# Flexible and Scalable Systems for Network Management

Arpit Gupta

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Nick Feamster

September 2018

# Abstract

Our daily lives are heavily reliant upon Internet-connected devices, services, and applications. This reliance makes it more critical than ever that the underlying networks they depend on be reliable, performant, and secure. At the same time, the increasing complexity and diversity of today's devices, services, and applications have made network management tasks more complicated than ever. Modern network management mandates that operators can systematically monitor what is going on in their networks (*network monitoring*) and use this information to take real-time preventive or corrective actions (*network control*). Achieving these goals while also adhering to the limited compute and storage resources available on modern network devices poses significant challenges.

The contribution of this dissertation is the design and implementation of two systems that enable flexible and scalable network monitoring and control. The network-monitoring system, *Sonata*, collects and analyzes network traffic to infer various network events in real time. The network-control system, *SDX*, enables fine-grained reactive control actions for interdomain traffic without disrupting the existing routing protocols. For each of these two systems, the dissertation focuses on (i) the *abstractions* that allow network operators to express flexible programs for both network monitoring and control; (ii) the *algorithms* that make the best use of limited compute and storage resources; and (iii) the *systems* that combine the high-level abstractions and the low-level algorithms and can be deployed in production settings.

The lessons learned from this dissertation can help us design next-generation network-management systems. More concretely, unlike existing systems that rely solely on a single device-type, this dissertation shows that designing systems that can pool resources from a heterogeneous set of devices (targets) is critical for building flexible and scalable network-management systems. It also demonstrates that as the networking technologies and protocols evolve rapidly with time, it is imperative to

design modular systems that can swiftly catch up with these changes. Finally, this research also illustrates that it is crucial to select strategic locations (e.g., Internet exchange points) for deployment to drive innovations in Internet-wide traffic monitoring and control.

# Acknowledgments

I am incredibly grateful to my advisor, Nick Feamster, for his patience, guidance, and positive reinforcement in the past five years. Nick was the best advisor I could have asked for. His leadership traits as an advisor resonated perfectly with the ones that I desired as a graduate student. It can't get any better than this. I especially admire his integrity as a researcher and how much he values academic freedom over the short-term gains. Nick not only taught me how to find the right problems to solve but also how to create an impact in the real world with academic research. Nick opened the gates of top-quality networking research for me. He ensured that I can travel to any conference I wanted, have all the resources required to succeed in my endeavors, and meet all the right people to excel as a researcher. The Ph.D. can be a very rough journey, and I think I was fortunate to have had Nick as my guide, ensuring that it was a swift and smooth ride for me.

I am very fortunate to have worked with Jennifer Rexford on both the SDX and the Sonata project. Both her energy-level and sense-of-humor are infectious. It is still an unsolved puzzle for me how she manages to be so productive yet so accessible, effortlessly providing in-depth and insightful comments. Jen taught me how to ask the right questions while doing research, how to create real-world impact with academic research, and how to offer leadership to the research community in areas of your expertise.

I am very thankful to Walter Willinger for tirelessly mentoring me. We worked together for the Sonata project. Walter was the shepherd for my SIGCOMM paper on SDX in 2014, so in a way, he mentored me for both the SDX and the Sonata projects. Walter opened the new world of streaming analytics for network monitoring to me. Walter taught me how to focus on developing an ambitious long-term research program, and how to manage risks while pursuing these ambitious goals. As a mentor, he was very approachable and always full of insightful comments and feedback. After

every meeting with him, I felt good about myself and was inspired to do better as a researcher. This experience made my last few years of Ph.D. incredibly blissful.

I am very thankful to Laurent Vanbever. He mentored me during the early years of my Ph.D. He taught me how to think like a researcher. I learned a lot working with him on the SDX and various other related projects. Laurent taught me how efficient time management makes it is possible to do excellent research while maintaining a reasonable work-life balance. I was incredibly lucky to have worked with Ethan Katz-Bassett on a couple of projects during my Ph.D. I learned from him how to create long-term impact by building and maintaining systems and tools that serve a broader research community. I am also very thankful to Anurag Kumar at Indian Institute of Science (IISc). I worked with him as a project assistant before starting graduate studies. Working with him was my first exposure to networking research and strengthened my resolve to pursue research in the networked systems area.

I am also very thankful to Marco Canini, who mentored me during the final few years of my Ph.D. Marco shared his system-building expertise with me and taught me how to ask the right questions as a systems researcher. I would also like to thank my mentors at Microsoft research (Ratul Mahajan, Monia Ghobadi, and Hongqiang Liu) and Google (Nandita Dukkipati), who exposed me to the real-world networked-systems problems. I owe a great debt to Russ Clark and Chris Tengi, who showed me various research problems related to campus networks. I am also very thankful to Rachit Agrawal, Wyatt Lloyd, Marshini Chetty, and Vyas Sekar for taking a deep interest in my work and providing feedback and support.

