practical experiments. To validate my solutions, I create RUM: a software layer that implements these techniques without requiring any modifications to controllers and switches.

- I demonstrate the effectiveness of the technique that detects functional differences between switches by applying it to the Reference Switch (55K lines of code) and Open vSwitch (80K lines of code), the two publicly available OpenFlow agent implementations. A tool implementing the aforementioned technique found several inconsistencies between the two switches.

- I show the usefulness of the performance measurement methodology by presenting a detailed study of switch flow table update rates. I also report several types of anomalous behavior in OpenFlow switches that were never revealed before.

- I show that RUM guarantees rule update confirmations precise enough to prevent any packet drops in a safe network update scenario, that loses traffic for up to 290ms otherwise.

1.6 Document Organization

In this chapter I presented a brief overview of Software Defined Networking and outlined my motivation for systematic analysis of the emerging SDN switches.

In Chapter 2, I present a systematic approach to detecting functional differences between OpenFlow switches. SOFT is a tool that relies on recent advancements in software analysis techniques to detect inputs for which two tested switches return different outputs.

Chapter 3 presents an exhaustive study of flow table update characteristics in SDN switches. The systematic methodology introduced in this chapter allows for measuring and analyzing both bare update performance, as well as finding corner cases and surprising behaviors.

In Chapter 4, I present a software-based solution to the inconsistent and incorrect Barrier implementations detected and described in Chapter 3. RUM is a middle layer between the switches and a controller. It provides the controller with reliable rule update confirmations even when working with unreliable switches.

Finally, I present the related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Detecting Functional Differences

This chapter focuses on the question how unclear specification and implementation freedom affect switches from the functional point of view. SOFT (Systematic OpenFlow Testing) — a tool introduced here — automates interoperability testing of OpenFlow switches by applying software analysis techniques. To achieve exhaustive testing, we present an approach that leverages the multiple, existing OpenFlow implementations to identify potential interoperability problems by crosschecking their behaviors. Instead of defining what the correct, expected behavior is, our method compares behavior of the tested devices. Exploring code behaviors in a systematic way is key to observing inconsistencies. Operating in two phases, SOFT uses symbolic execution and constraint solving. In the first testing phase, symbolic execution runs locally on each vendor's source code. Then, using the outputs of symbolic execution (not the source codes), SOFT determines the input ranges (*e.g.*, fields in OpenFlow messages) that cause two OpenFlow agent implementations to exhibit different behaviors.

2.1 Defining Inconsistencies

Switches that are capable of supporting the OpenFlow Switch Specification [11] do so by running an OpenFlow agent. This agent is a piece of software primarily responsible for state management. It receives and processes control messages sent by OpenFlow controllers (*e.g.*, `Flow Mod`, `Packet Out`, *etc.*), and configures the switch forwarding tables accordingly to the given commands. In addition, the OpenFlow agent may take part in packet for-

warding itself — in a hardware switch, for packets that are forwarded to the controller; in a pure software implementation, for every packet.

As such, the execution of the OpenFlow agent is mainly driven by external events (*e.g.*, rule installation requests). There are two channels that provide data to the switch: control channel to the controller and data plane connections. We make no distinction and use the term *inputs* to call the data arriving at the agent as either OpenFlow control messages or data plane packets.

Intuitively, an *inconsistency* occurs when two (or more) OpenFlow agents that are presented with the same input sequence produce different results. Here, the results refer to both externally observable consequences when processing an input (*e.g.*, replying to a request for flow table statistics), and internal state changes (*e.g.*, updating the flow table with a new entry).

To be able to identify inconsistencies, we assume the agents support the OpenFlow interface and we check for inconsistencies in operations at the interface level. To crosscheck behaviors, we rely either on externally observable results or, when necessary, on the probe packets to infer the internal state.

Note that, we are uninterested in verifying the underlying switching hardware's correctness. In fact, such verification is typically already part of the ASIC design process. However, we assume that there is a way to execute the OpenFlow agent without the switching hardware, *e.g.*, through an emulation layer that is commonly readily available for development and testing purposes.

2.2 Symbolic Execution Background

Our approach is inspired by the successful use of symbolic execution [53] in automated testing of systems software [23–25, 28, 37]. The idea behind sym-

bolic execution is to exercise all possible paths in a given program. Therefore, unlike normal execution that runs the program with concrete values, symbolic execution runs program code on symbolic input variables, which are initially allowed to take any value. During symbolic execution, code is executed normally until it reaches a branch instruction where the conditional expression *expr* depends (either directly or indirectly) on a symbolic value. At this point, program execution is logically forked into two executions — one path where the variables involved in *expr* must be constrained to make *expr* true; another path where *expr* must be false. Internally, the symbolic execution engine invokes a constraint solver to verify the feasibility of each path. Then, program execution resumes and continues down all feasible paths. On each path, the symbolic execution engine maintains a set of constraints, called the path condition, which must hold for the execution of that path. For every explored path, symbolic execution passes the path condition to a constraint solver to create a test case with the respective input values that led execution on that path. Since program state is (logically) copied at each branch, the symbolic execution engine can explore multiple paths simultaneously or independently.

Like others [57], we observe that, to deal with loops, symbolic execution would potentially need to explore an unbounded number of paths. As described in Section 2.5.2, we effectively side-step this problem by exploiting knowledge of the OpenFlow message grammar to construct inputs that ensure we explore a bounded number of paths.

Therefore, symbolic execution is a powerful program analysis technique — rather than having a linear execution where concrete values are used, symbolic execution covers a tree of executions where symbolic values are used. However, the usefulness of symbolic execution is limited by its scalability be-

cause the number of paths through a program generally grows exponentially in the number of branches on symbolic inputs. This problem is commonly known as the "path explosion" problem. The path explosion is exacerbated by the fact that the program under test interacts with its environment, e.g., by invoking OS system calls and calls to various library functions. External functions present an additional problem if the symbolic execution engine does not have visibility into their source code. A typical solution to this problem is to abstract away the complexity of the underlying execution environment using models. These models are typically a simplified implementation of a certain subsystem such as file system, network communication, etc. Besides using environment models to "scale" symbolic execution, it is possible and often sufficiently practical to selectively mark as symbolic only the inputs that are relevant for the current analysis. As we show later in Section 2.5.2, carefully mixing symbolic and concrete inputs is key to being able to symbolically execute OpenFlow agents.

2.3 SOFT Overview

Our approach to automatically finding inconsistencies among OpenFlow agent implementations is most easily introduced through an example.

Consider an input sequence that only includes one control message of type `Packet Out`. This message instructs the OpenFlow agent to send out a packet on port p, where p is a 16-bit unsigned integer that identifies a specific port or is equal to one of several preset constants (*e.g.*, flood the packet or send to controller). For the sake of presentation, we assume that only p is symbolic (*i.e.*, p is the only part of this input that varies) and we omit the case $p = 0$ (for which an error message would be produced).

We first symbolically execute an OpenFlow agent implementation while

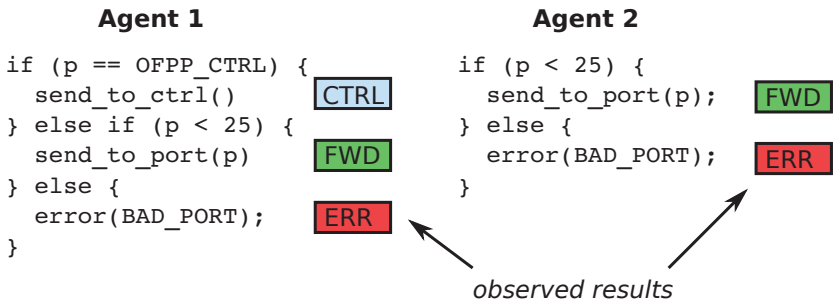


Figure 2.1: Example OpenFlow agents having different PACKET_OUT message implementations.

feeding it with this input sequence. When executing symbolically, we automatically partition the input space of p into several subspaces. Each subspace is an equivalence class of inputs that, in this case, describes which values of p follow the same code path. To make the point more tangible, consider Agent 1 in Figure 2.1: if $p \in [1, 24]$ the program executes the code path that sends the packet on port p; if $p =$ `OFPP_CTRL` (the predefined controller port) the program executes a different code path that encapsulates the packet in a `Packet In` message and sends it to the controller; and so on. Besides determining the input space partition, we log the output results produced when executing each code path (*e.g.*, we log what packet comes out from which port). Therefore, for each input subspace there exists a corresponding output trace.

Next, we symbolically execute a different OpenFlow agent implementation (Agent 2 in Figure 2.1) and determine the partitions of input space of p. However, assume that this second OpenFlow agent does not support the special port number `OFPP_CTRL`. Instead, the program sends an error message to the controller when it encounters this case. Likewise, we log the output results produced when executing each code path.

At this point, we have two input space partitions (one for each OpenFlow agent implementation), as depicted in Figure 2.2. Within each partition, we

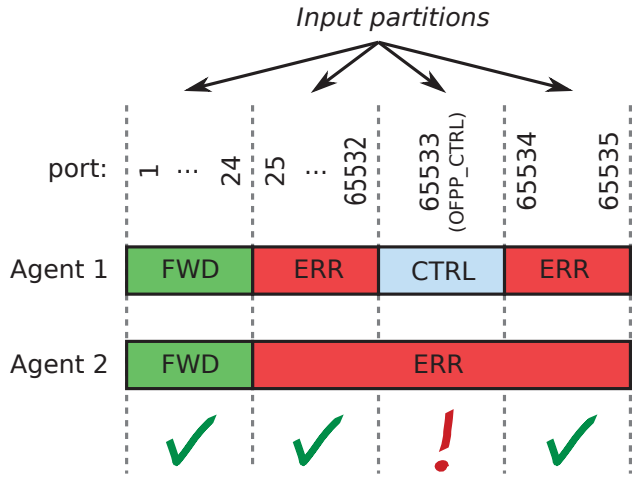


Figure 2.2: Input space partitions and inconsistency check.

then group the subspaces by output result (illustrated with different colors in Figure 2.2). That is, we merge together two subspaces (two code paths), if they produce the same outputs. Such grouping results in two coarse-grained input space partitions–one for each agent. Next, we consider the cross product of the coarse-grained partitions (*i.e.*, all pair-wise combinations of subspaces between the two partitions). From the cross product, we exclude pairs of subspaces that correspond to identical output results. Finally, we intersect the two subspaces in every remaining pair. A *non-empty intersection* defines a subspace of inputs that give different results for different OpenFlow agents: this is an inconsistency. For each inconsistency we discover, we construct a concrete test case that reproduces the observed results. Relative to our current example, we identify that one inconsistency exists and, to reproduce it, we construct the example with input $p =$ `OFPP_CTRL` as illustrated in Figure 2.2.

2.4 Usage

It is impractical to assume that a tool for interoperability testing has access to the source code of commercial OpenFlow implementations from all vendors. Therefore, the goal is to make symbolic execution scale to crosscheck different OpenFlow implementations and find interoperability issues *without having simultaneous access to all source codes*. The proposed solution allows switch vendors to use SOFT in two phases. In the first phase, each vendor independently runs SOFT on its OpenFlow agent implementation to produce a set of intermediate results that contain the input space partitions and the relative output results. One benefit of this approach is that a vendor does not require access to the code of other vendors.

In the second phase, SOFT collects and crosschecks these intermediate results to identify inconsistencies. This phase can take place as a part of an inter-vendor agreement (*e.g.*, under an NDA), or during wider interoperability events [9]. Alternatively, a third-party organization such as Open Networking Foundation (ONF) may conduct the tests.

While we focus the presentation of SOFT on interoperability testing, we want to clarify that there exist other applications. For example, SOFT can automate performing regression testing. In addition, it can be used to compare against a well-known set of path conditions that are bootstrapped from unit tests.

We observe that an OpenFlow agent is potentially a software component of a hardware device. As such, some operations can install state directly in the switching hardware (*e.g.*, forwarding rules), seemingly outside of SOFT's reach. We note, however, that vendors typically have a way of running their firmware inside a hardware emulator for testing purposes. We only

require that the hardware emulator is integrated with the symbolic execution engine. Previous work (*e.g.*, [28]) demonstrates that it is indeed possible to run complex software systems live, including closed-source device drivers.

2.5 Design

Our goal is to enable systematic exploration of inconsistencies across multiple OpenFlow agent implementations. In other words, we want to find whether there exists any sequence of inputs under which one OpenFlow agent behaves differently than another agent. To do this, we require a way of (*i*) constructing sequences of test inputs that cover all possible executions for each OpenFlow agent, and (*ii*) comparing the output results that each input produces to identify inconsistencies.

We accomplish the subgoal of finding test inputs by using symbolic execution. The outcome of symbolic execution is twofold: (*i*) a list of path conditions, each of which summarizes the input constraints that must hold during the execution of a given path, and (*ii*) a log of the observed output results for each path executed.

We then identify inconsistencies by grouping the path conditions that share the same output results on a per-agent basis and finding the input subspaces that satisfy the conjunction of the path conditions. Figure 2.3 provides an illustration of the operation of SOFT as described above. In the remainder of this section, we discuss our approach in detail. After a brief description of a strawman approach for utilizing symbolic execution in functional equivalence testing, we analyze improvements required to apply it to complex software such as OpenFlow agents. Finally, we discuss how we solve the second problem, namely collecting and comparing relevant outputs.

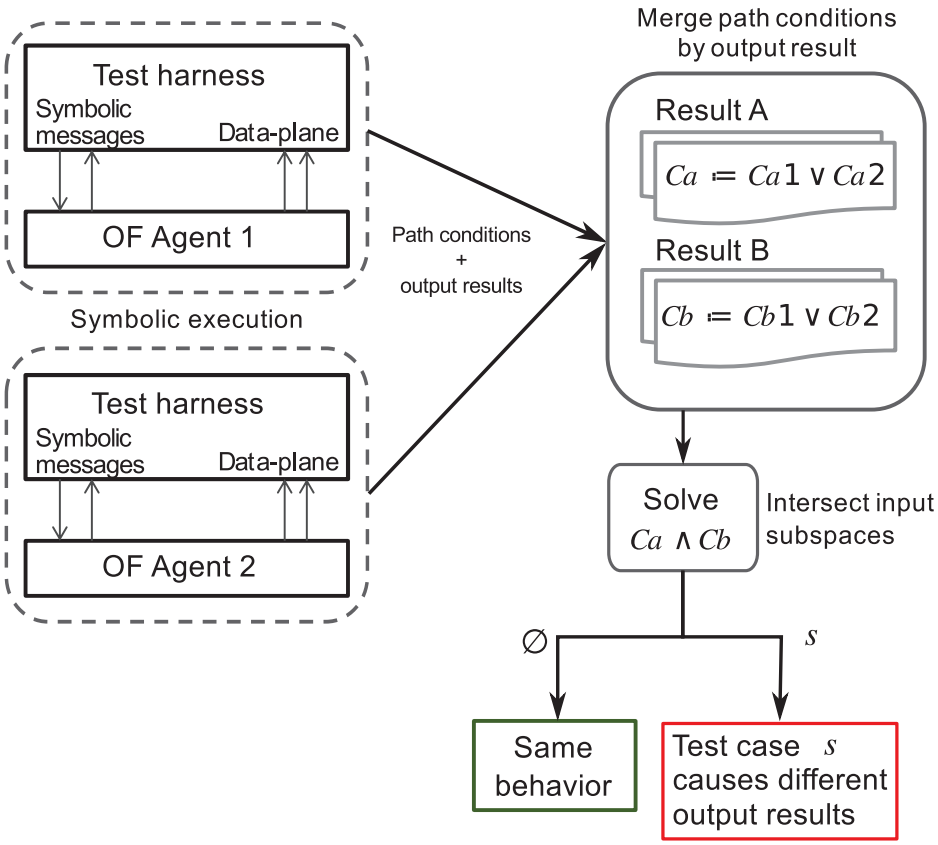


Figure 2.3: SOFT overview.

2.5.1 Automating Equivalence Testing

Our form of interoperability testing can be viewed as checking the functional equivalence of different OpenFlow agents at the interface level (*i.e.*, the OpenFlow API). To understand how we can use symbolic execution for this purpose, let us first consider a simpler problem.

A strawman approach. Assume we have two functions that implement the same algorithm differently and we want to test if they are indeed functionally equivalent. To do this, it is sufficient to symbolically execute both functions by passing identical symbolic inputs to both of them and checking whether they return the same value. If the results differ, the symbolic execution engine can construct a test case to exercise the problematic code path. In

essence, symbolic execution enables us to crosscheck the two functions' results through all possible execution paths. This simple approach is sound, *i.e.*, it identifies all cases where results differ, provided that symbolic execution can solve all constraints it encounters. It is also relatively straightforward to extend this approach to crosscheck console utility programs by running with the same symbolic environment and comparing the data printed to `stdout`, as shown in [24].

Challenges and approach. Scaling up this approach to our target system is not an easy task. An OpenFlow agent is a non-terminating, event-driven program that interacts intensely with its environment. In this case, the environment consists of the network data plane, other switch components (*e.g.*, flow tables) and the controller.

The first challenge this raises is that the input space is inherently infinite, thus making the problem of comparing OpenFlow agents over unbounded inputs intractable. Instead, to make our problem tractable, we must limit the length of any input sequence used for testing.

Second, crosschecking the results of different OpenFlow agents is challenging because there exists no notion of a "switch return value". Furthermore, there does not exist a universal `stdout` format that enables textual comparison unlike console utilities. Instead, we must collect a trace of switch *output results* that enables comparison using detailed information from both the OpenFlow and the data plane interfaces. In other words, we must for example, capture packets and OpenFlow messages emitted by the switch, and maintain a non-ambiguous representation of these events.

Finally, the approach above works by feeding both functions with the same symbolic input. In turn, this requires that both agents be locally available. However, we cannot assume that SOFT will operate on different Open-

Flow agents at the same time. Instead, we make a conscious design choice to decouple the symbolic execution phase from the crosschecking phase.

2.5.2 Creating Symbolic Inputs

An OpenFlow agent reacts to OpenFlow messages and data plane packets it receives. Therefore, sequences of such messages can be considered inputs to the agent. In this subsection, we only consider the control channel inputs (the messages sent by the controller) because our goal is to test a switch at the OpenFlow interface (and not the data plane interface).

2.5.2.1 Structuring Inputs

Feeding unstructured inputs is ineffective. As the input space containing sequences of arbitrary numbers of arbitrary messages is infinite, we need to enforce the maximum length of the sequence. A straightforward way to limit the input size would be to use N-byte symbolic inputs, with N bounded. Unfortunately, this approach quickly hits the scalability limits of exhaustive path exploration because these inputs do not contain any information that is of either syntactic or semantic value. As a result, symbolic execution must consider all possible ways in which these symbolic inputs can be interpreted (most of which represent invalid inputs anyway) to exhaust all paths. As an example, consider feeding an agent with the mentioned sequence of N symbolic bytes. Since there exist different types of control messages, some of which have variable lengths, this stream of N bytes can be parsed (depending on its content) as: one message of N bytes, or as any combination of two messages whose lengths add up to N, or as combinations of three messages, *etc.*

Moreover, some messages like `Flow Mod` and `Packet Out`, are variable

in length. This is because they both contain the actions field which is a container type for possible combinations of forwarding actions. The major issue arises as each individual action is itself variable in length. As such, we are again in the situation where symbolic execution is left to explore all possible combinations in which it can interpret N symbolic bytes as multiple action items. Although individual lengths must be multiple of 8 bytes to be valid, the combinatorial growth quickly becomes impractical.

Structuring the inputs improves scalability. We overcome the aforementioned problems by using a finite number of finite-size inputs. Most importantly, we construct inputs that adhere to valid format boundaries of OpenFlow control messages rather than leaving symbolic execution to guess the correct sizes. This means that we feed the agent with one symbolic control message at a time and pass the actual message length as a concrete value in the appropriate header field. In practice, we must also make the message type concrete before establishing a valid message length, as the latter is essentially determined by the former. This is not an issue, since every message must be identified by a valid code (at present about 20 codes exist, all described in the protocol specifications, *e.g.*, [11]). In a similar fashion, for messages that have variable length actions, we predetermine the number of action items and the relative lengths as concrete values.

2.5.2.2 Choosing the Size of Inputs

As we choose to limit the size of inputs, the immediate question we face is up to what input size is it practical to symbolically execute an OpenFlow agent, given today's technology? Indeed, it is known that the scalability of symbolic execution is limited by the path explosion problem: *i.e.*, the number of feasible paths can grow exponentially with the size of the inputs

and number of branches. On the other hand, to make testing meaningful, the chosen inputs need to provide satisfactory coverage of agent's code and functionality. In practice, we seek answers through empirical observations. The input size varies along two dimensions: (*i*) number of symbolic control messages, and (*ii*) number of symbolic bytes in each message.

Covering the input space of each message is generally feasible. We first explore to what extent the number of symbolic bytes in each message represents a hurdle to our approach. As we discuss below, we find that the overhead to exhaustively cover the input space of each message is generally acceptable, given the current protocol specifications. We already mentioned that the message length depends in the first place on the message type. It should also be clear that the processing code and especially the processing complexity varies across message types. For example, it is trivial to symbolically execute a message of type `Hello`, which contains no message body. On the other hand, the `Flow Mod` message, which drives modifications to the flow table, carries tens of data fields that need validation and ultimately determine what actions the switch will perform. Indeed, we observe through experimentation that the number of feasible paths varies significantly between different message types (two orders of magnitude between `Flow Mod` and `Packet Out` messages). Most importantly, symbolic execution runs to completion in all cases when testing with the reference OpenFlow switch implementation.[1]

Achieving good coverage requires just two symbolic messages. However, the question remains about how many symbolic control messages we should inject in practice. Again, the answer depends on what type of messages one considers. We find that for complex messages we can at most use

[1]Our experimental setup is introduced in Section 2.7.