I am fortunate to have shared office with wonderful colleagues. Sam Burnett, Hyojoon Kim, Srikanth Sundaresan, and Robert Lychev at Georgia Tech helped me get settled as a new graduate student. It was an incredibly rewarding experience to work with Rudy Birkner, Rob Harrison, and Robert MacDavid on various research projects. I learned a lot about programmable networking from my colleagues:

Mina Tahmasbi Arashloo, Srinivas Narayana, Xin Jin, Naga Katta, Mogjan Ghasemi. Sarthak Grover, Abhinav Narain, Swati Roy, Ben Jones, Sean Donovan, and Muhammad Shahbaz have been companions in both celebration and commiseration.

The research in this dissertation was funded through the following National Science Foundation (NSF)'s awards: CNS-1539902, CNS-1704077, CNS-1539920, CNS-1409056, CNS-1040705, CNS-1040838, CNS1162112, and CNS-1261357. The writing of this dissertation was fuelled by the coffee beans sourced by Small World Coffee at Princeton, and Blue Bottle Coffee at the Rockefeller Center in the New York City.

My wife, Ankita Pawar, was extremely patient and supportive of this endeavor. Many ideas in this dissertation, especially the ones related to system design, came after intense brainstorming sessions with her. She ensured that I spent my last five years only as a graduate student, not a poor graduate student. I am indebted to my brother who shielded me away from all the filial responsibilities, making sure that I only had my research to worry about. Words aren't enough to thank my parents who made great sacrifices for me and helped me stay grounded and focused in life. My wife, brother, and parents have been a constant source of love and inspiration, without which this accomplishment would have been impossible. The least I can do is to dedicate this dissertation to them. My success is also theirs.

To my wife, brother, and parents.

# Contents

# List of Tables

# List of Figures

# Bibliographic Notes

## Academic Papers

An early version of material presented in Chapter 2 appears in an ACM HotNets paper (2016) co-authored with Rüdiger Birkner, Marco Canini, Nick Feamster, and Chris MacStoker [38]; and an Arxiv paper (2017) co-authored with Rob Harrison, Rüdiger Birkner, Ankita Pawar, Marco Canini, Nick Feamster, and Jennifer Rexford [39]. However, most material in Chapter 2 appears in an ACM SIGCOMM paper (2018) co-authored with Rob Harrison, Rüdiger Birkner, Ankita Pawar, Marco Canini, Nick Feamster, and Jennifer Rexford. The early version of the material in Chapter 3 appears in a tech report co-authored with Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Hyojoon Kim, Russ Clark, Nick Feamster, Jennifer Rexford, and Scott Shenker [41]. Most of the material in this chapter appears in an ACM SIGCOMM paper (2014) co-authored with Laurent Vanbever, Muhammad Shahbaz, Sean Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett [42]. Finally, the material in Chapter 4 appears in a USENIX NSDI paper (2016) co-authored with Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever [40].

## Talks

Below, we present the list of talks that cover the subset of materials presented in this thesis.

- *Making the "Net" Work: Flexible and Scalable Systems for Network Management* at Texas A& M (02/18), UCSB (02/18), Northeastern University (03/18),

University of Virginia (03/18), University of Minnesota (04/18), and University of Toronto (04/18). Included in chapters 1 and 5.

- *Sonata: Query-Driven Streaming Network Telemetry* at ACM HotNets (11/16), NANOG 70 (05/17), P4 Workshop (05/17), Comcast (12/16), NIKSUN Inc. (06/17), AT&T (10/17), New England Networking & Systems Day, and Boston University (10/16). Included in chapters 1, 2, and 5.
- *SDX: A Software Defined Internet Exchange* at ACM SIGCOMM (08/14), GENI Engineering Conference 20 (06/14), NANOG 59 (10/13), OpenIX Summit (04/15), Facebook Inc. (08/14), Microsoft (08/14), and NetSeminar, Stanford University (10/14). Included in chapters 1, 3, and 5.
- *iSDX: An Industrial-Scale Software Defined Internet Exchange Point* at USENIX NSDI (03/16), USENIX ATC (06/16), GENI Network Innovators Community Event (12/16), AT&T (10/15), Project Endeavour (10/15), Corsa (11/15), CloudRouter (01/16), Open Networking Foundation Webinar (04/16), Appfest (05/16), Networked Systems Laboratory, and USC (08/15). Included in chapters 1, 4, and 5.

# Chapter 1

# Introduction

Our daily lives are heavily reliant on Internet-connected devices (e.g., mobile phones, smart devices) and applications (e.g., email, video streaming, augmented reality). In turn, these networked devices and applications rely on the underlying networks. Network operators, responsible for managing these networks, are required first to monitor various network events (*network monitoring*) and then react to these events in real time (*network control*). However, the scale and the diversity of these Internet-connected devices and applications have significantly increased the complexity of network management.

In this chapter, we first describe how the growing complexity of modern networks makes it harder for the network operators to perform flexible and scalable network monitoring and control in Section 1.1. We then present two systems that bridge the gap between flexibility and scalability for network monitoring and control in Section 1.2 and Section 1.3, respectively. We then summarize the lessons learned from this dissertation in Section 1.4. Finally, we present the outline of this dissertation in Section1.5.

## 1.1 Network Management

To keep the networks running, network operators need to perform two basic tasks: (1) network monitoring, and (2) network control. In this section, we describe the flexibility requirements of these two tasks and network resources available for their scalable execution—highlighting the gap between flexibility and scalability.

### 1.1.1 Flexibility Requirements

In this dissertation, we define the *flexibility* of programs for network management (*i.e.*, queries for network monitoring and policies for network control) in terms of the level of granularity at which they can operate (*match*) and the range of operations that they can perform (*actions*).

**Flexibility Requirements for Network Monitoring.** Network operators monitor the network to identify what events are going on in their network. For example, network operators managing the network for a cloud service provider, e.g., Google, might be interested in figuring out if the video streaming traffic that they deliver to their customers is jittery. They might also be interested in figuring out whether some of the hosts within their network are victims of a DNS-based reflection attack [84, 62], *i.e.* if any host in their network is receiving DNS response messages from too many distinct hosts. Note that both these monitoring tasks require collecting an aggregate metric (*i.e.*, jitter and the number of unique DNS response messages for the two tasks, respectively) from subsets of the total traffic (*i.e.*, video streaming and DNS response traffic, respectively). Clearly, there are many ways in which network operators can specify the subsets of traffic of interest (defined based on address, protocol, payload, device, location). Similarly, there are many metrics (e.g., jitter, distinct hosts, volume, delay, loss) they might want to extract. Thus, it is critical that net-

| | Routers | Programmable Switches | CPUs |
|---|---|---|---|
| **Match** | IP prefixes | All header fields | All header and payload fields |
| **Actions** | forward, drop | forward, drop, add, subtract, bit operations | Any |
| **Speed** | $O(ns)$ | $O(ns)$ | $O(\mu s)$ |
| **Memory** | $O(1M)$ | $O(100K)$ | $O(1B)$ |

Table 1.1: Available network resources. The match and action capabilities represent the flexibility, and the speed and memory represent the scalability of each target.

work operators should be able to express queries for network-monitoring tasks with as much flexibility as possible.

**Flexibility Requirements for Network Control.** Once the network operators have figured out what events are going on in the network, they want to react to those events in real time. For example, after detecting jitter in the video streaming traffic, the same operators at Google might want to use a different path for this traffic. Also, after identifying the presence of DNS reflection attack traffic, the network operators might want to block the attack traffic. Notice that for both of these tasks, network operators are interested in defining a specific action (e.g., redirect and block, respectively) for subsets of the total traffic (e.g. video streaming and DNS response traffic, respectively). Again, similar to the case of network monitoring, there can be many ways in which network operators might want to specify the portion of traffic that they are interested in and the actions (e.g., redirect, drop, rate limit, modify) they want to apply over the selected traffic. Thus, similar to network monitoring, it is critical that network operators should be able to express programs for network control tasks with as much flexibility as possible.

### 1.1.2 Available Network Resources

Network operators rely on various network devices, such as switches and routers, to execute multiple monitoring and control tasks. We classify these network devices into three categories based on the flexibility (*i.e.*, their match and action capabilities)

and scalability (*i.e.*, their packet processing, and storage capabilities) metrics (see Table 1.1).

**Conventional Routers.** Networks that carry high-volume traffic require network devices to maintain state for millions of unique flows (aggregated based on the destination IP prefixes) and process packets belonging to these flows in a few nanoseconds. The most commonly used devices for such networks rely on conventional routers that are specially designed to match on the destination IP prefixes using ternary content accessible memory (TCAM) and apply commonly-used actions such as `forward`, `drop`. They are designed to trade flexibility for scalability, maintaining the forwarding decisions for millions of flows (identified by destination IP address) in the network and processing the incoming packets in the order of few nanoseconds. They cannot apply any custom operations over the incoming traffic for monitoring or apply any fine-grained control actions. These devices are *scalable, but not flexible*.

**General-purpose CPUs.** At the other end of the design spectrum, we have general-purpose CPUs as network devices. They are generally not identified as network devices, but most of the existing flexible network-monitoring systems (e.g., Gigascope [23], NetQRE [129]) rely on CPU-based servers for answering a wide range of queries concerning various monitoring tasks. These devices can support more complex parsing operations, and can thus match on deeper packet fields, including the payload fields. They use dynamic random access memory (DRAM), and can, therefore, support a wide range of stateful operations at cheaper costs. However, this flexibility comes at the expense of speed because their packet-processing time is in the order of microseconds. Also, as these devices are typically not part of the data path and require mirroring traffic from the data plane for processing, they incur additional bandwidth cost. These devices are *flexible but not scalable*.

**Programmable Switches.** In recent years, we have witnessed significant developments in the design and implementation of programmable switches that are more flexible than routers (but not as flexible as CPUs) and faster than CPUs. Unlike conventional routers, which are designed to only match on destination IP prefixes, these devices are more flexible as they can be programmed to match on any arbitrary combination of multiple packet header fields. Unlike the CPUs, they are more scalable as they can process packets at line rate. However, this additional flexibility comes at the expense of limited state (*i.e.*, they can only support few hundred thousand match-action entries compared to millions supported by conventional routers).