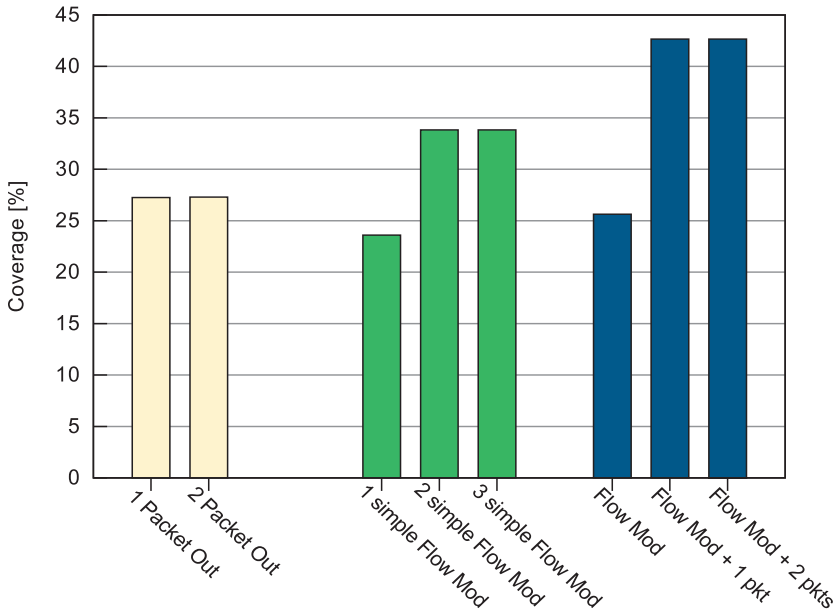


Figure 2.4: Reference switch code coverage as a function of the number of symbolic messages injected.

a sequence of three messages. This number may seem small, but it is worth noting that we do not need long message sequences for the type of testing we target. In fact, one symbolic message is already sufficient to cover all feasible code paths involved in message processing. With the subsequent message, we augment the coverage to include additional paths that depend on parts of switch state that are rendered symbolic as a result of running with the first symbolic message. Effectively, the second message enables us to explore the cross-interactions of message pairs. In addition, such interactions exist only for a small fraction of possible message type combinations. For example, two `Flow Mod` messages may affect the same part of the switch state; that is not true for `Echo Request` followed by `Flow Mod`. As such, the increase in instruction coverage due to the second message is a fraction of what the first message covers. A third message does not significantly improve coverage further as shown in Figure 2.4. Thus, careful consideration of inputs is key to successfully achieving our goals through symbolic execution.

2.5.2.3 Defining Relevant Input Sequences

Exploiting domain specific knowledge is essential to construct input sequences that target interesting uses of OpenFlow messages to further reduce the testing overhead. First, although the protocol specifications define about 20 messages, some of these are clearly more important than others. For example, the `Hello` and `Echo` messages are simple connection establishment and keep-alive messages, respectively. We focus on complex messages such as `Flow Mod`, `Packet Out`, `Set Config` that require validation and modify the state of an agent. We also note that because these messages are meant to affect different functional aspects of the agent, we find it unnecessarily time-consuming to check all pair-wise combinations of these messages. Section 2.7 details the actual sequences of messages we use for testing.

2.5.3 Collecting Output Results

So far we have shown how our approach uses exhaustive path exploration to obtain the input space partitions (or equivalence classes of inputs). However, we still need to know what end result each partition produces because only the results enable the comparison across different OpenFlow agents.

As we feed a symbolic message to an OpenFlow agent, its state might be updated. Additionally, there are two possible outcomes: (*i*) the agent outputs some data (*i.e.*, messages back to the controller or data plane packets), or (*ii*) the agent does not produce externally observable data. In this work we treat only data explicitly returned by an agent (OpenFlow messages and data plane packets) as an output. Instead of directly fetching the internal state, we use additional packets and messages to infer the impact of the state on agent's behavior.

Capturing output data. To collect the outputs, we make use of the

OpenFlow and data plane interfaces to capture data. Specifically, we log all OpenFlow messages and packets emitted by the agent. Note that the entire analysis runs in software (the output data may even contain symbolic inputs); therefore, with data plane interface we simply mean the socket API (or equivalent) that the agent uses to send packets.

Using concrete packets for probing state. Regardless of whether the agent does or does not output data, we cannot immediately determine if the symbolic message caused any internal state change (*e.g.*, the `Flow Mod` message installs a new rule in the flow table). Differences in the internal state do not necessarily result in differences in observed behavior. Moreover, we want to avoid directly fetching an agent's internal state as this would add a dependency on the specific implementation. As a solution, following any potentially state changing symbolic message, we inject a concrete packet through the data plane interface as a simple state probe. The effect of this probe is that it enables symbolic execution to exercise the code that matches incoming packets and the code that applies the forwarding actions. The probe packet is then either forwarded (to a port or controller), in which case we log it, or it is dropped, in which case we log an empty probe response.

Normalizing results. Rather than saving the logs verbatim, we normalize the output results to remove certain data from the results for which spurious differences are expected. For example, the buffer identifiers used by different agents may differ and such a difference should not be considered an inconsistency.

2.5.4 Finding Inconsistencies

In this phase, we seek to find inconsistencies between two OpenFlow agents denoted A and B.

With respect to agent A, we denote with PC_A the set of path conditions (outcome of symbolic execution). For each $pc \in PC_A$, let $\text{res}_A(pc)$ be the normalized output result when executing the path represented by pc. We denote the set of distinct output results as RES_A.

Grouping paths by output results. Our first step is to group all different path conditions that produce the same output result. Formally, $\forall r \in RES_A$ we set $C_A(r) = \bigvee\{pc \mid pc \in PC_A;\ \text{res}_A(pc) = r\}$ to be the disjunction of all path conditions that share the same output result. PC_B, RES_B, and C_B are similarly defined.

Intersecting input subspaces. In our second and last step, for each pair of different outputs of agents A and B, we check if there exists at least one common input that leads to these inconsistent outputs. For each pair (i, j) of results $i \in RES_A$, $j \in RES_B$ such that $i \neq j$, we query a satisfiability solver (STP [36]) to obtain an example test case that satisfies the condition $C_A(i) \bigwedge C_B(j)$. If the solver can satisfy this conjunction, then we have an inconsistency.

Discussion. It is easy to note that an upper bound of the number of queries to the solver for our approach is $|RES_A| \cdot |RES_B|$. In addition, note that our approach produces only one inconsistency example per pair of different output results. In other words, we do not provide one example for each path that produces the inconsistency. If this is desired, one can omit grouping all paths that share the same output. However, doing so has an inherent overhead cost because it increases the number of STP queries. Instead, our approach amortizes the start-up costs of a multitude of solver invocations by using fewer larger queries and enables the solver to apply built-in optimizations to handle such larger queries.

As with any bug finding tool, it is important to know whether our ap-

proach may report false positives/negatives. We observe that SOFT does not produce false positives: each identified inconsistency is an evidence of divergent behavior. Note that this does not necessarily mean that one agent does something in violation to the specifications. According to our previous definition, an inconsistency is reported if the tested agents perform different actions when exposed to the same input. However, the tool might have false negatives for two reasons. The first is that our path coverage may not be complete. For instance, symbolic execution might not cover all feasible paths due to path explosion. The second is that all agent implementations under test might contain the same bug, and therefore produce the same output.

2.6 Implementation Details

We built our SOFT prototype on top of the Cloud9 [23] symbolic execution engine. SOFT consists of three major components: (*i*) a test harness, which drives the testing of OpenFlow agents, (*ii*) a grouping tool to group path conditions that share output results, and (*iii*) a tool for finding inconsistencies.

2.6.1 Test Harness and Cloud9

To provide the necessary execution environment for the OpenFlow agent, we build a test harness that emulates both a remote controller and the underlying network. The emulated controller is capable of injecting symbolic inputs.

As a symbolic execution engine, Cloud9 can symbolically execute only a single binary. We therefore create a test "driver" by linking the OpenFlow agent and our test harness controller together. Upon startup, the test driver forks into two processes, one of which runs the OpenFlow agent while the second runs the test harness itself. The two processes are connected via

standard UNIX sockets. Upon startup, the OpenFlow agent connects to the test harness. After the connection setup and exchange of the initial `Hello` messages, the test harness injects a sequence of several symbolic OpenFlow messages and/or probes, one at a time (we discuss the input sequences in more details in Section 2.7). Upon confirming that the switch processed all messages and probes, we kill the execution.

To use Cloud9 for our goal, we had to improve its environment model. Cloud9 provides a symbolic model of the POSIX environment. Such a model, most importantly, allows us to efficiently use the socket API without accessing the entire networking stack. As a result, all symbolic variables remain symbolic after being transferred as data in a packet. However, such a model needs to provide all functions used by the tested application. Notably, we needed to implement the RAW socket API which was missing in Cloud9 but is used by the OpenFlow agents in our tests. Moreover, we replace or simplify some library functions as described next.

We assume that the agents correctly use network versus host byte ordering, and we change functions `ntoh` and `hton` to simply return their argument unchanged. This simplifies constraints by removing double-shuffling (first when the test harness creates a message, second when the OpenFlow agent parses the message). We also simplify checksum and hash functions to return constants or identities, because they cannot be reversed or it is computationally very expensive to do so (this is a well-known issue in using a constraint solver). The aforementioned modifications reduce complexity and improve symbolic execution efficiency.

Finally, the symbolic execution engine may use several search strategies that prioritize different goals while exploring the program. We choose to use the default Cloud9 strategy that is an interleaving of a random path

choice and a strategy that aims to improve coverage. However, the choice of the search strategy has small impact on our tool. By controlling the inputs we tend to exhaustively cover all possible execution paths, which in turn diminishes the impact of choosing a particular search strategy. Moreover, SOFT is capable of working with traces that are only partially covering agents' code.

2.6.2 Tools

Apart from the test harness, we provide two tools for manipulating Cloud9 results. Both of these tools are written in C++ and heavily reuse existing Cloud9 code for reading, writing and manipulating path conditions. The tools contain less than 200 lines of new code in total.

The group tool reads multiple files (results of Cloud9 execution), identifies different output results and groups the path conditions by result. To improve performance of further constraint parsing, we group path conditions by building a balanced binary tree minimizing the depth of nested expressions. The inconsistency finder tool expects two directories holding grouped results as its arguments. The tool iterates over all combinations of different results and queries the STP solver to check for inconsistencies. If there is an inconsistency (the condition is satisfiable), the STP solver provides an example set of variables that satisfy the condition. This is a test case that can be used to understand and trace the root cause of the inconsistency and verify if a behavior is erroneous.

2.7 Evaluation

We evaluate SOFT using two publicly available OpenFlow agents compatible with the specifications in version 1.0. The first one is a reference OpenFlow

Test	Description
Packet Out	A single `Packet Out` message containing a symbolic action and a symbolic output action.
Stats Request	A single symbolic `Stats Req`. It covers all possible statistics requests.
Set Config	A symbolic `Set Config` message followed by a probing TCP packet.
FlowMod	A symbolic `Flow Mod` with 1 symbolic action and a symbolic output action followed by a probing TCP packet.
Eth FlowMod	Symbolic `Flow Mod` with 1 symbolic action and a symbolic output action. Fields not related to Ethernet are concretized. The message is followed by a probing Ethernet packet.
CS FlowMods	2 `Flow Mod`. The first one is concrete, the second is symbolic.
Concrete	4 concrete 8-byte messages. These are the messages that do not have variable fields.
Short Symb	A 10-byte symbolic message. Only the OpenFlow version field is concrete.

Table 2.1: Tests used in the evaluation.

switch implementation written in C released with version 1.0 of the specifications. Its main purpose is to clarify the specifications and present available features. Although the reference implementation is not designed for high performance, it is expected to be correct as others will build upon and test against it. We are referring to this version as Reference Switch (55K lines of code). The second is Open vSwitch 1.0.0 [10] (80K lines of code). It is a production quality virtual switch written in C and used as a base for several commercial switches.[2] OpenFlow is just one the supported protocols. We also created a third OpenFlow agent by modifying the Reference Switch and introducing different corner case behaviors (Modified Switch). This way we can tell how efficiently SOFT finds the injected differences and which of them remain unnoticed.

To evaluate SOFT we use the set of tests summarized in Table 2.1. We run our experiments using a machine with Linux 3.2.0 x86_64 that has 128

[2]For example, in Pica8 products: `http://www.pica8.org/`.

GB of RAM and a clock speed of 2.4 GHz. Our implementation does not use multiple cores for a single experiment.

2.7.1 Can SOFT Identify Inconsistencies?

In this section, we report and analyze the inconsistencies SOFT detects. We apply a set of tests to all three OpenFlow agents and compare Reference Switch with both Modified Switch and Open vSwitch.

2.7.1.1 Modified Switch vs. Reference Switch

First, we look for differences between Reference Switch and Modified Switch. Two team members who did not take part in the tool's implementation and test preparation were designated to introduce a few modifications to the Reference Switch. The modifications were meant to affect the externally visible behavior of the OpenFlow agent. Having purposefully injected changes, we set out to check how many can be detected by SOFT.

SOFT is able to correctly pinpoint 5 out of 7 injected modifications. We further investigate the cases in which SOFT failed to flag the effect of the differences. It turns out that one of them concerns the `Hello` message received while establishing a connection to the controller. SOFT does not recognize this problem because it establishes a correct connection first and then performs the tests. The second missed modification manifests itself only when a rule is deleted because of a timeout. This occurs because the symbolic execution engine is not able to trigger timers. As part of our future work, we plan to extend our approach to deal with time, *e.g.*, similarly to MODIST [82].

2.7.1.2 Open vSwitch vs. Reference Switch

Knowing that SOFT is capable of finding inconsistencies, we compare the Reference Switch with Open vSwitch to verify how useful SOFT is when applied to a production quality OpenFlow agent. The list of differences between the two major software agents contains a few significant ones. In the following, we present the observed inconsistencies and analyze their root causes.

Packet dropped when action is invalid. This case describes a `Packet Out` message containing a packet that is silently dropped by Open vSwitch while the Reference Switch forwards it. The inconsistency appears when the `Packet Out` control message satisfies the following conditions: (*i*) it contains the packet that the agent should forward and (*ii*) one of the actions is setting the value of VLAN or IP Type of Service field. Further investigation leads us to the conclusion that Open vSwitch validates whether a new VLAN value set by the action fits in 12 bits and similarly whether the last two bits of the TOS value are equal to 0. When an action specified in the message does not pass this strict validation, Open vSwitch silently ignores the whole message. Additional tests with `Flow Mod` messages reveal a similar issue. These tests also show that the `vlan_pcp` field undergoes additional validation in Open vSwitch. Reference Switch does not validate values of the aforementioned fields, but it automatically modifies them to fit the expected format.

The specifications do not state that the OpenFlow agent should perform such a precise validation of any of the mentioned fields. Therefore, both implementations might be considered correct. However, such a difference in behavior might cause unexpected packet drops if the controller developers test their applications with switches that are different from those deployed in the network.

Forwarding a packet to an invalid port. Here we describe a case in which the tested OpenFlow agents return error messages concerning incorrect output ports in an inconsistent fashion. According to the specifications, the agent has to return an error message if the output port will never be valid. However, if the port may become valid in the future, the message might either be rejected with an error, or the agent may drop packets intended for this port while it is not valid. The differences in interpretation when the port will be invalid forever lead to a few differences between OpenFlow agents. First, when the ingress port in the match is equal to the output port, the Reference Switch returns an error, as no packets will ever be forwarded to this port.[3] Open vSwitch accepts such a rule and drops all matching packets. On the other hand, Open vSwitch immediately returns an error when the action defines an output port greater than a configurable maximum value. Reference Switch does not validate ports this way.

Thus, if the controller application relies on error messages received, it may misbehave when deployed with a different agent than it was tested with. If the agent used in testing considered a port valid but the other agent did not, the controller would fail to install rules it was designed to install. The opposite situation is equally unsafe. The rule installation that used to return an error succeeds, but all matching packets get dropped. As a result, some packets will not to be sent to the controller, although they were expected to be. Moreover, such a rule may cover another, lower priority one.

Lack of error messages. We have already presented a few cases when one of the agents silently drops the incorrect message without returning an error. SOFT detects another instance of such a problem in the Reference Switch while testing with `Packet Out` and `Flow Mod` messages. When the

[3] A special `OFPP_IN_PORT` port must be explicitly used to forward packets back to the port they came from [11].

`buffer_id` field refers to a non-existent buffer, the Reference Switch handles the message but does not apply actions to any packet and does not report any error. Open vSwitch replies with an error message, but installs the flow as well. We analyzed the Reference Switch source code and discovered that although the error is returned by the message handler, it is not propagated further as an OpenFlow message.

OpenFlow agent terminates with an error. There are three independent cases when the Reference Switch crashes. First, when the OpenFlow agent receives a `Packet Out` message with output port set to `OFPP_CTRL`. This may be a rare case (*e.g.*, when the developer demands such behavior) but it is not forbidden by the specifications. Second, when the agent executes an action setting the `vlan` field in a `Packet Out` message the same error appears and the agent crashes. Finally, when the agent receives a queue configuration request for port number 0, it encounters a memory error. All the aforementioned problems are not only inconsistencies, but also major reliability problems in the OpenFlow agent.

Different order of message validation. In this case, the order in which message fields should be validated is not made explicit in the specifications. This vagueness results in externally visible differences in agents' behavior. The same incorrect message may induce two different error messages, or an error message and a lack of response in case of the mentioned problem. We encountered such a situation for a `Packet Out` message with an incorrect buffer id and output port.

Statistics requests silently ignored. The Reference Switch silently ignores requests for statistics to which it is not able to respond. This behavior is a specific case of the "Lack of error messages" problem. Even though the handler returns an error it is not converted to an OpenFlow message. The

problem was detected because Open vSwitch sends an error in response to an invalid or unknown request.

Missing features. SOFT is able to detect features that are missing in one OpenFlow agent, but are present in the other. We were able to automatically infer that Open vSwitch does not support emergency flow entries that are defined in the specifications. Secondly, Reference Switch being purely an OpenFlow switch, does not support the traditional forwarding paths (`OFPP_NORMAL`).

2.7.2 What is the Overhead of Using SOFT?

In this section, we present the performance evaluation of the two key stages of SOFT's execution.

Symbolic execution. In the first stage, the OpenFlow agent is symbolically executed with an input sequence and SOFT gathers path constraints and corresponding outputs. For all three OpenFlow agents we report the running time, as well as the number and size (number of boolean operations in a path condition) of paths (equivalence classes of inputs) in Table 2.2. These metrics are strongly variable and depend not only on the input length but also on the message type. Moreover, adding a second message or a probe packet significantly increases complexity by orders of magnitude. Additionally, Open vSwitch–the most complex of the tested agents–is noticeably more challenging for symbolic execution (we note that it is possible to use even partial results of symbolic execution to look for inconsistencies). As a result of multiple additional validations, the test input space for Open vSwitch is partitioned into 3-15 times more subspaces than for the Reference Switch.

Subspaces intersections. We distinguish between two sub-stages of the second stage: (*i*) grouping input subspaces by the same output, (*ii*) inter-

Test	Message count	Reference Switch				Modified Switch				Open vSwitch			
		CPU time	Path count	Constraints avg size	Constraints max size	CPU time	Path count	Constraints avg size	Constraints max size	CPU time	Path count	Constraints avg size	Constraints max size
Packet Out	1	14s	49	71.57	96	24s	117	74.09	91	44s	241	63.48	79
Stats Request	1	44s	218	53.10	65	46s	218	53.10	65	186s	136	52.63	67
Set Config	2	446s	207	76.89	112	451s	207	76.89	112	569s	207	80.97	116
Eth FlowMod	2	40m	7680	101.12	132	83m	14280	106.62	136	198m	27682	98.56	127
FlowMod	2	373m	87828	109.77	161	>140h	>356753	123.09	164	60h	181620	123.28	159
CS FlowMods	2	69m	29179	96.63	136	121m	51419	99.74	139	36h	462488	115.22	173
Concrete	4	6s	1	0	0	6s	1	0	0	8s	1	0	0
Short Symb	1	50s	31	27.9	69	50s	31	27.9	69	12s	14	27.5	50

Table 2.2: Symbolic execution statistics for selected tests for all 3 OpenFlow agents. We report time, number of explored paths (input equivalence classes) and constraint size (average and maximum size).

Test	Grouping results				Inconsistency checking	
	Reference Switch		Open vSwitch			
	time	#res	time	#res	time	#
Packet Out	0.038s	6	0.090s	10	26s	14
Stats Request	0.116s	8	0.061s	9	10s	7
Set Config	0.141s	69	0.43s	69	236s	0
Eth FlowMod	8s	12	23s	31	23m	58
CS FlowMods	79s	4	344m	6	>28h	≥8
Short Symb	0.039s	9	0.01s	7	6s	4

Table 2.3: Time needed to find overlapping input subspaces and number of created test cases. Each test case represents one intersection of overlapping input subspaces. Additionally, time needed to group constraints by the output and a number of distinct outputs for Reference Switch and Open vSwitch.

secting subspaces corresponding to potential inconsistencies.

For the first sub-stage we report the time required to group and the number of distinct outputs. As presented in Table 2.3, this part requires orders of magnitude less time than symbolic execution. Grouping constraints dramatically reduces the number of expressions that need to be checked for satisfiability, as there are only up to 30 distinct outputs (a 1-5 orders of magnitude reduction compared to the initial number of equivalence classes).

The search for overlapping subspaces depends on the complexity of constraints and usually finishes within a couple of minutes. There is one exceptional case in which the STP solver is unable to solve the merged constraints in one day. In the future we plan to investigate grouping constraints into smaller groups for such cases.

The achieved results in finding inconsistencies confirm our expectations. Usually one difference manifests itself multiple times and affects many subspaces of inputs. In the extreme example, although there are 58 reported inconsistencies, manual analysis reveals only 6 distinct root causes of differences.

Test	Reference Switch		Open vSwitch	
	Inst.(%)	Branch(%)	Inst.(%)	Branch(%)
No Message	12.21	8.27	19.03	13.34
Packet Out	26.23	19.31	25.68	17.28
Stats Request	30.27	24.15	24.31	16.75
Set Config	26.23	19.31	23.98	16.16
Eth FlowMod	41.74	34.65	38.15	25.49
FlowMod	42.65	34.25	38.24	26.27
Concrete	17.13	11.42	20.16	13.62
Short Symb	19.92	13.39	21.60	14.34

Table 2.4: Instruction and branch coverage for selected tests for Reference Switch and Open vSwitch.