More recently, we have also witnessed the growing popularity of reconfigurable protocol-independent switch architecture (PISA) targets [15, 50, 120] in data-center and wide-area networks. Unlike conventional network devices, these switches support custom packet processing and state management, implementable in the P4 language [14]. More specifically, PISA targets allow programmatic specification of new header formats for parsing packets. They support computing complex aggregate metrics using static random access memory (SRAM) as hash tables and carrying state along with the packet through the packet processing pipeline or on to the next switch using custom metadata fields. They also support flexible match/action tables with programmable actions. These capabilities make it possible to apply flexible operations over the incoming traffic at scale.

### 1.1.3 Flexibility and Scalability Gap

As shown in Figure 1.1, there exists a gap between flexibility and scalability for efficiently managing today's complex networks. This gap exists for two reasons. On the one hand, we require systems for network management that can support limitless creativity regarding expressing programs for figuring out what events are going on in the network (e.g., congestion, link failures, DDoS attacks, etc.), and then reacting

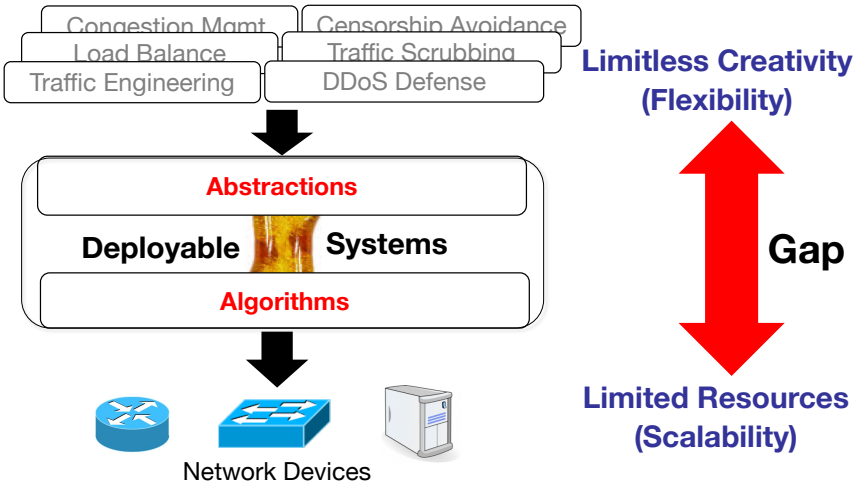



Figure 1.1: This dissertation focuses on filling the gap between flexibility and scalability for network management. It presents the design and implementation of two systems for network monitoring and network control respectively. Each makes unique contributions categorized regarding new abstractions, algorithms, and systems.

to those events in real time. At the same time, we are faced with limited resources in the network. While programmable switches are more flexible than conventional switches and faster than CPUs, they are just a tool and cannot fill the gap between the flexibility we require and the scalability we look for.

This dissertation focuses on the design and implementation of two systems, **Sonata** and **SDX**, both of which harness the power of programmable switches to bridge the gap between flexibility and scalability. To fill this gap (see Figure 1.1), we present: (1) the *abstractions* that we designed to make it easier for network operators to express flexible programs for network monitoring and control, (2) the *algorithms* that we developed to make the best use of limited network resources, and (3) *systems* that glue the high-level abstractions to the low-level algorithms.

## 1.2 Network Monitoring with Sonata

To describe Sonata, we first define the problem that it is trying to address and then elaborate on the unique observations that inspired Sonata's design. Finally, we sum-

marize the contributions in terms of abstractions, algorithms, and systems, highlighting how in combination they enable Sonata to fill the aforementioned flexibility-scalability gap.

### 1.2.1 Problem

Network operators need to monitor a wide range of network activities concurrently. For example, they are required to detect whether the network is under attack and also determine whether there is a device failure in the network at the same time. This process involves extracting multiple features from the traffic data and combining them to infer activities of interest in real time. Most existing real-time network-monitoring systems are either not flexible (*i.e.*, they support an insufficient set of monitoring tasks), or they are not scalable (*i.e.*, they fail to scale as the traffic volume intensifies and the number of monitoring tasks increases). This dissertation presents the design and implementation of a flexible and scalable network-monitoring system that can execute *multiple flexible* monitoring queries over *high-volume* network traffic in real time.

### 1.2.2 Observations

This dissertation leverages two observations to scale the execution of flexible queries for network monitoring. (1) Many existing network-monitoring systems take advantage of either scalable stream processing or programmable data-plane targets for network monitoring—but not both. While at first glance, these two technologies seem inherently different, they both apply a sequence of transformations over packets (tuples). This dissertation leverages the inherent similarity between the two targets to design a network-monitoring system that uses both stream processors and programmable switches for query execution. (2) For many network-monitoring tasks, the fraction of relevant traffic is typically tiny. This dissertation leverages this obser-

vation to design a *query-driven* network-monitoring system which relies on the output of queries to iteratively zoom-in over the relevant traffic relevant.

### 1.2.3 Contributions

**Abstractions for expressing network-monitoring queries.** Requiring network operators by themselves to individually configure both programmable switches and stream processors for each query separately can be overwhelming. We design a query interface that unifies the parsing and compute capabilities of a programmable switch with those of stream processors. This interface allows network operators to apply familiar dataflow operators (e.g., `map`, `filter`, `reduce`) over arbitrary combinations of packet fields without regard to where the query will execute. We show that a wide-range of network monitoring tasks (see Table 2.5) can be expressed in fewer than 20 lines of code.

**Query-planning algorithms.** To reduce the load on the stream processor, Sonata's query planner would like to execute the monitoring queries using only the hardware switches. However, the data-plane resources, such as switch memory and processing stages, are quite limited and inelastic. Thus, Sonata's query planner needs to partition the given monitoring queries between the switch and the stream processor. To design the *query-partitioning algorithm* that makes the best use of these resources, we developed a model that accurately captures the common data-plane resource constraints and quantified how high-level dataflow operators consume these resources.

To further use limited switch computational and memory resources efficiently, we developed a *dynamic-refinement algorithm* that, instead of processing all traffic, allows the switch to focus only on the subset of traffic that satisfy a query. We show that this technique applies to a wide range of monitoring queries and demonstrate how

Sonata's query planner considers the structure of queries and representative traffic traces to compute a refinement plan for each query.

**Sonata's design and implementation.** We designed the system to be modular such that it can support various types of data-plane and stream processing targets and can be easily extended to support operations over arbitrary packet fields. We also designed Sonata's query interface to be platform-agnostic, *i.e.*, the network-monitoring queries expressed using Sonata's query interface are agnostic to the underlying switch and streaming targets.

Sonata's prototype implementation consists of only about 9,000 lines of code and compiles queries to a single programmable switch. It implements drivers for both hardware [112] and software [114] protocol-independent switches (e.g., Barefoot Tofino [112]) as well as the Spark Streaming [111] stream processor. The current prototype parses packet headers for standard protocols and can be extended to extract other information, such as queue size [46]. We use real packet traces from operational networks to demonstrate that Sonata's query planner reduces the load on the stream processor by as much as *seven orders of magnitude* over existing network-monitoring systems. We also quantify how Sonata's performance gains depend on various data-plane constraints and traffic dynamics.

## 1.3 Network Control with SDX

To describe SDX, we first define the problem that it is trying to address and then elaborate on the unique observations that inspired SDX's design. We then summarize our contributions in terms of abstractions, algorithms, and systems, highlighting how these contributions enable filling the gap between flexibility and scalability. We also describe how we further improved the scalability of the SDX system by designing and implementing an industry-scale SDX (iSDX).

### 1.3.1 Problem

After using a network-monitoring system to infer various network events, network operators need to apply fine-grained reactive control actions in the data plane, usually in multi-domain settings. This part of the thesis focuses on applying flexible control for the interdomain network settings, where by default the networks use the Border Gateway Protocol (BGP) to exchange traffic with each other. In particular, this research demonstrates that BGP is not suited for applying flexible control actions. Ideally, one would like to replace BGP with a clean-slate solution supporting programmatic control at scale. However, the existing Internet-wide deployment of BGP-speaking routers makes such an approach impractical. Thus, our goal is to design and implement an incrementally deployable, flexible, and scalable system that ensures maximal impact with minimal deployment overhead while safely interoperating with BGP.

### 1.3.2 Observations

In recent years, the proliferation of Internet-connected devices and applications have also contributed to making the Internet flatter and more densely interconnected. The emergence of Internet exchange points (IXPs) contributes to the current state of global interconnections. These IXPs provide a common switching fabric for various networks for exchanging traffic with each other and are strategically located to influence a significant portion of the Internet's interdomain traffic. In this part of the dissertation, we present software-defined exchange (SDX) [34] that replaces the conventional switching fabric with programmable switches at IXPs. While simple in theory, this dissertation shows that building a practically deployable SDX requires striking a delicate balance between flexibility and scalability while ensuring that the system safely operates with the existing routing protocol, *i.e.*, BGP.

### 1.3.3 Contributions

**Abstractions for expressing network-control programs.** Participating networks need a way to create and run flexible control programs, without conflicting with each other or with the global routing system. SDX presents each participating network with the illusion of its virtual programmable switch that extends the footprint of its legacy network and enables flexible policies that interact safely with today's BGP.

**Compilation algorithms.** An SDX needs to compile flexible control programs for hundreds of participants, hundreds of thousands of IP prefixes, and policies that match on multiple packet-header fields—all while using a single commodity (programmable) hardware switch. We show how to combine the policies of multiple participants and join them with the current BGP routes while limiting the rule-table size and computational overhead. More specifically, we develop the *minimum-disjoint-set algorithm* that assigns a unique forwarding tag for all prefixes that exhibit forwarding equivalence. For iSDX, we developed the *reachability-encoding algorithm* that encodes reachability information for each IP prefix as a bitmask—further reducing the rule-table size in the data plane.