2.7.3 How Relevant is Input Sequence Selection?

To quantify the relevance of chosen tests, we measure the instruction and branch coverage provided by Cloud9. The instruction/branch reached at least once in the execution is considered covered, regardless of its arguments. We consider only the sections of OpenFlow agent's code relevant to OpenFlow processing. The initialization that is repeated for each test covers 12% of instructions and 8% of branches. The test specific results, shown in Table 2.4, are spread between 20 and 40%. To verify that the low reported coverage is a result of the fact that each test targets a few specific message handlers, we manually analyze cumulative coverage of all tests. We observe that SOFT covers approximately 75% of the code and that the remaining instructions belong mostly to code that is not accessible in standard execution (*e.g.*, command line configuration, dead code, cleanup functions, logging functions).

The importance of concretizing inputs. Due to time and memory constraints it is often convenient to concretize selected fields in the message. We evaluate the benefits and drawbacks of using the domain knowledge to reduce the input space. As a baseline, we choose a test where a single symbolic

Test	Time	Paths	Coverage
Fully Symbolic	31h	226224	42.93%
Concrete Match	12m	2634	40.60%
Concrete Action	193m	30396	37.32%
Concrete Probe	48m	9216	41.6%
Symbolic Probe	172m	33168	43.9%

Table 2.5: Effects of concretizing on execution time, generated paths and instruction coverage.

`Flow Mod` message containing 2 symbolic actions and 2 symbolic output actions is followed by a TCP probe packet. We then compare the results of: (*i*) the baseline, (*ii*) a version of the baseline with a concrete match (wildcard), and (*iii*) a version of the baseline with a single concrete action instead of 4 symbolic ones. All values are summarized in the upper part of Table 2.5. While the drop in the coverage percentage is only 2-5% in comparison to the baseline test, the difference in time and path count is noticeable. Specifically, the tests finish 10 to 50 times quicker, while generating 1 to 2 orders of magnitude less paths.

To verify how much coverage we lose by not using symbolic probes, we create a separate test. This test first installs a partially symbolic `Flow Mod` that applies actions to Ethernet packets. It then sends a short probe packet that is concrete or symbolic depending on the test version. Results in the lower part of Table 2.5 show that a symbolic probe adds just 2% to the coverage. The cost is 3.5 times longer running time and 3.5 times more paths.

To summarize, concretizing parts of the inputs significantly reduces the time needed to conduct the test at the cost of leaving small portion of additional instructions uncovered. Therefore, it is possible to use the concretized inputs to conduct regular tests more often. When combined with careful choice of concrete fields, the coverage is marginally affected. The fully sym-

bolic messages can be used just for the final checks before a major release when the best coverage possible is required, and testing time is less of an issue.

2.8 Summary

By combining symbolic execution with a novel use of constraint solvers, SOFT automatically determines functional differences between software agents running on switches. Applying the tool to two existing software switches reveals several inconsistencies. While some of them are a result of a buggy implementation, there are cases where agents diverge because of differences in interpreting ambiguous portions of the specification.

Finally, although the work presented in this chapter is centered around the specific details of OpenFlow, the approach is more general and can be applied to other router software and heterogeneous networked systems.

Chapter 3

Detecting Performance Differences

This chapter presents a methodology to detect performance related differences between switches. These issues manifest themselves only in the running devices and cannot be uncovered by methods that are relying on static analysis of switch software such as the one presented in Chapter 2. The methodology is based on systematic exploration of control plane message sequences sent to the devices, while monitoring the data plane behavior. The main focus of this chapter is on rule update performance, while measuring maximum data plane packet forwarding rates stays outside of its scope.

Applying the methodology to six hardware switches allows us to detect two general inconsistency types. In this chapter, we first expose behaviors that do not follow the specification, and if unnoticed, may lead to unsafe networks. Then, we present a detailed overview of rule update performance characteristics depending on various parameters. Each non-obvious finding is expanded with further experimental investigation, a hypothesis and, if possible, switch vendor comments.

3.1 Measurement Methodology

This section describes the methodology we follow to design the benchmarks that assess control and data plane update performance of switches under test.

3.1.1 Tools and Experimental Setup

In this study we focus on two metrics describing switch behavior: flow table rule update rate and correspondence between control plane and data plane views. The second metric is quantified by the time gap between when the switch confirms a rule modification and when the modified rule starts affecting packets. We designed a general methodology that allows for systematic exploration of switch behaviors under various conditions. At the beginning of each experiment, we prepopulate the switch flow table with R rules. Unless otherwise specified, the rules are non-overlapping and have the default priority. Each rule matches a flow based on a pair of IP source-destination addresses, and forwards packets to switch port α. For clarity, we identify flows using contiguous integer numbers starting from $-R+1$. According to this notation, the prepopulated rules match flows in the range $-R+1$ to 0, inclusive.

After initializing the switch's hardware flow table, we perform flow table updates and measure their behaviors. In particular, we send B batches of rule updates, each batch consisting of: B_D rule deletions, B_M rule modifications and B_A rule insertions. Each batch is followed by a barrier request. In the default setup, we set $B_D = B_A = 1$ and $B_M = 0$. If B_D is greater than 0, batch i deletes rules matching flows with numbers between $-R+1+(i-1)*B_D$ and $-R+i*B_D$. If B_A is greater than 0, batch i installs rules that match flows with numbers in range between $(i-1)*B_A+1$ and $i*B_A$ and forwards packets to port α. As a result, each batch removes the oldest rules. Note that the total number of rules in the table remains stable during most experiments (in contrast to previous work such as [59] and [72] that measure only the time needed to fill an empty table).

To measure data plane state, in some experiments, we inject and cap-

ture data plane traffic. We send packets that belong to flows F_{start} to F_{end} (inclusive) at a rate of about 300 packets per flow per second.

In our study, we have explored a wide range of possible parameters for our methodology. For brevity, in the next sections, we highlight results where we instantiate the methodology with specific parameters that led to interesting observations. In the experiment descriptions we call the setup described above with $B_D = B_A = 1$, $B_M = 0$ and all rules with equal priority as a general experimental setup. Finally, unless an experiment shows variance greater than 5% across runs, we repeat it three times and report the average. Because the results have a small deviation across runs, unless otherwise specified, we do not show confidence intervals.

Measurement tool: Based on our initial investigation, as well as previously reported results [44], we identify three main requirements for a measurement tool: (*i*) flexibility, (*ii*) portability, and (*iii*) sufficient precision. First, since the switches we test are often in remote locations with limited physical access, the measuring tool cannot use customized hardware (*e.g.*, FPGAs). Moreover, our previous experience suggests that switches behave unexpectedly, and thus we need to tailor the experiments to locate and dissect problems. Finally, as the tested switches can modify at most a few thousands of rules per second, we assume that a millisecond measurement precision is sufficient.

To achieve the aforementioned goals, we built a tool that consists of three major components that correspond to the three benchmarking phases: input generation, measurement and data analysis (Figure 3.1).

First, an input generator creates control plane rule modification lists and data plane packet traces used for the measurements. Unless otherwise specified, the forwarding rules used for the experiments match traffic based on IP source/destination pairs and forward packets to a single switch port. More-

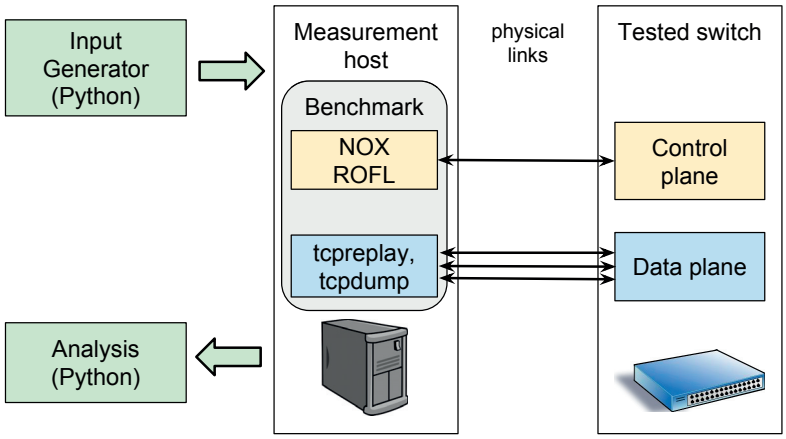


Figure 3.1: Overview of our measurement tools and testbed setup.

over, we notice that some switches can optimize rule updates affecting the same rule; we therefore make sure that modifications affect different rules. To ensure this, by default, we use consecutive IPs for matches. Furthermore, we cross-check our results using random matches and update patterns.

We refer to the control plane measurement engine as the controller as it emulates the behavior of an OpenFlow controller. We implement it using NOX [6] and ROFL [13] libraries that can issue rule updates at a much higher rate than what the hardware switches can handle.[1] The engine records time of various interactions with the switch (*e.g.*, flow modification sent, barrier reply received) and saves all its outputs into files. We additionally record all control plane traffic using tcpdump. We rely on existing tcpreplay and tcpdump tools to both send packets based on a pcap file and record them. To remove time synchronization issues, we follow a simple testbed setup with the switch connected to a single host on multiple interfaces — the host handles the control plane as well as generates and receives traffic for the data plane. Note that we do not need to fully saturate the switch data plane, and thus a

[1]Our benchmark with software OpenVSwitch handles ∼42000 updates/s.

conventional host is capable of handling all of these tasks at the same time.

Finally, a modular analysis engine reads the output files and computes the metrics of interest. Modularity means that we can add a new module to analyze a different aspect of the measured data. We implement the analysis engine as a collection of Python modules.

Switches under test: We benchmark three ASIC-based switches capable of OpenFlow 1.0 and two ASIC-based switches capable of OpenFlow 1.3 support: HP ProCurve 5406zl with K.15.10.0009 firmware, Pica8 P-3290 with PicOS 2.0.4, Dell PowerConnect 8132F with beta[2] OpenFlow support, Switch X and Switch Y. They use ProVision, Broadcom Firebolt, Broadcom Trident+, Switch X and Switch Y ASICs, respectively. We additionally compare how Switch X behaves with two firmware versions: V1 and V2. We anonymize two of the switches since we did not get a permission to use their names from their respective vendors. These switches have two types of forwarding tables: software and hardware. While hardware flow table sizes (about 1500, 2000, 750, 4500, and 2000 rules, respectively) and levels of OpenFlow support vary, we make sure that all test rules ultimately end up in the hardware tables. Moreover, some switches implement a combined mode where packet forwarding is done by both hardware and software, but this imposes high load on the switch's CPU and provides lower forwarding performance. Thus, we avoid studying this operating mode. Further, as mentioned before, analyzing the data plane forwarding performance is out of scope of this work. We also benchmark NoviSwitch 1132 — a network-processor based, OpenFlow 1.3 switch running firmware version 300.0.1.[3] Each of its 64 flow tables fits over 4000 rules. We caution that the results for

[2]There are plans to optimize and productize this software.

[3]We repeated our tests with firmware 300.0.5 but observed similar results. We briefly describe bigger differences when compared to earlier firmware versions in Section 3.2.3.

this switch may not directly compare to those of the other measured devices due to the different switch architecture. In particular, our methodology correctly characterizes the update rates of flow tables but does not establish a relation between flow table occupancy and maximum forwarding speed, for which ASICs and network processor might exhibit different behaviors.

Finally, since switches we test are located in different institutions, there are small differences between the testing machines and the network performance. However, the set-ups are comparable. A testing computer is always a server class machine and the network RTT varies between 0.1 and 0.5 ms.

3.2 Results: Flow Table Consistency

While the only view the controller has of the switch is through the control plane, the real traffic forwarding happens in the data plane. In this section we present the results of experiments where we monitor rule updates in the control plane and at the same time send traffic to exercise the updated rules. The unexpected behavior we report in this section may have negative implications for network security and controller correctness.

3.2.1 Synchronicity of Control and Data Planes

Many solutions essential for correct and reliable OpenFlow deployments (*e.g.*, [60, 70]) rely on knowing when the switch applied a given command *in the data plane* . The natural method to get such information is the barrier message.Therefore, it is crucial that this message works correctly. However, as authors of [72] already hinted, the state of the data plane may be different than the one advertised by the control plane. Thus we set out to measure how these two views correspond to each other at a fine granularity.

We use the default setup extended with one match-all low priority rule

Switch	Data plane
Switch X, firmware V1	falls behind indefinitely. Up to 4 minutes in our experiments.
Switch X, firmware V2	in sync with control plane
HP 5406zl	often falls behind up to 250 ms. Indefinitely in corner cases (up to 22 s in our tests).
Pica8 P-3290	reorders + behind up to 400 ms
Dell 8132F	in sync with control plane
Switch Y	in sync with control plane
NoviSwitch 1132	in sync with control plane

Table 3.1: Data plane synchronicity key findings summary.

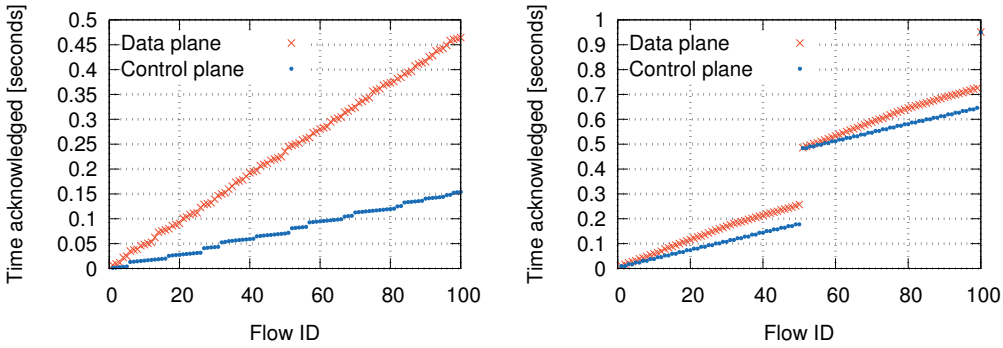
that drops all packets[4] and we inject data plane flows number F_{start} to F_{end}. For each update batch i we measure the time when the controller receives a barrier reply for this batch and when the first packet of flow i reaches the destination.

Figure 3.2 shows the results for $R = 300$, $B = 300$, $F_{start} = 1$ and $F_{end} = 100$. There are three types of behavior that we observe: desynchronizing data and control plane states, reordering rules despite barriers and correct implementation of the specification.

Switch X: The data plane configuration of *Switch X is slowly falling behind* the control plane acknowledgments — packets start reaching the destination long after the switch confirms the rule installation with a barrier reply. The divergence increases linearly and, in this experiment reaches 300 ms after only 100 rules. The second observation is that *Switch X installs rules in the order of their control plane arrival.* After reporting the problem of desynchronized data and control plane views to the switch vendor, we received a new firmware version that fixed observed issues to some extent. We report the improvements in Section 3.2.3.

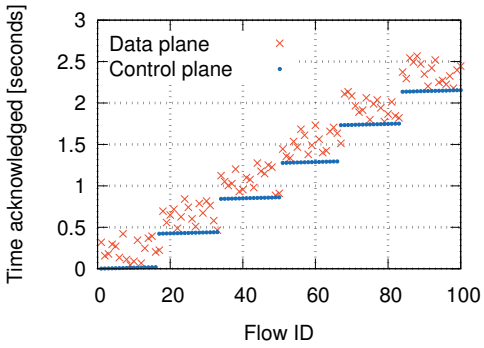
HP 5406zl: Similarly to Switch X, the data plane configuration of *HP*

[4]We need to use such a rule to prevent flooding the control channel with the `Packet In` messages caused by data plane probes or flooding the probes to all ports.

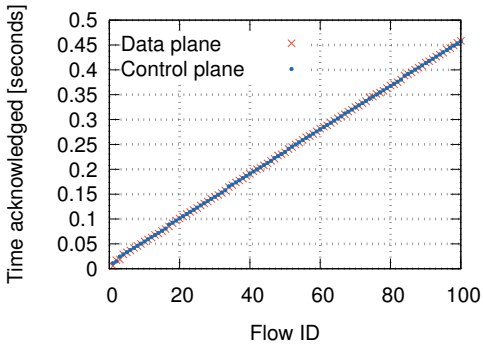


(a) Switch X, firmware version V1

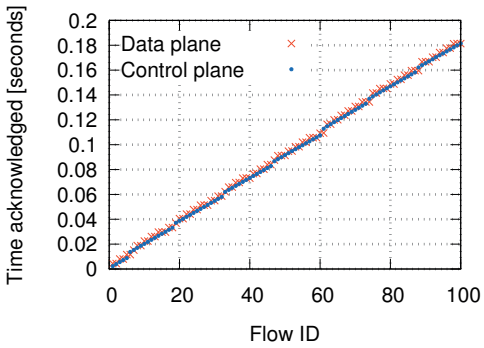
(b) HP 5406zl



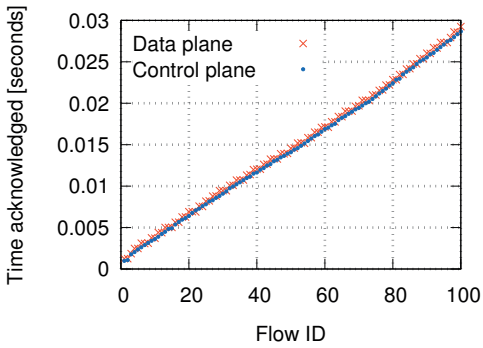
(c) Pica8 P-3290



(d) Dell 8132F



(e) Switch Y



(f) NoviSwitch 1132

Figure 3.2: Control plane confirmation times and data plane probe results for the same flows. Switch data plane installation time may fall behind the control plane acknowledgments and may be even reordered.

5406zl is slowly falling behind the control plane acknowledgments. However, unlike for Switch X, after about 50 batches, which corresponds to 100 rule updates (we observed that adding or deleting a rule counts as one update, and modifying an existing rule as two), the *switch stops responding with barrier replies for 300 ms*, which allows the flow tables to catch up. After this time the process of diverging starts again. In this experiment the divergence reaches up to 82 ms, but *can be as high as 250 ms* depending on the number of rules in the flow table. Moreover, the frequency and the duration of this period does not depend on the rate at which the controller sends updates, as long as there is at least one update every 300 ms. The final observation is that *HP 5406zl installs rules in the order of their control plane arrival.*

Pica8 P-3290: Similarly to HP 5406zl, Pica8 P-3290 stops responding to barriers in regular intervals. However, unlike HP 5406zl and Switch X, Pica8 P-3290 is either processing control plane (handling update commands and responding to barriers), or installing rules in TCAM and never does both at the same time. Moreover, *despite the barriers, the rules are not installed in hardware in the order of arrival.* The delay between data and control plane reaches *up to 400 ms* in this experiment. When all remaining rules get pushed into hardware, the switch starts accepting new commands in the control plane again. We confirmed with a vendor that because the synchronization between the software and hardware table is expensive, it is performed in batches and the order of updates in a batch is not guaranteed. When the switch pushes updates to hardware, its CPU is busy and it stops dealing with the control plane.[5]

Dell 8132F, Switch Y and NoviSwitch 1132: *All three switches make sure that no control plane confirmation is issued before a rule becomes*

[5]The vendor claims that this limitation occurs only in firmware prior to PicOS 2.2.

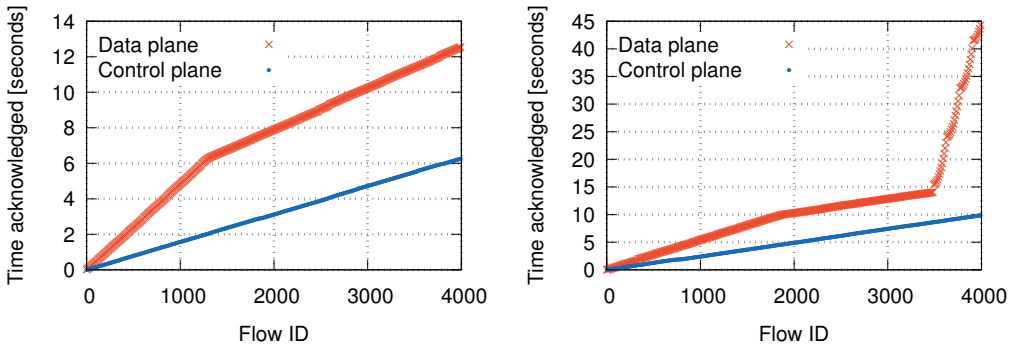
active in hardware. In this experiment we do not see any periods of idleness as the switch pushes rules to hardware all the time and waits for completion if necessary. Additionally, because NoviSwitch 1132 is very fast, we increased the frequency of sending data plane packets in order to guarantee required measurement precision.

Summary**: To reduce the cost of placing rules in a hardware flow table, vendors allow for different types (e.g., falling behind or reordering) and amounts (up to 400 ms in this short experiment) of temporary divergence between the hardware and software flow tables. Therefore, the barrier command does not guarantee flow installation.* ***Ignoring this problem leads to an incorrect network state that may drop packets, or even worse, send them to an undesired destination!

3.2.2 Variability in Control and Data Plane Behavior

The short experiment described in the previous section reveals three approaches to data and control plane synchronization. In this section we report more detailed unexpected switch behavior types observed when varying parameters in that experiment. The overall setup stays the same, but we modify the number of rules in the flow tables, length of the experiments and range of monitored rules.

Switch X: The short experiment revealed that facing a constant high update rate Switch X never gives the data plane state a chance to synchronize with control plane acknowledgments. In this extended experiment we issue 4000 batches of rule deletion and rule installation and monitor every 10th rule. Figure 3.3 shows the results for various flow table occupancy levels (7%, 25%, 50% and 75%). There are three main observations. First, the switch indeed does not manage to synchronize the control and data plane states.



(a) 7% (300 rules)

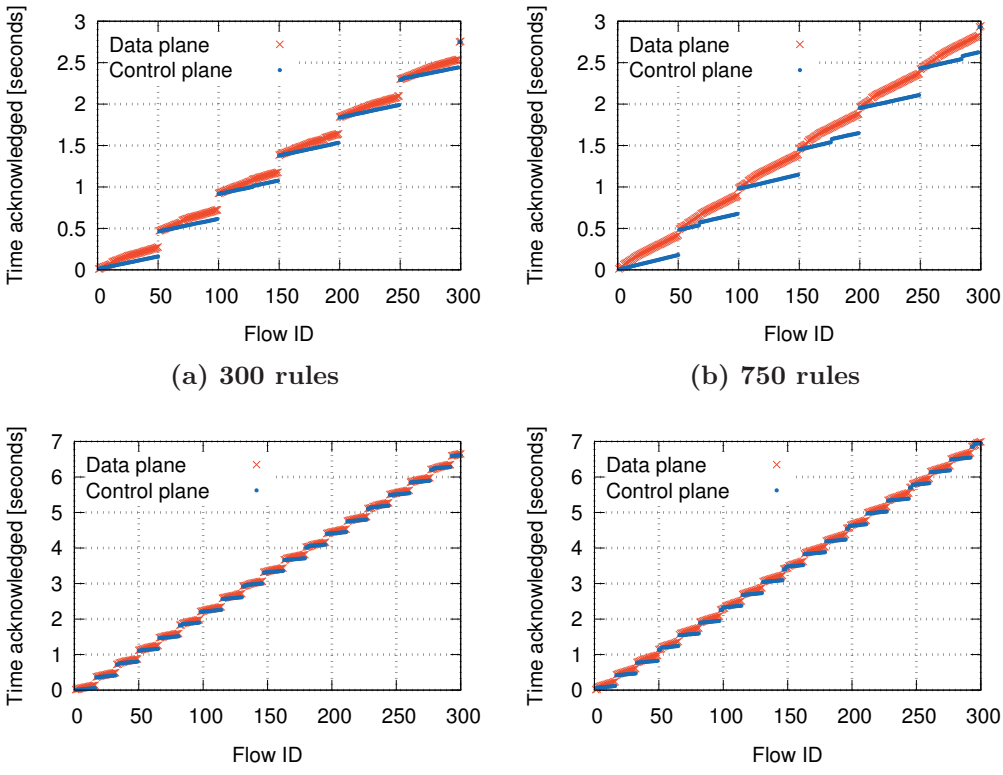
(b) 25% (1125 rules)

(c) 50% (2250 rules)

(d) 75% (3375 rules)

Figure 3.3: Control plane confirmation times and data plane probe results for the same flows in Switch X (firmware version V1) depending on flow table occupancy. The rate suddenly slows down after about 4600 flow installations (including initial rules installed before the experiment starts).

Second, the update rate increases when the switch is no longer busy with processing and sending control plane messages. This is visible as a change of slope of the data plane line in Figures 3.3a and 3.3b. We confirmed this observation by artificially occupying the switch with additional `Echo Request` or `Barrier` messages. If the switch control plane stays busy, the data plane line grows at a constant rate. We believe a low power CPU used in this switch can easily become a bottleneck and cause the described behavior. Finally, after installing about 4600 rules since the last full table clear, the switch becomes significantly slower and the gap between what it reports in the control plane and its actual state quickly diverges. We kept monitoring



(a) 300 rules

(b) 750 rules

(c) 770 rules

(d) 1300 rules

Figure 3.4: Control plane confirmation times and data plane probe results for the same flows in HP 5406zl depending on flow table occupancy. The rate slows down and the pattern changes for over 760 rules in the flow table.

the data plane for 4 minutes after the switch reported all rule modifications completed, and still not all rules were in place yet. We run additional tests and it seems that even performing updates at a lower rate (2 updates every 100 ms) or waiting for a long time (wait for 8 s after every 200 updates) does not solve the problem. The risk is that *the switch performance may degrade in any deployment where the whole flow table is rarely cleared*. We reported aforementioned issues to the switch vendor and received a confirmation and an improved firmware version.

HP 5406zl: The pattern observed in the previous experiment does not change when parameters vary except for two details depending on the flow

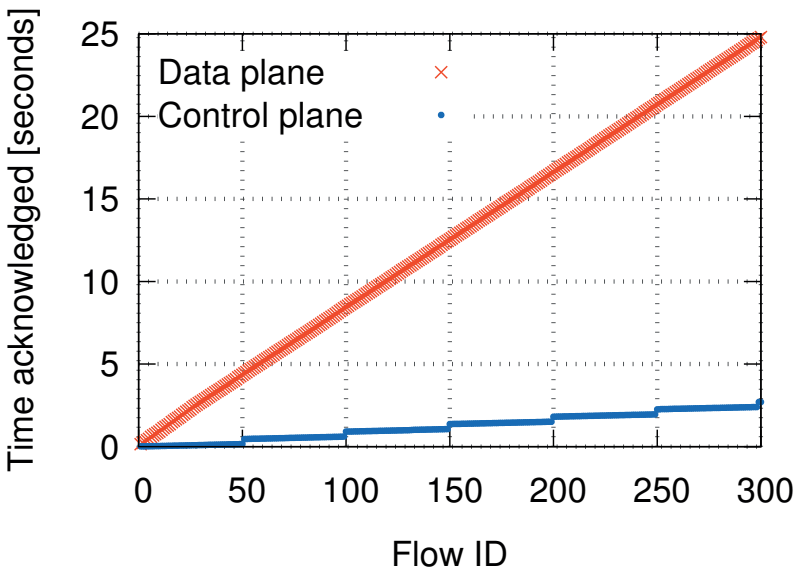


Figure 3.5: Control plane confirmation times fall behind the data plane probe results in HP 5406zl when using rules with different priorities. The scale of divergence is unlimited.

table occupancy. We show them in Figure 3.4. First, the 300 ms inactivity time is constant across all the experiments, but happens three times more often (every 33 updates) if there are over 760 rules in the flow table (Figure 3.4c). Second, when the number of rules in the flow table increases, the maximum delay between control and data plane update increases as well. It reaches 250 ms when there are 750 rules in the table (Figure 3.4b). For over 760 rules, the switch synchronizes more frequently, so the maximum delay is smaller again (Figure 3.4c) but goes back to 150 ms for 1300 rules (Figure 3.4d). We conclude that *the real flow table update speed in HP 5406zl depends on the number of rules in the table*, and the switch accounts for a possible delay by letting the data plane to catch up in regular intervals.

However, we found cases when the switch does not wait long enough, which may lead to unlimited divergence between the data and control planes. First, in Figure 3.5 we show that when different priorities are used (each rule has a different priority in this experiment), the switch becomes very slow in

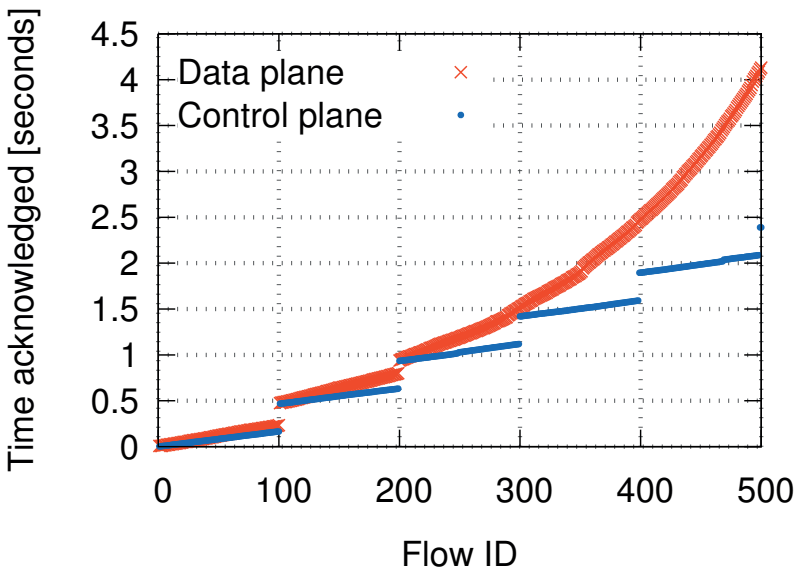


Figure 3.6: Control plane confirmation times fall behind the data plane probe results in HP 5406zl when filling the flow table.

applying the changes in hardware without notifying the control plane. This behavior is especially counter-intuitive since the switch does not support priorities in hardware. Second, our experiments show that rule deletions are much faster than installations. Figure 3.6 shows what happens when we install 500 rules starting from an empty flow table with only a single drop-all rule. Until there are 300 rules in the table, the 300 ms long periods every 100 updates are sufficient to synchronize the views. Later, the data plane modifications are unable to keep up with the control plane.

Pica8 P-3290: There are no additional observations related to Pica8 P-3290. The pattern from Figure 3.2c occurs during the whole experiment.

Dell 8132F: As depicted in Figure 3.7, the switch starts updating rules quickly, but suddenly slows down after 210 new rules installed and maintains this slower speed (verified up to 2000 batches). However, even after the slowdown, the control plane reliably reflects the state of the data plane configuration. Additionally, we observe periods when the switch does not

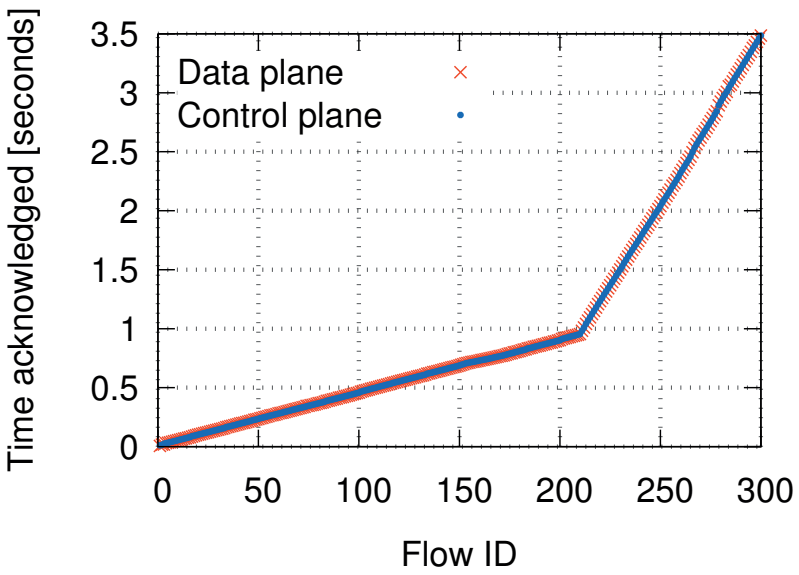


Figure 3.7: Control plane confirmation times and data plane probe results in Dell 8132F are synchronized, but the update rate suddenly slows down after about 210 newly installed rules.

install rules or respond to the controller, but these periods are rare, non-reproducible and do not seem to be related to the experiments.

Switch Y: Although in the original experiment we observe no periods of idleness, when the flow table occupancy and the experiment running time increase, the switch stops processing requests for hundreds of milliseconds (about 600 ms with 95% occupancy — Figure 3.8) every 2 seconds. Unlike HP 5406zl, here the idleness frequency depends on time, not the number of updates. Decreasing the rate at which the controller issues updates does not affect the idleness duration or frequency. During the period when the switch does not update its rules, it still responds to control plane messages (*e.g.*, barriers), but does it slightly slower, as if it was busy. We believe, this behavior allows the switch to reoptimize its flow tables or perform other periodic computations. We are in the process of explaining the root cause with the vendor.

NoviSwitch 1132: Behavior reported in Figure 3.2f repeats in longer

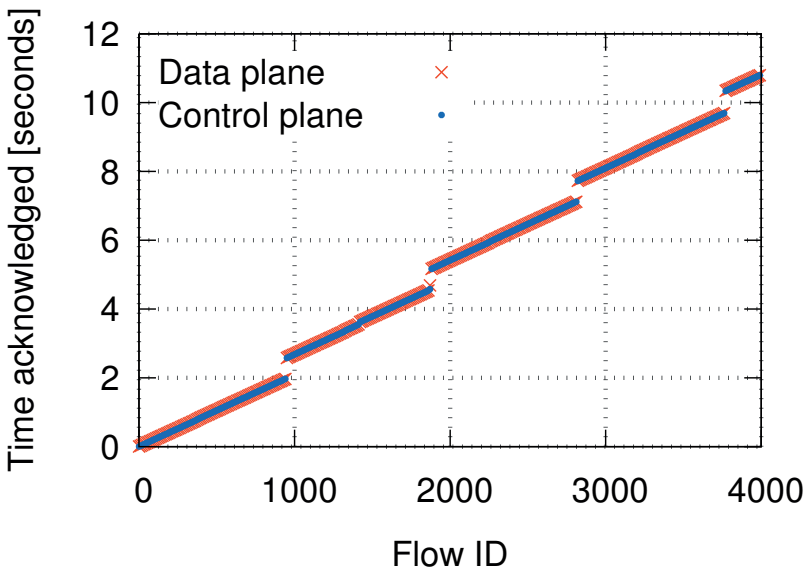


Figure 3.8: Control plane confirmation times and data plane probe results in Switch Y with 95% table occupancy are synchronized, but the switch stops processing new updates for 600 ms after every 2 s.

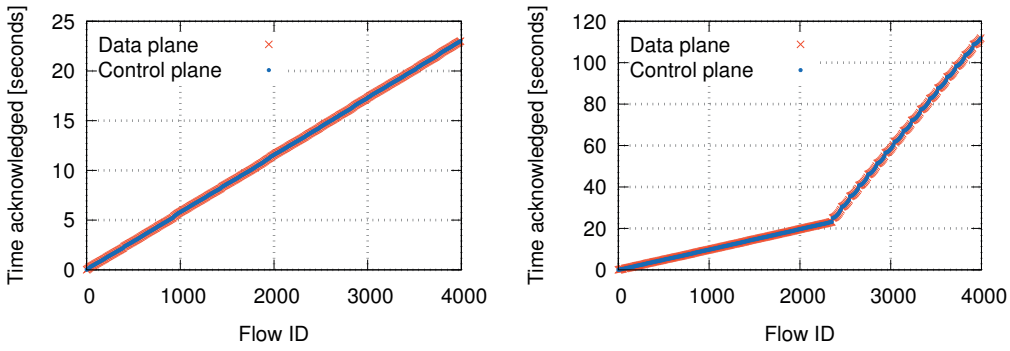
experiments as well.

Summary: *Flow table update rate often depends on the number of installed rules, but the control plane acknowledgments sometimes do not reflect this variability. A switch flow table state may be minutes behind what it reported to the control plane.*

3.2.3 Firmware Updates Can Improve Switch Performance

We reported our findings to switch vendors and some of them provided us with new, improved firmware versions.

Switch X: Most notably, Switch X with firmware version V2, no longer allows for data and control plane desynchronization. As we show in Figure 3.9, both views are synchronized and the rate does not increase when all control plane messages get processed, since they are no longer processed before the data plane update ends. On the other hand, the switch still sig-



(a) 7% (300 rules) **(b) 50% (2250 rules)**

Figure 3.9: Control plane confirmation times and data plane probe results for the same flows in Switch X (firmware version V2). Data and control plane views are synchronized, but the rate still slows down after about 4600 flow installations.

nificantly slows down after about 4600 rule installations without full table cleaning. We repeat the experiment where we perform single rule installations and deletions, keeping flow table occupancy stable. Then, we stop an experiment and resume it after 10 minutes. Figure 3.9b shows the results for occupancy of 50% (2250 rules). Behavior with the new firmware is the same as with the old version (Figure 3.3c). Finally, at the beginning, the updates are slightly slower than in the previous version and slightly faster when the switch slows down (compare to Figure 3.3).

NoviSwitch 1132: When we started our measurements of NoviSwitch 1132, the switch was running firmware version 250.3.2. The update rate was initially stable at about 60-70 rules/s, but after longer experiments started dropping to single digits and the switch required reboots. An investigation revealed that excessive logging was causing the disk space to run out in our longer and update-intensive experiments. We reported this fact and the vendor who provided us with a new firmware version: 250.4.4. A simple software upgrade allowed the switch to reach stable update rate of about 6000 rules/s — two orders of magnitude higher than before. Another upgrade

(to version 300.0.1 used to get all measurements reported in this chapter) increased the update rate by another 10-15% and fixed a bug that was causing the switch to crash when using long sequences of upgrades of rules with changing priorities.

***Summary**: Firmware is often responsible for switch faulty behavior and an upgrade can fix bugs or significantly improve performance without replacing hardware.*

3.2.4 Rule Modifications are not Atomic

Previously, we observed unexpected delays for rule insertions and deletions. A natural next step is to verify if modifying an existing rule exhibits a similar unexpected behavior.

A gap during a FlowMod: As before, we prepopulate the flow table with one low priority match-all rule dropping all packets and $R = 300$ flow specific rules forwarding packets to port α. Then, we modify these 300 rules to forward to port β. At the same time, we send data plane packets matching rules $101 - 200$ at a rate of about 1000 packets/s per flow. For each flow, we record a gap between when the last packet arrives at the interface connected to port α and when the first packet reaches an interface connected to β. Expected time difference is 1 ms because of our measurement precision, however, we observe gaps lasting up to 7.7, 12.4 and 190 ms on Pica8 P-3290, Dell 8132F and HP 5406zl respectively (Table 3.2). At HP 5406zl the longest gaps correspond to the switch inactivity times described earlier (flow 150, 200). A similar experiment with Switch X, Switch Y and NoviSwitch 1132 shows that average and maximum gaps are within our measurement precision.

Switch	Pica8 P-3290	Dell 8132F	HP 5406zl
avg/max gap in packets [ms]	2.9/7.7	2.2/12.4	10/190

Table 3.2: Time required to observe a change after a rule modification. The maximum time when packets do not reach either destination can be very long.

Drops: To investigate the forwarding gap issue further, we upgrade our experiment. First, we add a unique identifier to each packet, so that we can see if packets are being lost or reordered. Moreover, to get higher precision, we probe only a single rule (number 151 — a rule with an average gap, and number 150 — a rule with a long gap on HP 5406zl) and increase our probing rate to 5000 packets/s.

We observe that Pica8 P-3290 does not drop any packets. A continuous range of packets arrive at port α and the remaining packets at β. On the other hand, both Dell 8132F and HP 5406zl drop packets at the transition period for flow 150 (3 and 17 packets respectively). For flow number 150, HP 5406zl drops an unacceptable number of 782 packets. This suggests that the *update is not atomic* — a rule modification deactivates the old version and inserts the new one, with none of them forwarding packets during the transition.

Unexpected action: To validate the non-atomic modification hypothesis we propose two additional experiments. The setup is the same but in variant I the low priority rule forwards all traffic to port γ and in variant II, there is no low priority rule at all. Incorrectly, but as expected, in variant I both Dell 8132F and HP 5406zl forward packets in the transition period to port γ. The number and identifiers of packets captured on port γ fit exactly between the series captured at port α and β. Also unsurprisingly, in variant II, Dell 8132F floods the traffic during the transition to all ports (default

behavior for this switch when there is no matching rule). What is unexpected is that HP 5406zl in variant II, instead of sending PacketIn messages to the controller (default when there is no matching rule), floods packets to all ports. We reported this finding to the HP 5406zl vendor and still wait for a response with a possible explanation of the root cause.

The only imperfection we observed at Pica8 P-3290 in this test is that if the modification changes the output port of the same rule between α and β frequently, some packets may arrive at the destination out of order. We did not record any issues with rule modifications in Switch Y and Switch X.

Finally we observed that NoviSwitch 1132 reorders packets belonging to different flows, but the timescale of this reordering (microseconds) is much below our probing frequency. That suggests, that the reordering is unrelated to an incorrect order of rule modifications. Indeed, we confirmed that packets in different flows get reordered even if there are no rule modifications. We also checked, that packets in the same flow do not get reordered. The switch vendor confirmed that packets belonging to different flows may be processed by different cores of the network processor. They also ensured us, that assuming not too complicated actions, the processing power should be sufficient even for small packets.

***Summary**: Two out of six tested switches have a transition period during a rule modification when the network configuration is neither in the initial nor the final state.* ***The observed action of forwarding packets to undesired ports is a security concern**. Non-atomic flow modification contradicts the assumption made by controller developers and network update solutions. Our results suggest that either switches should be redesigned or the assumptions made by the controllers have to be revisited to guarantee network correctness.*

Variant	R_{hi}		R_{lo}	
	IP src	IP dst	IP src	IP dst
I	exact	exact	exact	exact
II	exact	*	*	exact
III	*	exact	exact	*
IV	exact	exact	exact	*
V	*	exact	exact	exact

Table 3.3: Combinations of overlapping low and high-priority rules.

3.2.5 Priorities and Overlapping Rules

The OpenFlow specification clarifies that, if rules overlap (*i.e.*, two rules match the same packet), packets should always be processed only by the highest priority matching rule. Since our default setup with IP src/dst matches prevents rule overlapping, we run an additional experiment to verify the behavior of switches when rules overlap.

The idea of the experiment is to install (in the specified order) two different priority rules R_{hi} and R_{lo} that can match the same packet. R_{hi} has a higher priority and forwards traffic to port α, R_{lo} forwards traffic to port β. We test five variants of matches presented in Table 3.3. R_{hi} is always installed before and removed after R_{lo} to prevent packets from matching R_{lo}. Initially, there is one low priority drop-all rule and 150 pairs of R_{hi} and R_{lo}. Then we send 500 update batches, each removing and adding one rule: $(-R_{lo,1}, +R_{hi,151}), (-R_{hi,1}, +R_{lo,151}), (-R_{lo,2}, +R_{hi,152}), \ldots$ We send data plane traffic for 100 flows. If a switch works correctly, no packets should reach port β.

Table 3.4 summarizes the results. First, as we already noted, Dell 8132F, Switch Y, Switch X and NoviSwitch 1132 do not reorder updates between batches and therefore, there are no packets captured at port β in any variant. The only way to allow some packets on port β in Dell 8132F is to increase

Switch	Observed/inferred behavior
Switch X	OK
HP 5406zl	Ignores priority, last updated rule permanently wins
Pica8 P-3290	OK for the same match. For overlapping match may temporarily reorder (depending on wildcard combinations)
Dell 8132F	OK (Reorders within a batch)
Switch Y	OK
NoviSwitch 1132	OK

Table 3.4: Priority handling of overlapping rules. Both HP 5406zl and Pica8 P-3290 violate the OpenFlow specification.

the batch size — the switch freely reorders updates inside a batch and seems to push them to hardware in order of priorities. On the other hand, Pica8 P-3290 applies updates in the correct order only if the high priority rule has the IP source specified. Otherwise, for a short period of time — 210 ms on average, 410ms maximum in the described experiment — packets follow the low priority rule. Our hypothesis is that the data structure used to store the software flow table sorts the rules such that when they are pushed to hardware the ones with IP source specified are pushed first. Finally, in HP 5406zl only the first few packets of each flow (for 80 ms on average, 103 ms max in this experiment) are forwarded to α and all the rest to β. We believe that the switch ignores the priorities in hardware (as hinted in documentation of the older firmware version) and treats rules installed later as more important. We confirm this hypothesis with additional experiments not reported here. Further, because the priorities are trimmed in hardware, when installing two rules with exactly the same match but different priorities and actions the switch returns an error.