**SDX's design and implementation.** We have built a prototype of SDX and created two example applications. Experiments demonstrate that our prototype scales (in terms of rule-table size and CPU time) to many participants, policies, and prefixes. We have also implemented a public, open-source implementation of iSDX, available on Github [48]; the system is based on Ryu [98], a widely used SDN controller, and comes with tutorials and instructions that have already helped spur early adoption. An extensive trace-driven evaluation demonstrates iSDX's scalability characteristics using real-world data from one of the world's largest IXPs. Our evaluation both

demonstrates that iSDX can scale to the workload at a large IXPs and provides insight into precisely how (and to what extent) each of our optimizations and algorithms helps iSDX scale.

## 1.4 Lessons Learned

The exponential increase in the number of Internet-connected devices and applications will require performing more and more flexible network monitoring tasks for high-volume traffic—overwhelming the human operators. We expect that the next generation of network-management systems will need to be not only more flexible and scalable but also capable of running the networks with minimal human intervention. We will now summarize the lessons learned from this dissertation that can guide the design and implementation of these next-generation network-management systems.

### 1.4.1 Pooling Heterogeneous Resources

In theory, we would like to design a special-purpose target that is tailor-made for the problem at hand. However, in practice, such an approach is not feasible. As a result, system designers end up selecting a best-fitting target, which they utilize for all their problems. However, as different network devices, in general, have different capabilities, using only a single device type irrespective of the nature of the problem is sub-optimal. It is critical to pool resources of multiple heterogeneous sets of targets for a given problem.

In this dissertation, we demonstrate that combining the capabilities of a heterogeneous set of network devices or targets creates opportunities to build both flexible and scalable systems for network management. To ensure flexible network control at IXP, SDX compensates for the limited TCAMs of the programmable switch by designing mechanisms that offloads the task of matching on IP prefixes to the fixed-function

border routers, making the best use of the two different targets. Similarly, to ensure flexible and scalable network monitoring, Sonata compensates for the slow packet processing capabilities of the CPU-based targets by opportunistically offloading as much packet processing as possible to the programmable switch, again making the best use of the two different targets.

### 1.4.2 Designing Modular and Extensible Systems

Network monitoring (control) systems perform various complex operations, a majority of which are common across multiple systems. Many standard tools or libraries exist to perform these operations. For example, most network-monitoring systems can use the Scapy [115] for parsing packet fields. Building a system that implements all these functionalities from scratch is wasteful. Thus, designing a modular system, which divides the system into independent modules, ensures that it can leverage the existing tools to expedite system development. Also, networking technologies and protocols evolve rapidly with time. Designing a system tied to a specific technology or protocol inhibit its evolvability. Thus designing an extensible system that can swiftly adapt to these changes over time ensures that the system can evolve with time.

In this dissertation, we designed and implemented modular and extensive systems for network control and monitoring. The modular design helped identify operations that can be offloaded to existing open-source software tools. For example, SDX's em bgp-handler module used an existing open-source tool ExaBGP [31] to handle BGP messages, and Sonata's *emitter* modules used Scapy [115] to parse packets. Extensible design ensured that the systems could evolve with time. For example, the capabilities of programmable switches in terms of their support for flexible bitmask matching on destination MAC address field evolved between the year 2013 and 2015. This development inspired our new hierarchical attribute-encoding algorithm, capable of

leveraging the new flexible bitmask matching capability of the switch to improve the scalability of the SDX system further. An extensible design helped us augment the *attribute-encoding* module without requiring any changes to other modules, enabling SDX to evolve with time.

### 1.4.3 Selecting Strategic Locations for Deployment

Different network management systems are best suited for deployments at different locations. Selecting the right locations for deployment is critical for the system's real-world adoption. This task entails identifying the locations where the system can be incrementally deployed to perform the desired tasks while incurring minimal deployment overhead, and where the new system can incentivize the stakeholders by improving state of the art.

In this dissertation, we demonstrate that selecting IXPs for enabling flexible traffic control ensured incremental deployability. Enabling flexible traffic control at IXPs benefits hundreds of networks either physically or remotely present at the IXPs. For example, more than 800 networks are connected to DE-CIX at Frankfurt. Also, as existing IXPs (e.g., LINX, AMS-IX, DE-CIX, etc.) are at the frontline of peering tussles between the content providers (e.g., Google, Netflix, Facebook, etc.) and eyeball ISPs (e.g., AT&T, Verizon, etc.), the network operators at these IXPs are receptive to the idea of flexible traffic control at IXPs. As a result, our solution, iSDX, became the de-facto system for the Endeavour platform in Europe [29, 109], which led to a couple of trial deployments across Europe. We expect to see the deployment of SDXs in production settings soon.

## 1.5 Dissertation Outline

Chapter 2 presents how we use programmable switches to build Sonata, a scalable and flexible network-monitoring system. In Chapter 3, we detail the design of SDX, a novel scalable and flexible network-control system that can be deployed at large IXPs. In Chapter 4, we show how we make the best use of new programmable switches to develop the industry-scale network-control system iSDX. We describe how changes in programmable switch technologies enabled us to develop a more efficient attribute-encoding algorithm and a more modular prototype deployable in production settings. Chapter 5 concludes the dissertation by summarizing the contributions and illustrating how the lessons learned from this research can guide the design and implementation of next-generation intelligent network management systems that can run the networks by themselves without requiring any human intervention.