***Summary**: Results (Table 3.4) suggest that switches may permanently or temporarily forward according to incorrect, low priority rules.*

Experiment	In-flight batches	Batch size (del+add)	Initial rules R
In-flight batches	1-20	1+1	300
Flow table occupancy	2	1+1	50 to max for switch
Priorities	as in Flow table occupancy + a single low priority rule in the flow table		
Access patterns	2	1+1	50 to max for switch +priorities
Working set	as in Flow table occupancy, vary the number of rules that are not updated during the experiment		
Batch size	2	1+1 to 20+20	300

Table 3.5: Dimensions of experimental parameters we report in this section. Note, that we also run experiments for other combinations of parameters to verify the conclusions.

3.3 Results: Flow Table Update Speed

The goal of the next set of experiments is to pinpoint the most important aspects that affect rule update speed. We first pick various performance-related parameters: the number of in-flight commands, current flow table occupancy, size of request batches, used priorities, rule access patterns. Then we sample the whole space of these parameters and try to identify the ones that cause some variation. From the previous section we know that although the control plane information is imprecise, in a long run the error becomes negligible, because all switches except for Switch X synchronize the data and control plane views regularly. Therefore, we rely on barriers to measure update rates in long running experiments used in this section. Based on the results, we select a few experimental configurations which highlight most of our findings and present them in Table 3.5.

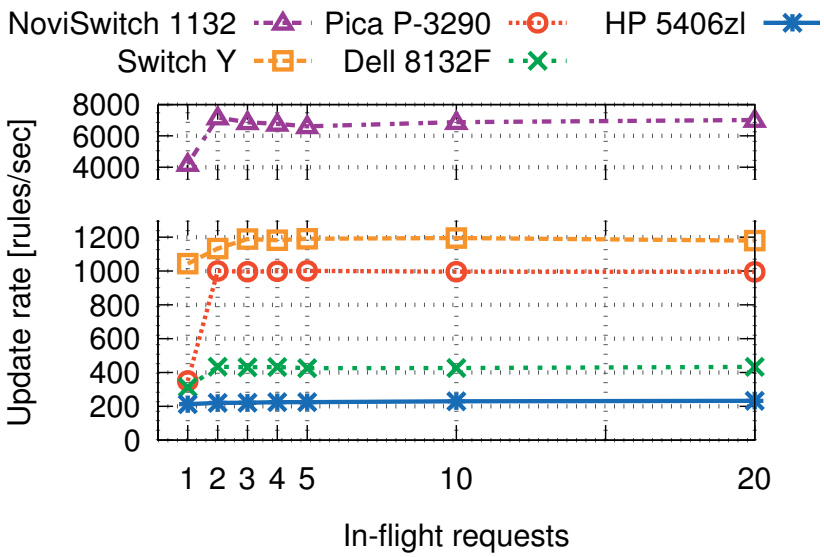


Figure 3.10: Switch performance increases with the number of in-flight requests. However, the improvements beyond the case where the controller waits for confirmation of the previous request before sending the next one ($k = 1$) are negligible.

3.3.1 Two In-flight Batches Keep the Switch Busy

Setting the number of commands a controller should send to the switch before receiving any acknowledgments is an important decision when building a controller [67]. Underutilizing or overloading the switch with commands is undesired. Here, we quantify the tradeoff between rule update rate and the servicing delay (time between sending a command and the switch applying it) to find a performance sweet spot.

We use the default setup with $R = 300$ and $B = 2000$ batches of rule updates. The controller sends batch $i+k$ only when it receives a barrier reply for batch number i. Note that barrier replies come always in order, *i.e.*, we receive configuration of batch i only after all batches $< i$ are confirmed. We vary k and report the average rule update rate, which we compute as $2 * B/T$ where T is the time between sending the first batch and receiving a barrier reply for the last and 2 comes from the fact that each batch contains one add

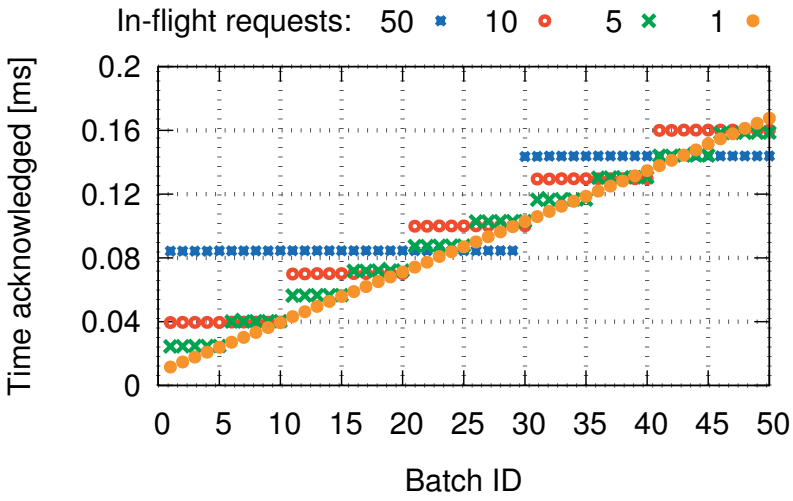


Figure 3.11: HP 5406zl barrier reply arrival times. HP 5406zl postpones sending barrier replies until there are no more pending requests or there are 29 pending responses.

and one delete message.

Figure 3.10 shows the average update rate. The rule update rate with one outstanding batch is low as the switch is idle for at least a network RTT. However, even two in-flight batches are usually sufficient to saturate tested switches given our network latencies. Thus, we use 2 in-flight batches in all following experiments. Since the update rate for NoviSwitch 1132 is often an order of magnitude higher than other switches, we use plots with a split y axis.

Looking deeper into the results, we notice that with a changing number of in-flight batches HP 5406zl responds in an unexpected way. In Figure 3.11 we plot the barrier reply arrival times normalized to the time when the first batch was sent for $R = 300$, $B = 50$ and a number of in-flight batches varying between 1 and 50. We show the results for only 4 values to improve readability. If there are requests in the queue, the switch batches the responses and sends them together in bigger groups. If the constant stream of requests is shorter than 30, the switch waits to process all, otherwise, the first response

comes after 29 requests. This observation makes it difficult to build a controller that keeps the switch command queue short but full. The controller has to either let the queue get empty, or maintain the length longer than 30 batches. But based on the previous observation, even letting the queue to get empty has minimal impact on the throughput.

***Summary**: We demonstrate that with LAN latencies two or three in-flight batches suffice to achieve full switch performance. Since, many in-flight requests increase the service time, controllers should send only a handful of requests at a time.*

3.3.2 Current Flow Table Occupancy Matters

The number of rules stored in a flow table is a very important parameter for a switch. Bigger tables allow for a fine grained traffic control. However, there is a well known tradeoff — TCAM space is expensive, so tables that allow complex matches usually have limited size.

We discover another, hidden cost of full flow tables. Namely, we analyze how the rule update rate is affected by the current number of rules installed in the flow table. We use the default setup fixing $B = 2000$ and changing the value of R.

In Figure 3.12 we report the average rule update rate when varying switch flow table occupancy. There are three distinct patterns visible. Pica8 P-3290, Dell 8132F and Switch Y express similar behavior. The rule update rate is high when the flow table contains a small number of entries but quickly deteriorates as the number of entries increases. As we confirmed with one of the vendors and deduced based on statistics of another switch, there are two reasons why the performance drops when the number of rules in the table increases. First, even if a switch ultimately installs all rules in

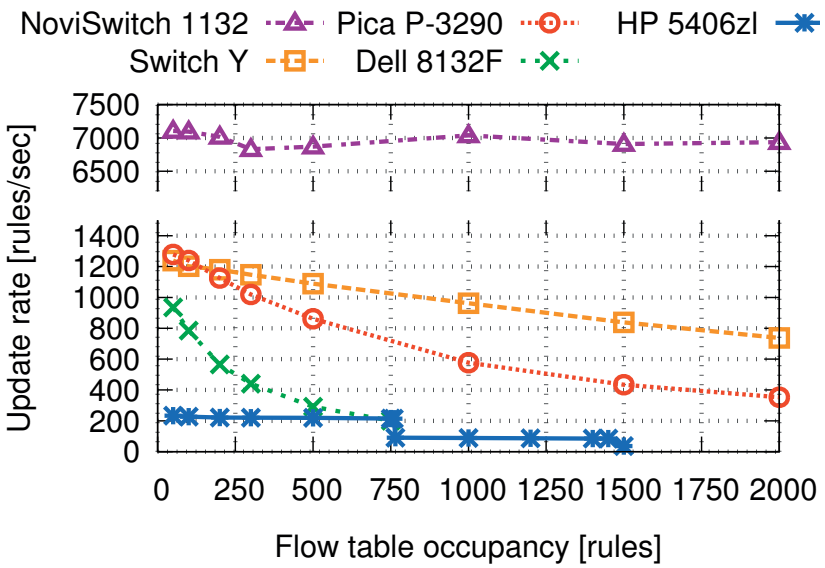


Figure 3.12: For most switches the performance decreases when the number of rules in the flow table is higher.

hardware, it keeps a software flow table as well. The flows are first updated in the software data structure which takes more time when the structure is bigger. Second, the rules need to be pushed into hardware (the switch ASIC), which may require rearranging the existing entries. Unlike other ASIC-based switches, HP 5406zl maintains a lower, but stable rate following a step function with a breaking point around 760 rules in the flow table. This stability is caused by periods of inactivity explained in Section 3.2. An update rate for NoviSwitch 1132 is an order of magnitude higher than for other switches. Additionally, the fast update rate (about 7000 updates/s) and its stability that is independent of the flow table occupancy for this device contrasts with all other switches.

Since Switch X update rate changes during an experiment and in older firmware version it does not offer a reliable way to measure its performance based on the control plane only, we manually computed update rates from the data plane experiments. As previously explained, there are three phases

Occupancy	phase I	phase II	phase III
7% (300 rules)	415 rules/s	860 rules/s	—
25% (1125 rules)	374 rules/s	790 rules/s	34 rules/s
50% (2250 rules)	340 rules/s	—	28 rules/s
75% (3375 rules)	320 rules/s	—	20 rules/s
95% (4275 rules)	302 rules/s	—	8 rules/s

Table 3.6: Flow table update rate in Switch X depending on switch state and flow table occupancy. The rate gradually decreases with increasing number of rules in the flow table. After installing a total number of about 4600 rules, the switch update rate drastically decreases.

in this switch operation: slow rate when the switch is busy with control plane, fast rate when the switch does not deal with the control plane, and a very slow phase after the switch has installed about 4600 rules. Table 3.6 contains update rates in these three phases depending on the flow table occupancy (phase II is missing when the transition to phase III happens before all control plane messages are processed, phase III is missing for 7% occupancy, because the experiment is too short to reveal it). The results show that the switch performs similarly to other tested devices (Figure 3.12) until it installs 4600 rules during the experiment. After that point the performance drops significantly (phase III). It is also visible that the switch can modify rules two times quicker when it does not need to process control plane messages (phase II).

***Summary**: The performance of most tested switches drops with a number of installed rules, but the absolute values and the slope of this drop vary. Therefore, controller developers should not only take into account the total flow table size, but also what is the performance cost of filling the table with additional rules.*

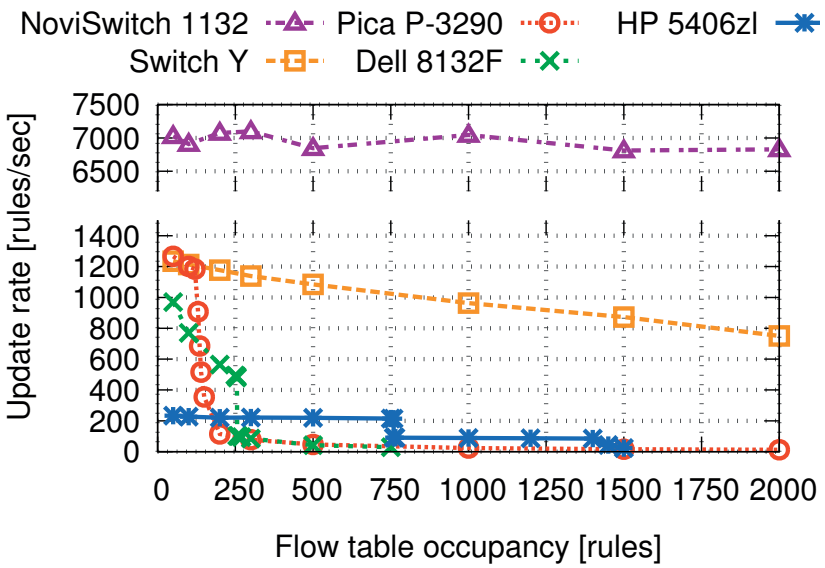


Figure 3.13: Priorities cripple performance — Experiment from Figure 3.12 repeated with a single additional low-priority rule installed reveals a massive fall in performance for two of the tested switches.

3.3.3 Priorities Decrease the Update Rate

OpenFlow allows to assign a priority to each rule, but all our previous experiments considered only rules with equal, default priorities. A packet is always processed according to the highest priority rule that matches its header. Furthermore, in OpenFlow 1.0, the default behavior for a packet not matching any rule is to encapsulate it in a `Packet In` message and send to the controller. To avoid overloading the controller, it is often desirable to install a lowest priority all-matching rule that drops packets. We conduct an experiment that mimics such a situation. The experiment setup is exactly the same as the one described in Section 3.3.2 with one additional lowest priority drop-all rule installed before all flow-specific rules.

Figure 3.13 shows that for a low flow table occupancy, all switches perform the same as without the low priority rule. However, Pica8 P-3290 and Dell 8132F suffer from a significant drop in performance at about 130 and

255 installed rules respectively. After this massive drop, the performance gradually decreases until it reaches 12 updates/s for 2000 rules in the flow table for Pica8 P-3290 and 30 updates/s for 750 rules in the flow table for Dell 8132F where both switches have their tables almost full. Interestingly, HP 5406zl's update rate does not decrease, possibly because it ignores the priorities. Switch Y and NoviSwitch 1132 update their flow tables at the same rate with and without the low priority rule. Again, for plot readability we do not show the rate for NoviSwitch 1132, which is an order of magnitude higher than other switches. We confirm that the results are not affected by the fully wildcarded match or the drop action in the low priority rule by replacing it with a specific IP src/dst match and a forwarding action.

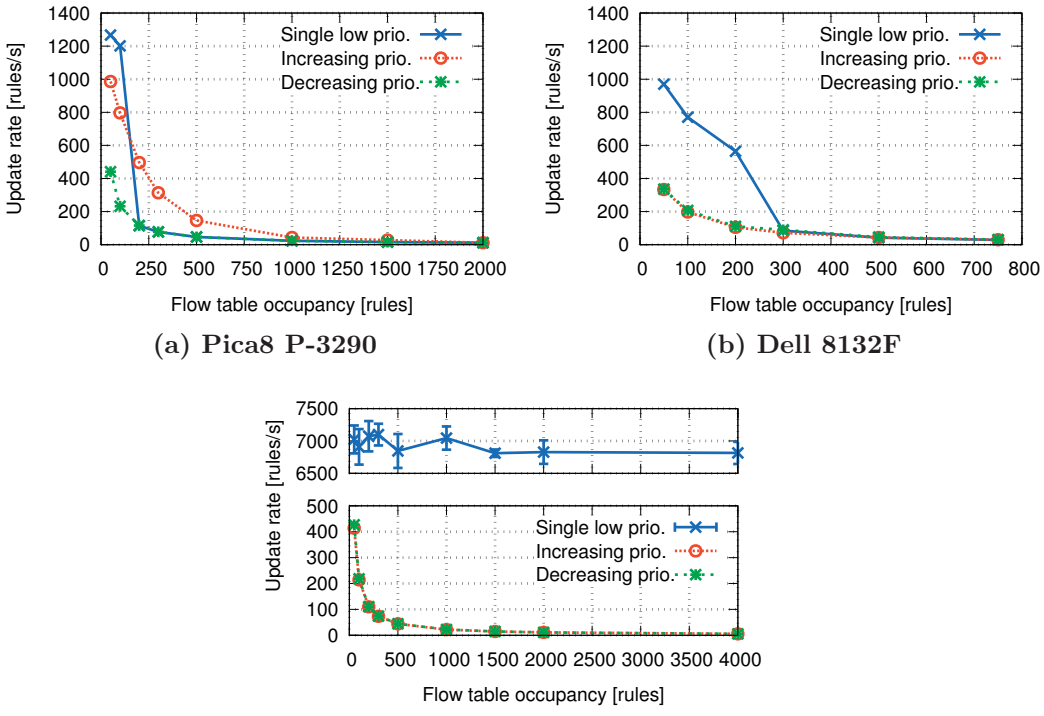
Finally, we rerun the experiments from Section 3.3.1 with a low priority rule. The rates for Pica8 P-3290 and Dell 8132F are lower, but the characteristics and the conclusions hold.

More priorities: Next, we check the effect of using different priorities for each rule. We modify the default set-up such that each rule has a different priority assigned and install them in an increasing (rule i has a priority $D+i$, where D is the default priority value) or decreasing (rule i has a priority $D-i$) order.

Switches react differently. As it is visible in Figure 3.14, both Pica8 P-3290's and Dell 8132F's performance follows a similar curve as in the previous experiment. There is no breaking point though. In both cases the performance is higher with only a single different priority rule until the breaking point, after which they become equal. Further, Pica8 P-3290 updates rules quicker in the increasing priority scenario.[6]

Figure 3.14 shows that also NoviSwitch 1132 becomes significantly slower

[6]This is consistent with the observation made in [59], but the difference is smaller as for each addition we also delete the lowest priority rule.



(a) Pica8 P-3290

(b) Dell 8132F

(c) NoviSwitch 1132

Figure 3.14: Switch rule update performance for different rule priority patterns.

when there are additional priorities used. Based on the figure, the update rate depends on the number of rules in the flow table. Even with just 50 installed rules, the rate drops from original 7000 updates/s to about 420. When the table occupancy increases the rate is as low as 5 updates/s. Update pattern does not matter — in the decreasing priority scenario the rate is minimally higher (up to 3%). In both cases, the update rate is inversely proportional to the occupancy. A deeper analysis shows, that the rate depends more on the number of priorities used than a total number of rules (Table 3.7). For example, the rate with 1000 rules in the table when rule i has a priority $D - \lfloor \frac{i}{10} \rfloor$ is almost equal to the rate with 100 initial rules in Figure 3.14. Further, it also seems that adding a rule with a new priority to the table takes a lot of time. When we run the experiment with rules using the same

Priorities	1000 rules	2000 rules
$D - \lfloor \frac{i}{10} \rfloor$	216 rules/s	110 rules/s
$D - \lfloor \frac{i}{20} \rfloor$	374 rules/s	215 rules/s
$D - (i\%10)$	5222 rules/s	5588 rules/s
$D - (i\%20)$	6468 rules/s	6142 rules/s

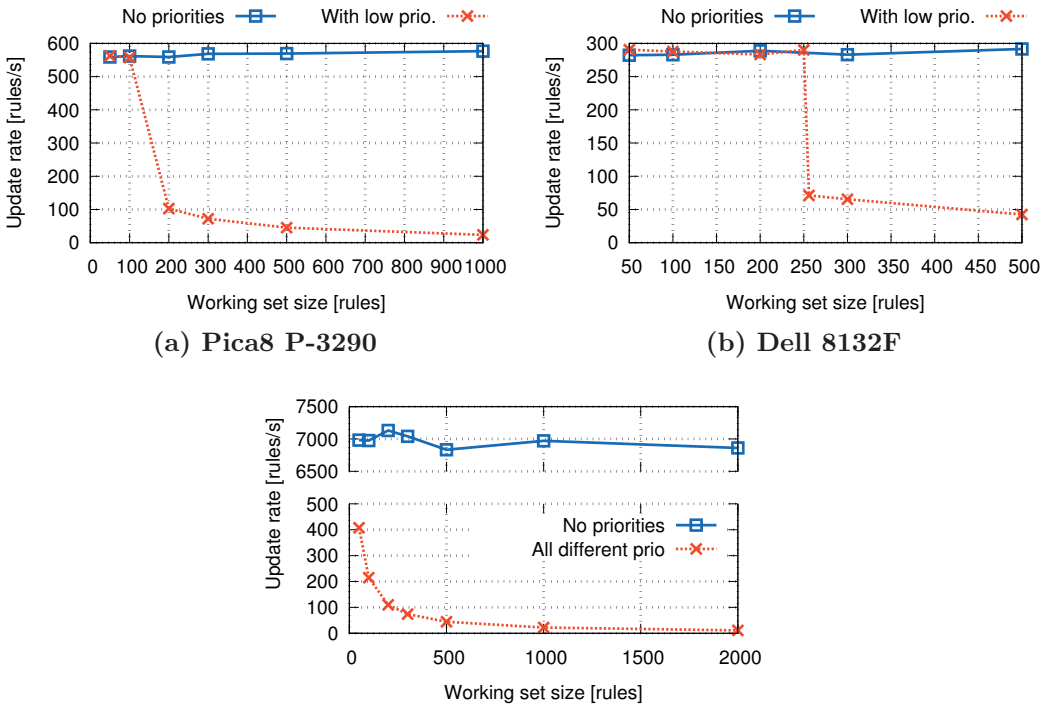
Table 3.7: Flow table update rate in NoviSwitch 1132 depending on priority patterns and flow table occupancy. The rate depends on the number of priorities in use and number of newly added priorities.

priorities as rules installed in the table before the experiment started, the rate is much higher. The vendor confirms that handling many priorities requires the switch to move some rules in TCAM, which makes updates slower. They use optimizations to reduce an impact of this movements when the number of priorities is small.

HP 5406zl control plane measurement is not affected by the priorities, but as our data plane study shows there is a serious divergence between the control plane reports and the reality for this switch in this experiment (see Section 3.4). Finally, using different priorities does not affect Switch Y performance.

Working set size: Finally, we check what happens if only a small subset of rules in the table (later referred as "working set") is frequently updated. We modify the default experiment setup such that batch i deletes the rule matching flow number $i - W$ and installs a rule matching flow i. We vary the value of W. In other words, assuming there are R rules initially in the flow table, the first $R - W$ rules never change and we update only the last W rules.