# Chapter 2

# Network Monitoring with Sonata

This chapter introduces the design, implementation, and evaluation of the network-monitoring system, Sonata, that lets network operators express monitoring tasks as dataflow queries over a stream of packet tuples. It opportunistically uses both programmable switches and stream processor to scale query execution with the number of queries and traffic volume.

## 2.1 Overview

Network operators routinely perform continuous monitoring to track events ranging from performance impairments to attacks. This monitoring requires continuous, real-time measurement and analysis—a process commonly referred to as *network monitoring* [125]. Existing monitoring systems can collect and analyze measurement data in real time, but they either support a limited set of monitoring tasks [76, 90], or they incur substantial processing and storage costs as traffic rates and the number of queries increase [23, 129, 13].

Existing monitoring systems typically trade off scalability for flexibility, or vice versa. Monitoring systems that rely on stream processors alone are flexible but not scalable. For example, systems such as NetQRE [129] and OpenSOC [90] can support

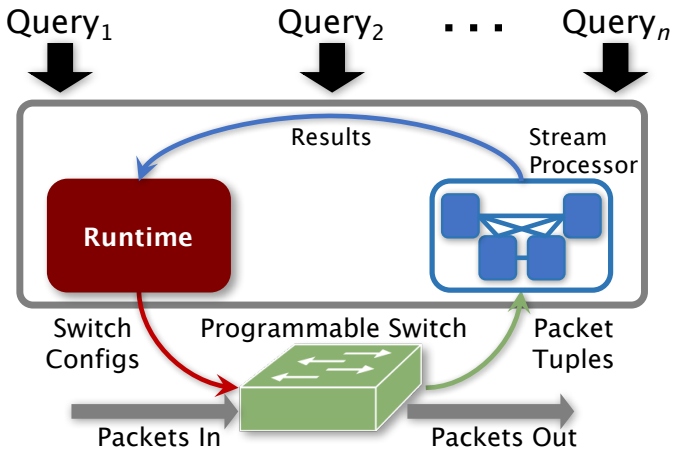


Figure 2.1: Sonata's Architecture.

a wide range of queries using stream processors running on general-purpose CPUs, but they incur substantial bandwidth and processing costs to do so. Large networks can require performing as many as 100 million operations per second for rates of 1 Tbps and packet sizes of 1 KB. Scaling to these rates using modern stream processors is prohibitively costly due to their low (2–3 orders of magnitude lower than the line rate) processing capacity per core [85, 87, 130, 91]. On the other hand, monitoring systems that rely on programmable switches alone can scale to high traffic rates, but they give up flexibility to achieve this scalability. For example, Marple [76] and OpenSketch [126], can perform monitoring tasks by executing queries solely in the data plane at line rate, but the queries that they can support are limited by the capabilities and memory in the data plane.

Rather than accepting this apparent tradeoff between flexibility and scalability, we observe that stream processors and programmable switches share a common processing model; they both apply an ordered set of transformations over structured data in a pipeline. This commonality suggests that an opportunity exists to combine the strengths of both technologies in a single monitoring system that supports flexible queries, while still operating at line rate for high traffic volumes and large numbers of queries.

To explore this idea, we develop *Sonata* (Streaming Network Traffic Analysis), a *query-driven* network-monitoring system. Figure 2.1 shows the design of Sonata: it provides a declarative interface that can express queries for a wide range of monitoring tasks and also frees the network operator from reasoning about where or how the query will execute. To scale query execution, Sonata (1) makes use of both programmable data-plane targets and scalable stream processors and (2) iteratively zooms-in on subsets of the traffic that satisfy the query—making the best use of limited data-plane resources. By unifying stream processing and data-plane capabilities, Sonata's runtime can refine query execution in the data plane to reduce load on the stream processor. This ability to dynamically refine queries is important because monitoring queries often require finding "needles in a haystack" where the fraction of the total traffic or flows that satisfies these queries is tiny. We present the following contributions:

**Unified query interface.** We design a query interface that unifies the parsing and compute capabilities of a programmable switch with those of stream processors. This interface allows network operators to apply familiar dataflow operators (e.g., map, filter, reduce) over arbitrary combinations of packet fields without regard for where the query will execute. We show that a wide-range of network-monitoring tasks can be expressed in fewer than 20 lines of code (Table 2.5).

**Query partitioning based on data-plane constraints.** To reduce load on the stream processor, we design an algorithm that partitions queries between the switch and the stream processor. We first show how dataflow queries can be partitioned without compromising accuracy. However, data-plane resources, such as switch memory and processing stages, are quite limited and inelastic. To make the best use of these resources, we develop an accurate model of common data-plane resource constraints and show how high-level dataflow operators consume these resources. Sonata's query

planner uses this model to decide how to partition query execution between the switch and stream processor.

**Dynamic query refinement based on query and workload.** To efficiently use limited switch compute and memory resources, we develop a dynamic refinement algorithm that, rather than processing all traffic, allows the switch to focus only on subsets of traffic that actually satisfy a query. We show that this technique applies to a wide range of monitoring queries and demonstrate how Sonata's query planner considers the structure of queries and representative traffic traces to compute a refinement plan for each query.