The results show that HP 5406zl performance is unaffected and remains the same as presented in Figures 3.12 and 3.13 both below and above the threshold of 760 rules in the flow table. Further, for both Pica8 P-3290 and



(a) Pica8 P-3290

(b) Dell 8132F

(c) NoviSwitch 1132

Figure 3.15: Size of the rule working set size affects the performance. For both Pica8 P-3290 and Dell 8132F when the low priority rule is installed, the performance depends mostly on the count of the rules being constantly changed and not on the total number of rules installed (1000 for Pica8 P-3290 and 500 for Dell 8132F in the plots). The same can be said about NoviSwitch 1132 with various rule priorities (2000 installed rules in the plot).

Dell 8132F a small working set for updates makes no difference if there is no low priority rule. For a given R (1000 for Pica8 P-3290 and 500 for Dell 8132F in Figure 3.15), the performance is constant regardless of W. However, when the low priority rule is installed, the update rate characteristic changes as shown in Figure 3.15. For both switches, as long as the update working set is smaller than their breaking point revealed in Section 3.3.2, the performance stays as if there was no drop rule. After the breaking point, it degrades and is only marginally worse compared to the results in Section 3.3.2 for table occupancy W.

A working set size affects NoviSwitch 1132 as well. In this case, we analyze its performance when using multiple priorities (Figure 3.15) with $R = 2000$. The rate depends on the working set size and is almost the same as the rate with the same total number or rules in the flow table.

***Summary**: The switch performance is difficult to predict — a single rule can degrade the update rate of a switch by an order of magnitude. Controller developers should be aware of such behavior and avoid potential sources of inefficiencies.*

3.3.4 Barrier Synchronization Penalty Varies

A barrier request-reply pair of messages is very useful, as according to the specification, it is the only way for the controller to (*i*) force an order of operations on the switch, and (*ii*) make sure that the switch control plane processed all previous commands. The latter becomes important if the controller needs to know about any errors before continuing on with the switch reconfiguration. Because barriers might be needed frequently, in this experiment we measure the overhead given a frequency with which we use barriers.

We repeat our general experiment setup with $R = 300$ preinstalled rules, this time varying the number of rule deletions and insertions in a single batch. To keep flow table size from diverging during the experiment, we use an equal number of deletions and insertions.

As visible in Figure 3.16, for both Pica8 P-3290 and HP 5406zl the rate slowly increases with growing batch size, but the difference is marginal: up to 14% for Pica8 P-3290 and up to 8% for HP 5406zl for a batch size growing 20 times. On the other hand, Dell 8132F speeds up 3 times in the same range if no priorities are involved. The same observation can be made for Switch Y.

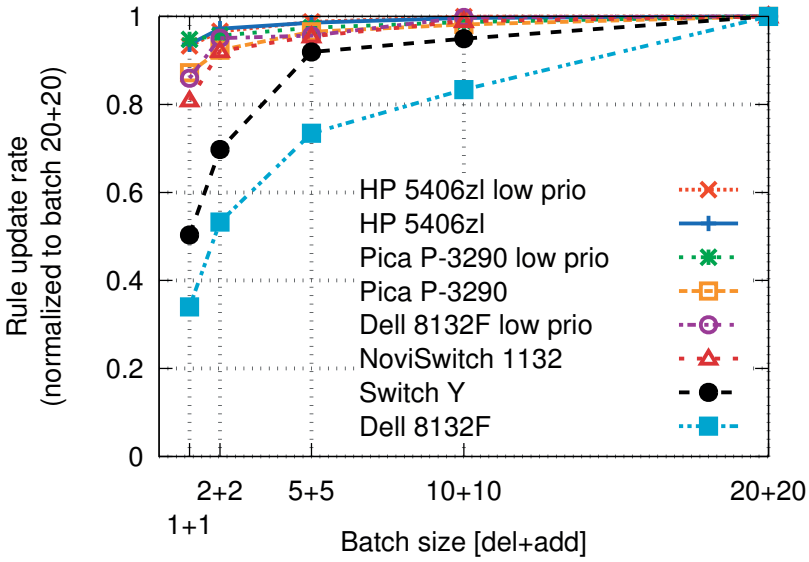


Figure 3.16: Cost of frequent barriers is modest except for the case of Dell 8132F with no priorities (i.e., with high baseline speed) and Switch Y where the cost is significant.

While further investigating these results, we verified that the barrier overhead for each particular switch recalculated in terms of milliseconds is constant across a wide range of parameters — a barrier takes roughly 0.1-0.3ms for Pica8 P-3290, 3.1-3.4ms for Dell 8132F, 1ms for Switch Y, 0.6-0.7ms for HP 5406zl and 0.04ms for NoviSwitch 1132. This explains the high overhead of Switch Y and Dell 8132F for fast rule installations in Figure 3.16 — barriers just take time comparable to rule installations. Taking into account that Switch Y and Dell 8132F are the only tested ASIC-based switches that provide correct barriers, our conclusion is that a working barrier implementation is costly.

***Summary**: Overall, we see that barrier cost varies across devices. The controller, therefore, should be aware of the potential impact and balance between the switch performance and potential notification staleness. Moreover, there is a tradeoff between correct barrier implementation and performance.*

3.4 Results: Other Surprises and Trivia

In the process of running the experiments and digging deep to find and understand the root causes of various unexpected behaviors we made additional observations. They are not worth a section on their own because they have lower practical importance or we cannot fully explain and confirm them. However, we briefly report them as someone may find this information useful or inspiring to further investigate the issue.

Rule insertion may act as a modification. In one of the experiments we show that two out of six switches are unable to perform an atomic rule modification. However, when receiving a rule that has the same match and the same priority as an already installed one, but a different set of actions, all the tested switches modify the existing rule. Moreover, this operation does not lead to any packet drops on HP 5406zl, which is better than the actual rule modification. The behavior on Dell 8132F remains unchanged.

Data plane traffic can increase the update rate of Pica8 P-3290. We noticed that in some cases, sending data plane traffic that matches currently installed rules at Pica8 P-3290 can speed up the general update rate and even future updates. We are still investigating this issue and can not provide an explanation of this phenomenon nor confirm it with full certainty, but we report it anyway as something completely counter intuitive.

Dell 8132F performs well with a full flow table. In Section 3.3.3 we report that the performance of Dell 8132F with a low priority rule installed decreases with the growing table occupancy and drops down to about 30 updates per second when the flow table contains 751 rules. We observed that this trend continues, until the table is full or there is one slot left. Surprisingly, the switch performs updates that remove a rule and install a

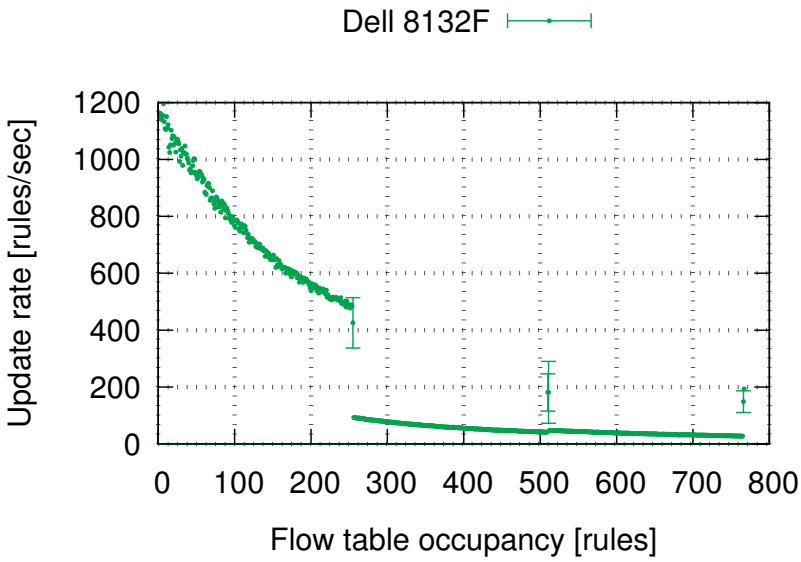


Figure 3.17: An update rate in Dell 8132F suddenly increases for 4 specific flow table occupancy values.

new one with a full table at a rate comparable to that observed without the low priority rule. We show the update rate measured for all possible flow table occupancies in an experiment with 2000 update batches in Figure 3.17. There is a sudden performance drop at 510 and 511 rules. Measurements in both these points have a very high standard deviation, but the results for a full table are stable.

3.5 Summary

The methodology presented in this chapter allows us to advance the general understanding of OpenFlow switch performance. Specifically, by focusing on analyzing control plane performance and Forwarding Information Base (FIB) update rate in hardware OpenFlow switches we detected that: (*i*) control plane performance is widely variable, and it depends on flow table occupancy, rule priorities, size of batches, and even rule update patterns — in particular, priorities can cripple performance; (*ii*) switches might periodically

or randomly stop processing control plane commands for up to 500 ms; (*iii*) the data plane state might not reflect the control plane — it might fall behind by several minutes and it might also manifest rule installations in a different order; (*iv*) seemingly atomic data plane updates might not be atomic at all.

The impact of our findings is multifold and profound. The non-atomicity of seemingly atomic data plane updates means that *there are periods when the network configuration is incorrect despite looking correct from the control plane perspective*. The existing tools that check if the control plane is correctly configured [50–52] are unable to detect these problems. Moreover, the data plane can fall behind and unfortunately *barriers cannot be trusted*. This means that approaches for performing consistent updates need to devise a different way of defining when a rule is installed; otherwise they are not providing any firm guarantees. Finally, because the performance of a single switch depends on previously applied updates, developers need to account for this variable performance when designing their controllers.

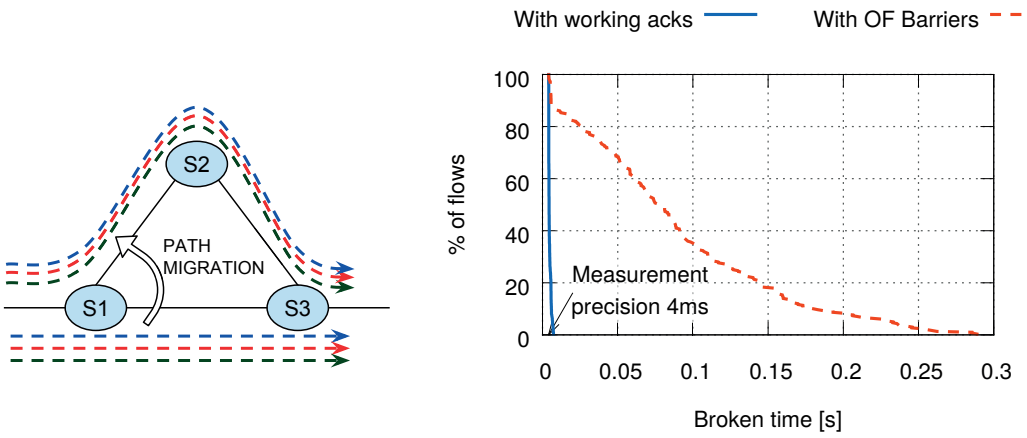
Chapter 4

RUM: Software-based Solution to Performance Differences

This chapter presents a software-based technique that addresses the security-critical issue revealed in Chapter 3: inconsistent `Barrier` implementations.

The `Barrier` commands are commonly used during network updates, when a controller modifies forwarding behavior of one or multiple switches. The network update process is complicated and if not conducted carefully, may lead to transient problems such as black holes, forwarding loops, link overload, and packets reaching undesired destinations. There are many approaches that guarantee various correctness properties [49, 60, 62, 70]. These all split an update into many smaller stages, and rely on knowing when a particular rule modification was applied at the switch(es) before proceeding to the next stage and issuing further modifications. This necessitates positive acknowledgments confirming rule modifications. Unfortunately, in OpenFlow there is no mechanism with a sole purpose of acknowledging rule modifications. Instead, there exists a `Barrier` command with a more general functionality. However, as we show in the previous chapter, not all switches satisfy the specification in this command that is crucial for correctness.

Although ultimately vendors should fix all errors in the switches, this process takes time. Instead we take advantage of one of the main selling points of SDN, and add new functionality in software quickly and for low cost. The solution presented in this chapter introduces a transparent layer below an SDN controller that provides reliable acknowledgments for rule modifications. This method relies on various methods, including data plane



(a) (b)

Figure 4.1: Consistent network update using a hardware switch. Despite theoretical guarantees, for most flows switch S1 gets updated before S2 and the network drops packets for up to 290 ms. Using our system eliminates this problem.

probing schemes, that achieve the aforementioned goal depending on switch capabilities. When using these techniques, the controller can never receive an acknowledgment before a corresponding rule is installed in the data plane. Moreover, at the cost of a higher overhead, such a layer can also provide barrier-like guarantees to the controller working with switches that do not implement barriers correctly.

4.1 Motivating Example

To demonstrate the magnitude of the problem, we prepare a small end-to-end test: We set up a network in a triangle topology with the hardware switch S2 and two software switches S1 and S3 (Figure 4.1a). We preinstall paths for 300 IP flows between hosts H1 and H2 going through switches S1 and S3. Then, we perform an update that modifies the paths to S1-S2-S3 in a consistent manner, such that a given packet can follow the old rules only or the new rules only [70].

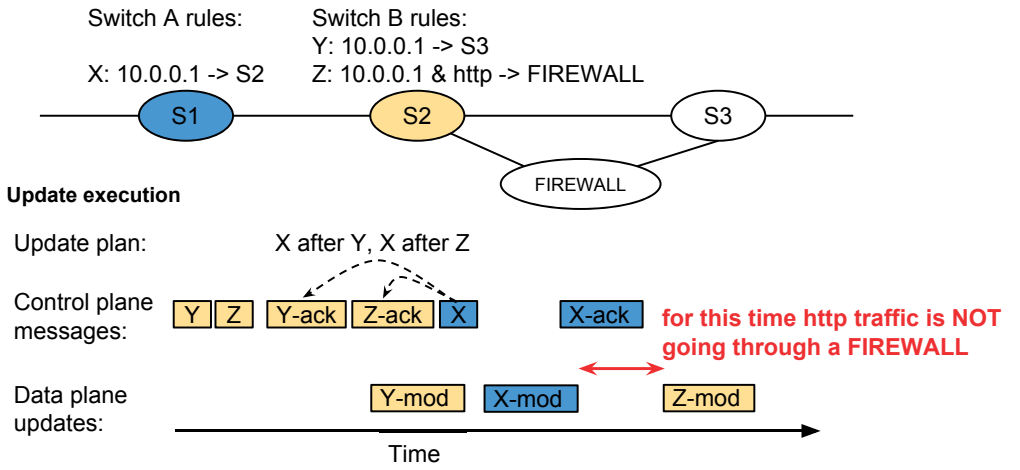


Figure 4.2: If switch B does not report data plane updates correctly, the theoretically safe update that adds rules for trusted and untrusted traffic from the same host turns into a transient security hole.

Despite using consistent updates, some flows drop packets for an extended period of time (Figure 4.1b). A detailed analysis shows that the switch sends the barrier reply up to 290 ms before the rule modification becomes visible to data plane traffic. Based on Chapter 3 we know that other switch models not only reply to barriers too early, but also reorder rule updates across barriers. The consequences of this observation have great impact — even though provably correct in theory, *none of the consistent network update techniques work in practice with buggy switches*, and all systems that build upon these techniques are unsafe as well. In particular, buggy behavior may lead to security violations, broken bandwidth guarantees, or black holes — an example of the first is depicted in Figure 4.2. If the issues that we bring up here are not addressed (*e.g.*, by adopting one of our schemes) the SDN deployments that are increasingly taking place in enterprises are in jeopardy.

While an incorrect barrier implementation may be just a temporary problem and not a fundamental limitation (some of the tested switches do implement barriers correctly), we see three main reasons why it should be imme-

diately addressed. First, there are many solutions that rely on barriers and it cannot be expected that all switches in a network will correctly function. Second, even after five major revisions the OpenFlow specification is unclear — it does not explicitly state that the commands must be applied in the data plane, instead it may be understood that the barrier enforces control plane-ordering only. This in turn means that, unless there is a high pressure from customers, vendors will have no incentive to provide data plane-level confirmations, and therefore the problem might not disappear in future switch generations. Finally, we argue that controllers need acknowledgments of each rule installation, rather than only high level barriers [68]. Therefore, we go one step further than just fixing barriers, and the solutions we propose provide such fine grained rule update acknowledgments.

4.2 System Overview

We have two main requirements in mind when designing our system called RUM (Rule Update Monitoring). First, it needs to work with existing switches and take into account their capabilities and limitations. Second, the system should provide reliable barrier commands in a backward-compatible way without requiring any modifications to the existing controllers and switches. However, it should allow the RUM-aware controllers to benefit from fine grained acknowledgments.

Acknowledging rule modifications. The first goal of the system is to provide reliable rule modification acknowledgments to the OpenFlow-speaking controllers. We design RUM as a transparent layer between the switches and the controller that intercepts and modifies the communication between them similarly to FlowVisor [77] or VeriFlow [52]. In contrast with these systems, RUM plays a more active role in the interception, as it can buffer, rate-limit,

remove or add messages. To allow easy deployment and transparency for controllers that are not designed to work with the fine grained acknowledgments, RUM adapts existing OpenFlow messages to convey successful modifications (such notifications are not available in OpenFlow). Depending on the required precision and available switch properties, the techniques (Section 4.3) rely only on the control plane communication with the tested switch, or may install additional rules and involve the neighboring switches.

Providing reliable barriers. To provide reliable barriers, RUM intercepts all `Barrier` requests and replies. After capturing a `Barrier` request, RUM holds off sending the corresponding `Barrier` reply and following messages from the switch until it can ensure that the switch completed all pending operations. An ability to correctly acknowledge commands issued to the switch is therefore the key to reliable barriers. Additionally, when working with switches that reorder modifications across barriers, RUM buffers all commands that the controller sends after the last unconfirmed barrier. It releases them to the switch after acknowledging the barrier. The barrier layer uses standard OpenFlow `Barrier` commands and is therefore transparent to any controller.

4.3 Data Plane Acknowledgments

RUM aims to acknowledge rule modifications as soon as the new rule is active in a switch data plane, but not sooner. Because different switches have different limitations and capabilities, we discuss several possible solutions to the problem at hand.

4.3.1 Control Plane Only Techniques

The first class of techniques uses control plane information only and requires modeling the switch behavior. They make minimal assumptions about the switches but instead do not offer strong guarantees.

Using OpenFlow barrier commands. Relying on `Barrier` messages is a natural way to receive acknowledgments in OpenFlow (the only way defined by the specification). Therefore, we present it as a baseline. A switch must send a `Barrier` reply message only after it finishes processing all previous commands. However, our measurements in Chapter 3 show that some switches respond to a barrier immediately, before the modifications were applied to the data plane, and as a result the data plane is often between 100 and 300 ms behind what may be assumed based on `Barrier` replies. This confirms previous measurements [72] indicating that barriers cannot be trusted and should not be used as rule update confirmations.

While one can imagine using other OpenFlow commands instead of barriers (*e.g.*, using statistics requests), we believe that such an approach does not solve the underlying problem — the reply from the switch is still based on its control plane view and/or it does not have enough temporal granularity (*e.g.*, flow statistics might be updated only once per second). Therefore, in the rest of this section we introduce techniques that still rely on the barriers, but take into account the data plane delay.

Delaying barrier acknowledgments. The first technique relies on experiments prior to deployment. If the maximum time between the barrier reply and the rule modification being applied is bounded and can be measured, RUM waits for this time after receiving a reply before confirming earlier modifications.

The main drawback of this method is that it requires precise delay mea-

surements or overestimation. We observe that in practice the delay depends on many, often difficult to predict factors and therefore providing strong guarantees is difficult (Chapter 3). For example, if the data plane is typically delayed by up to 100 ms, but there are cases of a 300-ms delay, one needs to assume the worst case scenario and always wait for 300 ms. Even then, in hard to predict corner cases, the delay may reach several seconds, which is impractical to use as the upper bound. Therefore, waiting for a timeout after each barrier has a negative impact on update performance and rule modification rate.

Adaptive delay. Adaptive timeout improves the performance of the previous technique, but requires even more detailed measurements to develop a precise switch model. Based on such models and knowing the rate at which a controller issues modification commands, RUM estimates when a particular rule modification will take place in the switch. Thus, the timeout is adjusted accordingly. However, this method requires building detailed switch performance models, which is difficult.

4.3.2 Data Plane Probes

The basic idea of data plane probes is to inject special packets into the network and use them, as well as special probing rules, to monitor which rules are active in the data plane. There are two aspects of OpenFlow `Barrier` commands: (*i*) a switch should respond with a `Barrier` reply after processing all previous commands, and (*ii*) a switch should never reorder commands separated by barriers. In practice some switches violate either the first of these properties (because they process commands in the control plane, but push the rules to the data plane later), or both. The two techniques presented in this section are designed to work correctly with such two classes of switches.

4.3.2.1 Sequential Probing

If a switch violates only the first barrier property (responds to barriers too early) two modifications separated by a barrier are never reordered in the data plane. Therefore, a strawman solution follows each real rule modification with a barrier and an additional rule installation for probing. By the time the probing rule is determined to be active (*i.e.*, it forwards a probing packet), the original rule must be in place as well.

Implementation-wise, the probing rule matches only the specially selected probe packets and has a high priority so that no other rule can override it. The probing rule sends the matching packets to the controller. RUM then repetitively injects probe packets (using a `Packet Out` message) into the switch forwarding pipeline and when the probe arrives back to RUM, it means that the probe rule is installed and therefore the corresponding real rule is active as well. Finally, after probing rule is confirmed, it is no longer needed and can be removed.

There are, however, technical details of the strawman solution that make it impractical and require improvements. First, from the correctness perspective, it assumes that the `Packet Out` processing and probe rule matching are performed in hardware. Unfortunately, this might not be the case – rules sending packets to the controller are often kept in software and may start forwarding traffic before the previous hardware rules are pushed into the data plane. As such, we modify our solution to use hardware-only probing rules – we use two additional switches[1] as depicted in Figure 4.3.

Second, inserting one probe rule after each normal rule is prohibitively expensive. Instead, we notice that a single probe rule installed after a batch

[1] In principle, switches A and C can be the same switch. We keep them separated for the presentation purposes.

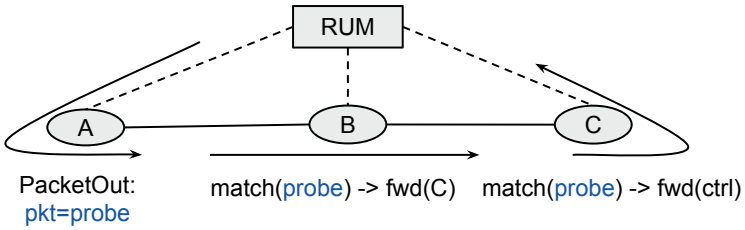


Figure 4.3: Probing the data plane at switch B. The controller (RUM) sends a probe packet from switch A to switch B. If B installed the probing rule, it forwards the packet to switch C which sends it back to the controller.

of several rule modifications acknowledges the whole batch at the same time. This way the probing overhead gets amortized over more rules at the expense of a longer acknowledgment delay. Moreover, the probe rules can be optimized even more – instead of installing a new probe rule for each batch and then deleting it, we use a single probing rule which rewrites a particular field in the packet header (*e.g.*, ToS or VLAN) with a version number of this probing rule. Then, we just update the rule to write the new version number to the probe packet header. RUM recognizes the last version of probe rule based on the probing packet headers it receives back.

Multi-switch deployment. The approach described so far requires setting up different probe rules matching different packets for each probed switch, because otherwise forwarding the probe packet on the next switch will interfere with probe collection on that switch. We overcome this problem by choosing two header fields H_1 and H_2 to be used by probing. These can be any rewritable fields in a packet header. Additionally, we reserve two special values of H_1; we call these values *preprobe* and *postprobe*. In our solution, all switches install a high priority probe-catch rule that sends all packets with $H_1 == postprobe$ to the controller. We also install one probing rule per each switch. It matches packets with $H_1 == preprobe$ and rewriting them to post-probes while also storing the per-switch unique probe rule version in

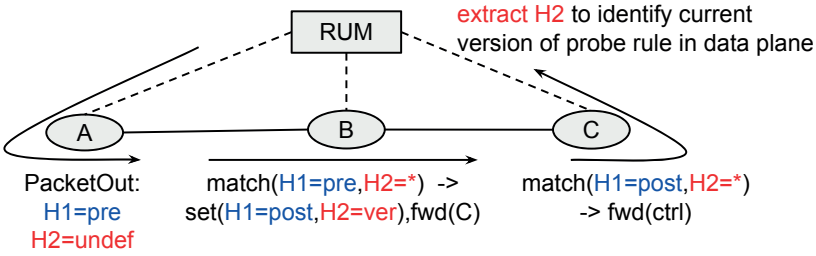


Figure 4.4: Network-wide probing solution. There are two rules preinstalled at each switch and only the version of the probing rule is updated over time.

H_2 ($H_1 \leftarrow postprobe$, $H_2 \leftarrow ver$). To do the probing, RUM sends a probe packet with $H_1 = preprobe$ inside a `Packet Out` message through switch A towards switch B as depicted in Figure 4.4.

This technique comes with two sources of overhead. First, a switch needs to install the probe rules which reduces its usable rule update rate. Further, the probe rules are probed by data plane packets, which affects the neighboring switches' control planes though the `Packet Out` and `Packet In` messages. Thus, there is a trade-off between notification delay and the usable update rate.

4.3.2.2 General Probing

The final strategy addresses the problem of switches that reorder rule modifications despite the use of barriers. In such a case, confirming that the last update took place is insufficient to acknowledge all previous updates. Specifically, it means that we cannot rely on probe rules described previously. Instead RUM needs to confirm each modification separately, where a modified rule may match an arbitrary set of header fields.

In this strategy, we need to reserve a header field H that is not used in the network, meaning that all normal rules have it wildcarded and no packet has it set to a value used by RUM (*e.g.*, VLAN, MPLS or ToS depending

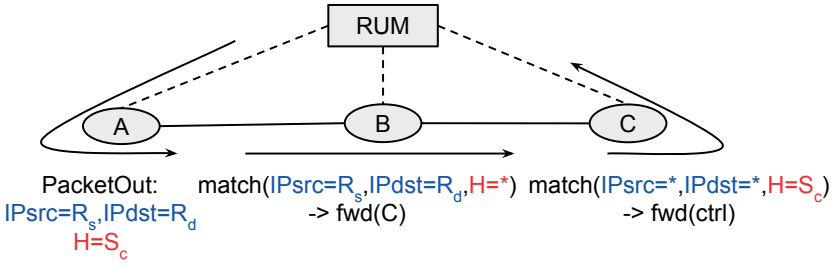


Figure 4.5: Probing for a rule matching IP packets with source R_s and destination R_d. A probe packet matches the tested rule at B and a send-to-controller rule at C.

on the deployment). At the beginning, each switch i gets assigned a unique value S_i of field H. Each switch i then installs a high priority probe-catching rule that sends all packets that match on $H == S_i$ to the controller. Figure 4.5 shows a scenario where RUM confirms the installation of the rule that matches packets with an IP source R_s and IP destination R_d and forwards them to switch C. Assuming the action of the probed rule is to send the traffic to switch C, we use switch C with its probe-catch rule matching on $H == S_c$ to receive the probes. To create the probe packet, RUM computes an intersection of the probed rule on switch B and probe-catch rule on switch C. In our example, the probe packet has $IP_{src} = R_s$, $IP_{dst} = R_d$, $H = S_c$ and arbitrary remaining header fields. This probe gets injected through any neighbor of switch B (*e.g.*, switch A). As soon as the tested rule gets installed, the controller observes the probe packet coming from switch C inside a `Packet In` message. The same method can detect rule deletions (probes stop arriving at the controller) or rule modifications (probes reach the controller from a different neighbor of B or have header fields modified to new values in case of header rewriting rules).

Overlapping rules. The previous, simplified description does not take into account the fact that there is more than one rule installed at a given switch. When creating a probe packet for a particular rule, RUM needs to take into

account other rules such that the probe does not get forwarded by any other, already installed rule. In particular, probe generation needs to address two issues.

First, the generated probe packet must not match any higher-priority rule which overlaps with the probed rule. While finding a probing packet that hits exactly the tested rule is NP in a general case, others [50,87] show that in practice the problem can be solved quickly for real forwarding tables. Second, the generated probe packet must have a different forwarding action (*i.e.*, either a different output port or a different rewrite action) than the lower-priority rule matching the probe when the probed rule is not installed yet (otherwise we cannot distinguish if the packet was processed by the probed rule or the lower-priority rule). Note that as a special case, we can probe for rules dropping packets if there exists an overlapping lower-priority which does not drop the packets (a common case of ACLs and forwarding rules combination). If no suitable probe exists, RUM falls back to one of the control plane-based techniques. For example, if the probed rule is fully covered by higher priority rules, or if it covers other, already installed lower priority rules that have exactly the same actions, probing cannot reveal when the rule got installed.

Reducing the number of switch-specific values. This technique relies on using a header field and values that are unused by the live traffic in the network. Because there may be few such fields and values, it is essential to reduce the number of required values. However, to prevent the tested switch from sending the probe directly to the controller, each two adjacent switches need to have different identifiers. Thus, instead of using a network-wide unique value of S_i for each switch i, one can solve an instance of the vertex coloring problem for which there are well known heuristics [80].

4.4 Implementation

We implement a RUM prototype that works as a TCP proxy between the switches and the controller. The switches connect to the proxy as if it was a controller, and the proxy then connects to a real controller using multiple connections, impersonating the switches. This design allows us to modularly compose RUM as a chain of proxies to add functionality and freely replace components. For example, a barrier layer built on top of the acknowledgment layer is just another proxy. We implement the proxies using the POX platform.

In the current implementation we assume IP-only traffic and rely on the ToS field for probing. Because there are only 64 ToS values, we need to periodically recycle them in longer experiments. Moreover, we assume that the rules do not overlap,[2] and therefore, selecting a probing packet degrades to using the same source and destination addresses as in the rule's match.

While the OpenFlow specification lacks messages to confirm that a rule modification was successfully applied, it defines error messages used when something goes wrong. We reuse an error message with a newly defined (unused) error code for positive acknowledgments. Alternatively, one could potentially add vendor-specific messages to the protocol.

Finally, the hardware switch we use does not support priorities but takes the rule installation order to define the rule importance. Therefore, we carefully place the low priority rules early, and make sure that other rules do not hide the high priority ones.

[2]Except a low priority drop-all and high priority probe rules

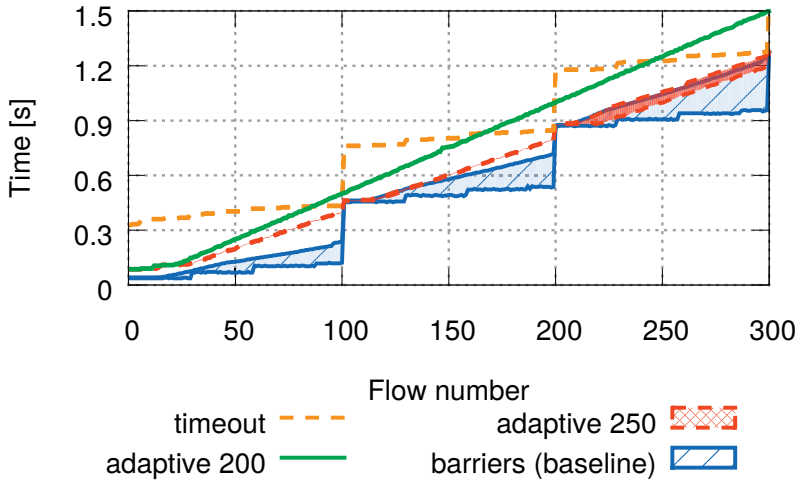


Figure 4.6: Flow update times when using control-plane only techniques. The reliability depends on correct estimation of the switch performance.

4.5 Evaluation

We evaluate RUM in the same end to end experiment as presented before, and using a hardware OpenFlow switch (HP 5406zl) that incorrectly implements barriers.[3] Further, we use low level benchmarks to analyze the properties and trade-offs in our techniques. Admittedly, these are just small scale experiments. However, a large scale test would require a testbed built of hardware switches because emulators use software switches that perform differently than the real ones. We do not have access to such a testbed.

4.5.1 End to End Experiment

We first show that the presented techniques solve the dropped packets problem described in Section 4.1. The setup is as in Section 4.1 and we send data plane traffic at a rate of 250 packets/s per flow (75000 packets/s in total). We use the previously described control plane-only techniques and in Figure 4.6

[3]The precise characteristics can be found in Chapter 3.

plot the times when the last data plane packet following the old path and the first packet going along the updated path arrives at the destination. The area between the two lines visually represents the periods when packets get dropped.[4]

An update with `Barrier` messages is the fastest, but because the `Barrier` replies are arriving too soon, rules at switch S1 get updated before rules at switch S2 are in place, which leads to extensive periods of packet drops (a total of 6000-7500 packets got lost in each of multiple runs of this experiment). The three visible steps in flow installation times are an artifact of the way how the switch synchronizes the data and control plane (check Chapter 3).

Using a 300-ms timeout solves a packet drop problem, but increases the average time it takes before a flow starts following a new path from 592 ms to 815 ms. Finally, while the 300-ms timeout is sufficient when there are up to 300 rules in the switch flow table, it becomes too short when the table occupation grows (Chapter 3).

Based on the measurements, we set the adaptive timeout to assume that a switch performs 200 and 250 rule modifications per second. We see that the technique offers a stable performance over time, however when flow table occupancy increases and the assumed update rate is overestimated (250), the acknowledgments arrive too early and the network starts dropping packets.

Figure 4.7 shows the results of the same experiment but when using the data plane probing techniques, which guarantee no packet drops. For comparison, we plot the result when all flow modifications are issued at once to all the switches (*no wait*). It shows the shortest update duration one can get, limited only by the slowest switch update rate, but also offers no theoretical consistency guarantees. The *sequential probing technique* requires additional

[4]If the delay between the two packets is lower than our measurement precision, we plot a single line.

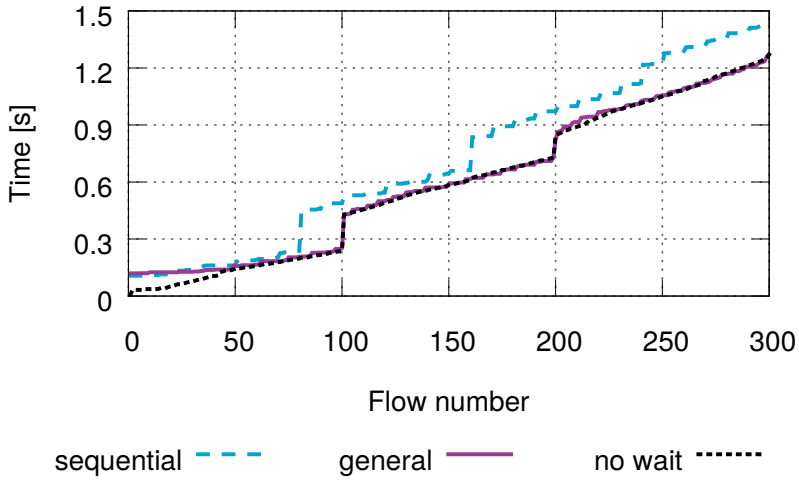


Figure 4.7: Flow update times with probing. There are no packet drops and the overhead of the general technique is negligible compared to the best achievable update time.

rule modifications (we modify a probing rule after every 10 real modifications). This fact is noticeable, because the data plane synchronization steps are more frequent which hurts the update performance. On the other hand, the *general probing technique* does not require additional rule updates, but only sending and receiving data plane probes. If probing up to 30 oldest flow modifications at once, every 10 ms, the flows get updated almost as quickly as the lower bound.

We originally send packets belonging to each of the updated flows every 4 ms and observe no drops. To verify that there are no transient periods shorter than 4 ms when packets are dropped, we randomly select a single flow ten times and send traffic for the flow at 10000 packets a second. Once again we observe no drops.

Barrier Layer Performance. To validate the overhead of a full barrier layer, we rerun the same experiment with our reliable barrier layer introduced before and sending a `Barrier` after every 10 flow modifications. When a switch does not reorder modifications across barriers, the total update time

and the particular curve of flow update times is the same as for the normal sequential probing technique. On the other hand, if the switch can reorder modifications and RUM needs to buffer them to ensure correct ordering, the overhead is big and the total update time is twice that of the general probing technique. Understandably, this time increases even more (up to 5 times) if the barriers are more frequent (up to a barrier after each command).

4.5.2 Low Level Benchmarks

After observing that RUM achieves its main high level goal - allowing for reliable network updates with consistency guarantees even on unreliable switches, we analyze how changing variables in each technique affects various aspects of the update.

The setup in the next two experiments is the same. Initially, there is a single, low priority drop-all-packets rule at the switch. Then, a controller modifies R rules in the switch in a way that at most K modifications are unconfirmed at any time. When a modification confirmation comes, the controller issues a new update. Meanwhile, we send data plane traffic matching the modified rules again at a rate of 250 packets/s for each rule.

Data plane delay. First, we measure when packets matching a particular rule start arriving at the destination (data plane activation) and when the controller receives a confirmation that the rule was installed (control plane activation). In Figure 4.8 we plot the delay between the data plane and control plane activations for various techniques for $R = 300$ and $K = 300$ (send all rules at once). All values below zero mean incorrect behavior and positive values cause a delay during an update. Thus, the ideal behavior would be a vertical line at $x = 0$. We see that, as mentioned in the introduction, `Barrier` replies arrive even 300 ms before the rule gets applied. Using

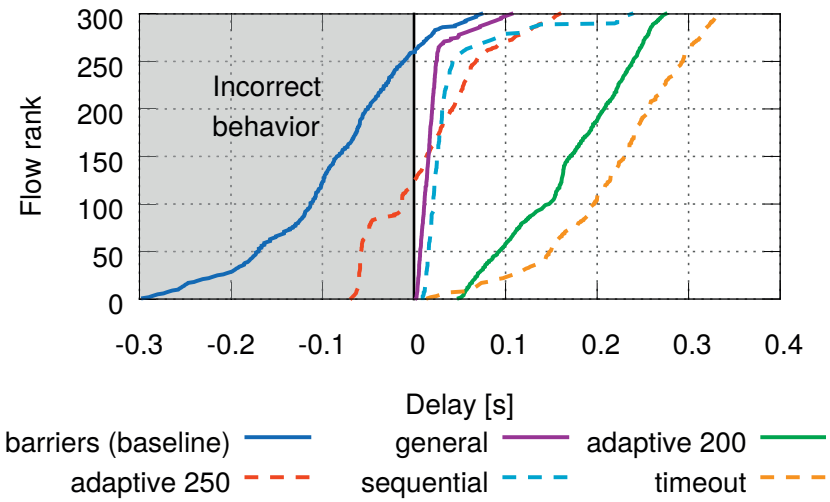


Figure 4.8: Delay between data plane and control plane activation. Using barriers leads to incorrect behaviors, control plane techniques increase the update time and the data plane techniques strike a balance.

a 300-ms timeout fixes the correctness problem in this case, but is very inefficient — for the median the update wastes 230 ms on each `Barrier`. The adaptive timeout technique achieves very good results, however, it requires precise models, otherwise the delay can fall below zero (possible inconsistencies). Finally, both probing techniques never incur a negative delay and, accordingly, are within 70 ms and 30 ms after the data plane modification for 90% of modifications.

Impact of probing rules. A technique that relies on installing probing rules to confirm that previous modifications took place requires finding a balance between the frequency of such confirmations and measurement precision. In this experiment we issue $R = 4000$ modifications and vary the number of modifications after which RUM sends a probing rule, as well as the number of allowed, unconfirmed modifications (K). Table 4.1 shows that the usable modification rate (rate of real modifications, not counting probes) is proportional to the number of rules probed at once and is usually close

Probing frequency	$K = 20$	$K = 50$	$K = 100$
after 1 update	51%	51%	51%
after 2 updates	64%	68%	68%
after 3 updates	69%	77%	77%
after 4 updates	72%	82%	82%
after 5 updates	74%	86%	86%
after 10 updates	76%	93%	94%
after 20 updates	74%	95%	98%

Table 4.1: Usable rule update rate with the sequential probing technique (normalized to a rate with barriers).

to the expected rate. When the number of allowed unconfirmed messages is low compared to the number of rules confirmed at once, the controller does not receive the confirmations quickly enough to saturate the switch.

Number of probes a switch can process. Sending data plane probes requires a switch to process two types of messages. First, an injecting switch receives a PacketOut and forwards a probe packet to the required port. Then, the receiving switch gets the packet, encapsulates it in a PacketIn message, and sends it to the controller. In the previous experiments, we used software switches as sending and receiving switches. Here, we instead benchmark the performance of a real hardware switch. We measure the PacketOut rate by issuing 20000 PacketOut messages and observe when the corresponding packets arrive at the destination. Similarly, we install a rule forwarding all traffic to the controller and inject traffic to the switch to measure the PacketIn rate. The rates are 7006 PacketOut/s and 5531 PacketIn/s, averaged over 5 runs. Both of these values are sufficient to allow RUM to probe the rules frequently.

Finally, our additional experiments show that processing PacketIn requests in parallel with rule modifications has minimal impact on the rule modification throughput — new rate is over 96% of the original rate without any other messages. Similarly, processing PacketOut messages in parallel

with rule modifications decreases the rule update rate by at most 13% for the ratio of PacketOut messages to rule modifications up to 5:1.

4.6 Summary

A system presented in this chapter shows how to utilize data and control plane separation in SDN as well as software resources of the controller machine to improve reliability of computer networks. RUM, at the cost of short added latency, makes unreliable switches usable in a network without requiring any changes in the controllers or the devices themselves.

Chapter 5

Related Work

Software Defined Networks are becoming popular in both the scientific and commercial settings. Such interest leads to many research directions that quickly advance the state-of-the-art. This chapter contains an overview of work most closely related to the topics presented in this dissertation.

5.1 Functional Testing

Functional testing tools for SDN focus either on switches or controllers, where the controller correctness is often checked by verification of rules and network configurations that it generates.

5.1.1 Switch-Level Testing

Switch-level testing treats the device under test as a black box. To ensure that the test results do not depend on external factors, the interactions with the controller and other network elements are commonly emulated by the testing framework. This approach typically requires a large number of test cases to achieve high coverage. A test specification for OpenFlow 1.0 is over 100 pages long [1] and the test specification for 1.3 version of the protocol is already 400 pages long [2]. Each test case is carefully designed to target a specific feature and checks the correctness of simple functionalities. Developers, using tools such as OFTest (a unified framework used to test correctness of OpenFlow switches) [8] have to manually provide step by step execution scenarios containing the inputs and expected outputs. SOFT automates this time-consuming and complicated process of designing test cases.

Another, more automatized, approach to black-box testing of SDN devices [83] relies on switch models to generate test cases. While the resulting set of tests allows for systematic exploration of switch behaviors, its quality depends on the model. Creating such models is difficult, requires adjustments after any specification change and if done incorrectly negatively affects test coverage. SOFT instead infers the model automatically by exploring the code itself. It also does not require correct behavior specifications.

Instead of testing the switches before deployment, some argue for verification in production [74]. This approach adds an additional load on the network and is more closely related to monitoring than testing. Cost of bugs in a running network is usually higher than for ones uncovered before deployment.

SOFT design is based heavily on recent developments in research on symbolic execution. Symbolic execution [24] and selective symbolic execution [28] is capable of testing even large systems. However, blindly applying symbolic execution results in an exponential explosion of code paths. It also requires excessive human effort to specify correct behavior. SOFT effectively overcomes these issues and goes one step further by coalescing constraints that result in the same output, and using the constraint solver to identify inconsistent behaviors.

Symbolic execution can be also efficiently applied to verify software data plane pipelines [31]. The authors show that splitting packet processing programs into smaller segments improves verification times by orders of magnitude. While functionally similar to SDN switches, software packet processors are easy to split by design and switches' code is monolithic. SOFT uses a similar idea when sending partially concretized input messages in separate test cases.

Complementary to SOFT, Kothari *et al.* [57] use symbolic execution to identify protocol manipulation attacks. The goal here is for a node to try to determine harmful behavior induced upon itself by received messages from other participants. In contrast, SOFT systematically determines and compares the input subspaces of multiple implementations to find inconsistencies, without prior knowledge of correct behavior.

While the main focus of this dissertation is on SDN switches, the topic of network devices testing is not new. We assume that the switch hardware is verified using standard verification techniques [30, 33]. SOFT is designed to check software instead. There are also various methods and tools that check correctness and performance of routers and switches as a whole [22,27,41,89]. However, these methods are mostly ad-hoc and depend on a specific protocol and device features. By simply comparing two implementations, the theory behind SOFT is protocol independent.

5.1.2 Network-Level Testing and Debugging

In the recent years researchers developed a large set of tools designed to verify network policies. However, tools such as Anteater [63], Header Space Analysis [50], NetPlumber [51], VeriFlow [52] and Libra [88] cannot detect any problem with switches. They model switch behavior instead of using real implementations.

Other tools like NoD [61] and Batfish [34] incorporate a domain specific language to analyze configuration correctness. But they also sidestep the problem of real switch correctness.

Finally, NICE [26] is a tool for testing unmodified OpenFlow controller applications. It combines model checking and concolic execution in order to systematically explore the behavior of the network under a variety of possible

event orderings. NICE and SOFT target fundamentally different parts of the network: controller vs. switches. In NICE, only the controller is running the unmodified application, while other elements (switches, end hosts) are replaced with simplified models. In contrast, SOFT finds inconsistencies among the implementations of OpenFlow agents that run in the switches.

5.2 Switch Performance Measurements

Switch data and control plane performance is essential for successful OpenFlow deployments, therefore it was a subject of measurements in the past. During their work on the FlowVisor network slicing mechanism, Sherwood *et al.* [77] report switch CPU-limited performance of about few hundred OpenFlow port status requests per second. Similarly, as part of their work on the Devoflow modifications of the OpenFlow model [29], Curtis *et al.* identify and explain the reasons for relatively slow rule installation rate on an HP OpenFlow switch. OFLOPS [72] is perhaps the first framework for OpenFlow switch evaluation. Its authors used it to perform fine-grained measurements of packet modification times, flow table update rate, and flow monitoring capabilities. This work made interesting observations, for example that some OpenFlow agents did not support the `Barrier` command. OFLOPS also reported some delay between the control plane's rule installation and the data plane's ability to forward packets according to the new rule. OFLOPS-Turbo [71] is a more recent extension of the previous work. It improves measurement precision and increases traffic generation speed. Huang *et al.* [44] perform switch measurements while building High-Fidelity Switch models that will be used during emulation with Open vSwitches. This work quantifies the variations in control path delays and the impact of flow table design (hardware, software, combinations thereof) at a coarse grain (average be-

havior). This paper also reports surprisingly slow flow setup rates. Relative to these works, we dissect switch performance at a finer grain, over longer time periods, and more systematically in terms of rule combinations, initial parameters, etc. In addition, we identify thresholds that reveal previously unreported anomalous behaviors.

Another recent measurement study [42] focuses on data plane-based update rates. We observe both data and control planes and compare states in both. We also reveal performance variability present only in longer experiments.

Jive [59] proposes to build a proactive OpenFlow switch probing engine. Jive measures performance using predetermined patterns, *e.g.*, inserting a sequence of rules in order of increasing/decreasing priority, and reports large differences in installation times in an hardware switch. The observed switch behavior can be stored in a database, and later used to increase network performance. We show that the switch performance depends on so many factors that such a database would be difficult to create.

In the early days of SDN, Bianco *et al.* [20] measured and compared the data plane packet processing performance of a software OpenFlow implementation in Linux. This work is orthogonal to our benchmarks, since we focus on flow table updates, not the packet forwarding.

5.3 Network Monitoring and Debugging

Monitoring and debugging tools constantly observe and analyze the state of the entire network. If a particular invariant gets violated, they raise an alarm. All the techniques discussed in this section work in a running system and can act only after a failure happens. Solutions presented in this dissertation work offline, before deployment. Therefore, problems get detected before they can

affect real traffic.

Automatic Test Packet Generation [87] is a solution based on Header Space Analysis that creates a minimum set of test packets required to cover all links or rules in the network. Then, ATPG uses these packets to detect and localize failures.

Similarly, Monocle [66] generates probe packets for each forwarding rule installed in a network. It then injects the probes in nearby switches to make sure that all rules expected by the controller are actively forwarding packets.

Both these systems can quickly detect problems in networks, and potentially pin them down to a single switch or even a single rule. However, they cannot answer the question if a rule is incorrect because of a bug in the switch, a faulty controller, an incorrect policy or any other reason. We concentrate on the heterogeneity of switches.

NetSight [40] is a platform that records and allows for quick retrieval of packet histories. The histories contain information about switches and rules traversed by each packet. While useful in debugging of a running network, this tool has the same limitation as the two works discussed above.

OFRewind [81] and STS [73] help with debugging network problems after they occur. Both enable temporary consistent network event trace recording in a running system. Since they rely on switch models when replaying control messages, both tools are helpless if the root cause of a particular problem lies in a switch.

5.4 Techniques to Improve SDNs

FlowVisor [77] is probably the first example of using a software layer between switches and controllers in SDN in order to add new network functionality without complicating the controller. It allows multiple controller applica-

tions to share the same network without risking conflicts with each other. FlowVisor shows that introducing such a middle layer is acceptable from the performance point of view, and is later followed by many proxy-based solutions (RUM among them).

RUM is the first attempt to look at the network update consistency from the practical point of view, using the real switches. There many solutions that guarantee particular properties during an update, but they all assume correctly-functioning switches. For example, consistent updates [70] guarantee that packets can follow either a network configuration before the update started or after it ended, never a mix of the two. Incremental consistent updates [49] reduce the rule overhead required to provide consistent updates. To avoid overloading links when rerouting big flows, zUpdate [60] takes into account additional information about the load induced by each flow. Some techniques [79] can guarantee consistency properties even in networks that use both SDN and traditional switches at the same time. Finally, Mahajan and Wattenhofer [62] introduce a taxonomy of network update consistency levels and analyze the costs of each. Chapter 3 shows that an assumption about correctly functioning switches does not hold, and RUM is a potential workaround that allows the aforementioned solutions to work correctly.

Dionysus [46] and ESPRES [67] reduce mean flow rerouting times by treating the entire network update as a scheduling problem with dependency constraints. They rate-limit and reorder particular rule updates based on runtime information to fully utilize available switches. Both systems are based on an observation that switches may apply rule updates at different, often unpredictable, speeds and therefore, following a static update schedule is inefficient. Like RUM, Dionysus and ESPRES are software based solutions, but instead of fixing switch limitations, they concentrate on improving

performance. Moreover, ESPRES relies on a correctly working `Barrier` message. RUM introduced an additional layer in the software stack that may provide the required guarantees.

NOSIX [84] notices the diversity of OpenFlow switches and creates a layer of abstraction between the controller and the switches. This layer provides a translation of commands to optimize for a particular switch based on its capabilities and performance. However, its authors do not analyze dynamic switch properties, so the results from this thesis may be useful for NOSIX to improve the optimization process. Tango [58] sets to solve a similar problem as NOSIX and improves network performance by adjusting commands to switch capabilities. Its authors go one step further and propose a method for inferring switch performance similar to the one we presented in Chapter 3. These two pieces of work are the closest in spirit to this dissertation.

In a similar fashion, Parniewicz *et al.* [65] design a hardware abstraction layer (HAL). HAL is a software layer that adds OpenFlow functionality to legacy network devices. It translates between OpenFlow messages and the proprietary configuration interface commands to mask that there are various switches in the network. We instead start with a more specific problem where all switches support OpenFlow, but not equally well.

Chapter 6

Conclusions and Future Work

Software Defined Networks owe their increasing adoption to the promise that increased programmability will lower costs and simplify network management. However, unexpected failures in production networks can quickly negate trust in such a new technology. Given how much the existing OpenFlow switches diverge from the specification and differ from each other, programming controllers is difficult. Developers have to, first, realize what special behavior each switch exhibits, and then, incorporate appropriate case handling in their code. Even measuring switch behavior in the corner cases is not trivial and in the absence of systematic methods can lead to unnoticed issues and show an incomplete picture. Further, the resulting controller software is complex, difficult to test and not future-proof.

This dissertation makes first steps towards providing controllers with a unified switch view. The two presented testing and benchmarking tools: SOFT and the switch benchmark allow users to better understand heterogeneous switch behaviors. Moreover, switch vendors and network administrators can use these tools to quickly detect problems and get confidence that their devices work correctly. SOFT detected seven inconsistencies between two software switches implementing the same specification version. They vary from relatively harmless missing features and different error messages, to such serious ones as program crashing and silently ignoring configuration messages. The switch benchmark not only classifies flow table update performance depending on changing parameters, but also revealed safety-affecting errors. Untrustworthy barriers and nonatomic rule modifications are against

common assumptions made by controller developers, and if left unaddressed may compromise network security.

RUM is a software layer that sidesteps the problem of unreliable barriers by providing true rule modification confirmations. It is also an implementation of a wider vision, where between SDN switches and controllers, there is a software processing pipeline that hides issues caused by switch heterogeneity. It takes advantage of the "software" part in SDN, since applying fixes in software is often quicker and cheaper than modifying switch hardware.

Finally, this dissertation would ideally serve as a call for other researchers to contribute to building such a software switch unification layer.

6.1 Future Work

Solutions presented in this dissertation are sufficient to defect a wide range of issues with SDN switches, but there are still many ways in which they can be improved. The ideas presented here aim to simplify the benchmarking process while improving its coverage. They also envision using collective community efforts to advance our understanding of switches and improve the quality of testing solutions.

6.1.1 Automatically Inferring Performance Corner Cases

Currently the switch benchmark includes a set of test cases that cover the space of parameters that affect flow table update characteristics. The space is sampled with statically chosen concrete values of these parameters. When operators running the tool encounter unexplained switch behaviors, they have to manually adjust the parameters in search of the root cause. Such a human-driven investigation is time consuming and requires an expert to lead the exploration.

Instead, the benchmark could take advantage of recent advancements in machine learning and optimizations. The parameter space could be explored automatically, with a goal to maximize/minimize a given metric. Such a metric can be defined, for example, as the delay between data and control plane rule activation or the overall update rate.

The improvement here can go one step further. There are techniques that automatically detect minimal causal sequences causing an error to occur [73, 85, 86]. Similarly, after finding dangerous switch behavior the benchmark could trim the sequence of messages that caused it to a minimal subset required to reproduce the case.

After incorporating such changes, the benchmark would become accessible to any switch owner. After recording an unexpected behavior and producing a minimal sequence required to reproduce the issue, users could simply send it to the switch vendors for further analysis.

6.1.2 Trustworthy Switch Models

Limited access to switches posed one of the main challenges while performing the studies reported in this thesis. A researcher or a controller developer usually has access to only few devices. Considering how much the switches differ, it is an insufficient number to properly test any solution that should work in a real network. Instead, developers usually rely on emulators and software switches. However, performance of such software switches rarely corresponds to any hardware switch.

Preparing a collection of software models of hardware switches can solve this problem. After a public release, many users of the automatized benchmark should be able to report measurements collected on devices they have access to. Precise software models can be then constructed based on such

reports and made available to the developers.

Such models would be useful in multiple contexts. First, tools like RUM should take them into account to improve the precision of rule modification confirmations. More importantly, such models are practical in controller testing. Built as a proxy placed between the controller and a switch, such a model intercepts the communication between the two. The proxy adjusts the timing and content of messages, while staying transparent for both ends of the communication channel.

We already prepared such models for HP and Pica8 switches based on measurements reported in Chapter 3. They proved to be useful when evaluating RUM and various other tools developed afterwards.

Bibliography

[1] Conformance Test Specification for OpenFlow Switch Specification 1.0.1. http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-test/conformance-test-spec-openflow-1.0.1.pdf.

[2] Conformance Test Specification for OpenFlow Switch Specification 1.3.4. http://www.opennetworking.org/images/stories/downloads/working-groups/OpenFlow1.3.4TestSpecification-Basic.pdf.

[3] CPqD OpenFlow 1.3 Software Switch. http://cpqd.github.io/ofsoftswitch13/.

[4] Floodlight Controller. http://floodlight.atlassian.net/wiki/display/floodlightcontroller.

[5] LINC OpenFlow Software Switch. http://github.com/FlowForwarding/LINC-Switch.

[6] NOX Controller. http://github.com/noxrepo/nox.

[7] OFELIA - OpenFlow in Europe: Linking Infrastructure and Applications. http://www.fp7-ofelia.eu.

[8] OFTest. http://projectfloodlight.org/oftest/.

[9] ONF Holds Its First Test Event. http://www.opennetworking.org/?p=249&option=com_wordpress&Itemid=174.

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

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

[12] POX Controller. http://github.com/noxrepo/pox.

[13] ROFL Library. http://github.com/bisdn/rofl-core.

[14] Ryu Controller. http://osrg.github.io/ryu/.

[15] Standards Supported in Cisco IOS and IOS-XE Software. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-software-releases-12-4-mainline/prod_bulletin0900aecd802eaa4f.html.

[16] Standards Supported in Cisco IOS and IOS-XE Software. http://www.juniper.net/documentation/en_US/junos12.3/information-products/pathway-pages/reference-hierarchy/junos-supported-standards.html#supported-standards.

[17] TCAMs and OpenFlow What Every SDN Practitioner Must Know. http://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/.

[18] The Cisco-Arista Battle Over CLI. http://networkcomputing.com/networking/cisco-arista-battle-over-cli/763197015.

[19] Berde, P., Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *HotSDN*, 2014.

[20] Bianco, A., Robert Birke, Luca Giraudo, and Manuel Palacin. Openflow switching: Data plane performance. In *ICC*, 2010.

[21] Bifulco, R. and Anton Matsiuk. Towards scalable sdn switches: Enabling faster flow table entries installation. In *SIGCOMM*, 2015.

[22] Bradner, S. and Jim McQuaid. Benchmarking methodology for network interconnect devices. Technical report, 1996.

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

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

[25] Canini, M., Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *USENIX Annual Technical Conference*, 2011.

[26] Canini, M., Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[27] Chang, D.-F., Ramesh Govindan, and John Heidemann. An empirical study of router response to large bgp routing table load. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[28] Chipounov, V., V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1):1–49, 2012.

[29] Curtis, A., J.C. Mogul, Jean Tourrilhes, and Praveen Yalagandula. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[30] Deal, G. K., Mark W Jennion, and Oleg Rodionov. Methods for evaluating systems of electronic components, 2002. US Patent 6,353,915.

[31] Dobrescu, M. and Katerina Argyraki. Software dataplane verification. In *NSDI*, 2014.

[32] Erickson, D. The beacon openflow controller. In *HotSDN*, 2013.

[33] Evans, A., Allan Silburt, Gary Vrckovnik, Thane Brown, Mario Dufresne, Geoffrey Hall, Tung Ho, and Ying Liu. Functional verification of large asics. In *Design Automation Conference*, 1998.

[34] Fogel, A., Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[35] Foster, N., Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

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

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

[38] Gude, N., Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3), 2008.

[39] Gupta, A., Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. In *SIGCOMM*, 2014.

[40] Handigol, N., Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[41] Hao, R., David Lee, Rakesh Kumar Sinha, and Dario Vlah. Testing of network routers under given routing protocols, 2004. US Patent 6,728,214.

[42] He, K., Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in SDN-enabled switches. In *SOSR*, 2015.

[43] Hong, C.-Y., Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

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

[45] Jain, S., Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[46] Jin, X., Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *SIGCOMM*, 2014.

[47] Kang, N., Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-Defined Networks. In *CoNEXT*, 2013.

[48] Katta, N., Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *SOSR*, 2016.

[49] Katta, N. P., Jennifer Rexford, and David Walker. Incremental Consistent Updates. In *HotSDN*, 2013.

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

[51] Kazemian, P., Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

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

[53] King, J. C. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, 1975.

[54] Kobayashi, M., Srini Seetharaman, Guru Parulkar, Guido Appenzeller, Joseph Little, Johan Van Reijendam, Paul Weissmann, and Nick McKeown. Maturing of openflow and software-defined networking through deployments. *Computer Networks*, 61, 2014.

[55] Koponen, T., Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[56] Koponen, T., Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

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

[58] Lazaris, A., Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *CoNEXT*, 2014.

[59] Lazaris, A., Daniel Tahara, Xin Huang, Li Erran Li, Andreas Voellmy, Y. Richard Yang, and Minlan Yu. Jive: Performance Driven Abstraction and Optimization for SDN. In *ONS*, 2014.

[60] Liu, H. H., Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A Maltz. zUpdate : Updating Data Center Networks with Zero Loss. In *SIGCOMM*, 2013.

[61] Lopes, N. P., Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

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

[63] Mai, H., Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel T. King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[64] McKeown, N., Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[65] Parniewicz, D., Roberto Doriguzzi Corin, Lukasz Ogrodowczyk, Mehdi Rashidi Fard, Jon Matias, Matteo Gerola, Victor Fuentes, Umar Toseef, Adel Zaalouk, Bartosz Belter, et al. Design and implementation of an openflow hardware abstraction layer. In *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*. ACM, 2014.

[66] Perešíni, P., Maciej Kuzniar, and Dejan Kostić. Rule-level data plane monitoring with monocle. In *CoNEXT*, 2015.

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

[68] Perešíni, P., Maciej Kuźniar, and Dejan Kostić. OpenFlow Needs You! A Call for a Discussion about a Cleaner OpenFlow API. In *EWSDN*. IEEE, 2013.

[69] Pfaff, B., Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, 2015.

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

[71] Rotsos, C., Gianni Antichi, Marc Bruyere, Philippe Owezarski, and Andrew W Moore. Oflops-turbo: Testing the next-generation openflow switch. In *ICC*, 2015.

[72] Rotsos, C., Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. Oflops: An open framework for openflow switch evaluation. In *PAM*, 2012.

[73] Scott, C., Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, et al. Troubleshooting blackbox sdn control software with minimal causal sequences. In *SIGCOMM*, 2014.

[74] Sharma, S., Wouter Tavernier, Sahel Sahhaf, Didier Colle, Mario Pickavet, and Piet Demeester. Verification of flow matching functionality in the forwarding plane of openflow networks. *IEICE Transactions on Communications*, 98(11), 2015.

[75] Shenker, S., M Casado, Teemu Koponen, N McKeown, et al. The future of networking, and the past of protocols. *Open Networking Summit*, 2011.

[76] Sherwood, R. Tutorial: White box/bare metal switches. In *Open Networking User Group meeting, New York*, 2014.

[77] Sherwood, R., Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[78] Singh, A., Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM*, 2015.

[79] Vissicchio, S., Laurent Vanbever, Luca Cittadini, Geoffrey Xie, and Olivier Bonaventure. Safe Updates of Hybrid SDN Networks. Technical report, UCL, 2013.

[80] Welsh, D. J. and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 1967.

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

[82] Yang, J., Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[83] Yao, J., Zhiliang Wang, Xia Yin, Xingang Shiy, and Jianping Wu. Formal modeling and systematic black-box testing of sdn data plane. In *ICNP*, 2014.

[84] Yu, M., Andreas Wundsam, and Muruganantham Raju. NOSIX: A Lightweight Portability Layer for the SDN OS. *ACM SIGCOMM Computer Communication Review*, 44(2), 2014.

[85] Zeller, A. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE99*, 1999.

[86] Zeller, A. and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.

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

[88] Zeng, H., Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.

[89] Zeng, H., Xun Zhou, and Bo Song. On testing of ip routers. In *PDCAT*, 2003.

Biography

Maciej Kuźniar is originally from Kraków, Poland. In 2011 he obtained his Master of Science degree in Computer Science from AGH University of Science and Technology in Kraków. He received awards for academic achievements in years 2007 - 2011 and graduated with honors (Diploma Summa Cum Laude). In 2011 he joined EPFL as a research intern, and in 2012 got admitted to EPFL's Graduate School to pursue his Ph.D. under the supervision of Prof. Dejan Kostić and Prof. Willy Zwaenepoel.

His research focuses on Software Defined Networks, especially aspects affecting their reliability. Additionally, he is interested in the wider area of distributed systems and computer networks. In 2015 he spent 6 months interning in an SDN team in Google, Mountain View.

His publications as of April 2016 are as follows:

- "Systematically testing OpenFlow controller applications", Peter Perešíni, Maciej Kuźniar, Marco Canini, Daniele Venzano, Dejan Kostić, and Jennifer Rexford, Computer Networks 92, 2015
- "What you need to know about SDN flow tables", Maciej Kuźniar, Peter Perešíni, and Dejan Kostić, Passive and Active Measurement Conference (PAM), 2015
- "Rule-Level Data Plane Monitoring With Monocle", Peter Perešíni, Maciej Kuźniar, and Dejan Kostić, Proceedings of the 11th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), 2015
- "Providing reliable fib update acknowledgments in sdn", Maciej Kuźniar, Peter Perešíni, and Dejan Kostić, Proceedings of the 10th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT) 2014
- "ESPRES: transparent SDN update scheduling", Peter Perešíni, Maciej Kuźniar, Marco Canini, and Dejan Kostić, 3rd ACM SIGCOMM Workshop on Hot Topics in SDN (HotSDN), 2014
- "Automatic failure recovery for software-defined networks", Maciej Kuźniar, Peter Perešíni, Nedeljko Vasić, Marco Canini, and Dejan Kostić, 2nd ACM SIGCOMM Workshop on Hot Topics in SDN (HotSDN), 2013
- "OF. CPP: Consistent packet processing for OpenFlow", Peter Perešíni, Maciej Kuźniar, Nedeljko Vasić, Marco Canini, and Dejan Kostić, 2nd ACM SIGCOMM Workshop on Hot Topics in SDN (HotSDN), 2013

- "OpenFlow needs you! a call for a discussion about a cleaner OpenFlow API", Peter Perešíni, Maciej Kuźniar, and Dejan Kostić, Proceedings of the 2nd European Workshop on Software Defined Networks (EWSDN), 2013
- "How to use Google App engine for free computing", Maciej Malawski, Maciej Kuźniar, Piotr Wojcik, and Marian Bubak, Internet Computing, IEEE 17 (1), 2013
- "A SOFT way for openflow switch interoperability testing", Maciej Kuźniar, Peter Perešíni, Marco Canini, Daniele Venzano, and Dejan Kostić, Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT), 2012
- "OFTEN testing OpenFlow networks", Maciej Kuźniar, Marco Canini, and Dejan Kostić, Proceedings of the 1st European Workshop on Software Defined Networks (EWSDN), 2012