**Modular and extensible software architecture.** To support different types of data-plane and streaming targets, we design Sonata so that it could be extended to support operations over arbitrary packet fields. The queries expressed using the Sonata interface are agnostic to the underlying switch and streaming targets. Our current prototype implements drivers for both hardware [112] and software [114] protocol-independent switches (e.g., Barefoot Tofino [112]) as well as the Spark Streaming [111] stream processor. The current prototype parses packet headers for standard protocols and can be extended to extract other information, such as queue size along a path [46].

The Sonata prototype consists of about $9,000$ lines of code and it currently compiles queries to a single programmable switch. We use real packet traces from operational networks to demonstrate that Sonata's query planner reduces the load on the stream processor by as much as *seven orders of magnitude* over existing monitoring systems. We also quantify how Sonata's performance gains depend on data-plane constraints and traffic dynamics. To date, our open-source software prototype has been used by both researchers at a large ISP and in a graduate networking course.

We first review the state of the art in network monitoring in Section 2.2, and then present Sonata's programming abstraction in Section 2.3. We show how Sonata makes the best use of programmable PISA switch in Section 2.4, and how it further reduces the memory footprint for stateful dataflow operators using dynamic refinement in Section 2.5. We present the design and implementation of Sonata in Section 2.6, and its evaluation in Section 2.6. Finally, we survey the related work in Section 2.8.

## 2.2 Background and Motivation

In this section we review the state of the art in network monitoring. Whereas Sonata uses queries to *jointly* perform data collection and analysis, existing network monitoring systems primarily tackle *either* collection or analysis, with analysis typically occurring only after collection. This section surveys the state of the art in collection and analysis separately.

### Traffic Collection

Network traffic collection and monitoring falls into two classes: packet-level monitoring (sometimes referred to as "deep packet inspection") and flow-level monitoring.

**Packet monitoring** Packet-level monitoring can be performed with software libraries such as `libpcap`, or in hardware, using devices such as the Eagle 10 or Endace capture cards [28]. Commonly, collection infrastructure is deployed on a switch span port, which mirrors traffic going through the switch. A device connected to the span port—typically a server—captures and stores the mirrored traffic. The collection infrastructure can be configured with filters that can specify conditions for capturing traffic; configuration can also determine whether complete packet payloads are captured, or simply an excerpt of the packet, such as packet headers. Packet-level monitoring can provide precise information for calculating statistics like the

instantaneous bitrate, packet loss, or round-trip latency experienced by individual flows. Access to packet payloads can also be useful for a variety of purposes, such as determining whether any given packet carries a malicious payload.

Unfortunately, packet-level monitoring has significant drawbacks, due to the high overhead of collection, storage, and analysis. The sheer volume of network traffic makes it prohibitive to capture every packet. Even if the infrastructure could capture every packet, operators face daunting storage and analysis hurdles associated with storing a complete log of all network traffic. As such, despite the rich possibilities that packet-level traffic capture offer, many networks do not deploy this type of infrastructure on a widespread basis. For example, recent figures from a large access ISP have indicated that deep-packet inspection capabilities are deployed for less than 10% of the network capacity. This sparse deployment makes it essentially prohibitive to generally perform the types of queries involving network performance or security that could benefit from packet-level statistics.

**Flow monitoring** An alternative to packet-level monitoring is flow-level monitoring—standardized in the Internet Engineering Task Force (IETF) as IPFIX, and commonly referred to by the Cisco "NetFlow" moniker. IPFIX permits each switch to collect flow-level statistics that contain coarse-grained information such as the number of packets and bytes for a particular flow (e.g., as defined by the source and destination IP address, source and destination port, and protocol), as well as the start and end time of the flow. This type of information is often gathered in a "sampled" fashion: on average, one out of every $n$ packets is tabulated in an IPFIX flow record; typical sampling rates for an ISP backbone network can be in the $1{,}000 < n < 10{,}000$ range, meaning that low-volume flows may often not be captured at all. Additionally, IPFIX records do not contain detailed information about flows, such as packet loss rates or packet timings, let alone packet payload information.

Both packet-level and flow-level monitoring systems can, of course, be tailored to capture specific subsets of traffic. Packet-level monitoring can be customized with filters that focus on specific subsets of traffic, and flow-level monitoring can be tuned so that sampling rates are higher for specific links of interest. The advent of programmable data planes has also enabled programmatic collection of individual data flows [101, 65, 126, 132, 108, 129, 89]. Yet, these tools either support limited sets of queries that can only operate over fixed packet headers (e.g., UnivMon [65] and OpenSketch [126]) or require custom tools to analyze one specific type of data (e.g., BigTap [101] and PLT [89]) precluding any analysis that requires fusing multiple data streams.

Furthermore, the level of flexibility that all of these systems offers is limited in the sense that (1) in general, their configurations remain static; (2) decisions about capturing more fine-grained information are completely decoupled from the queries or analysis. In short, because the monitoring process is decoupled from analysis, all of these decisions must be made far in advance of analysis, often resulting in traffic collection that is either too sparse or too voluminous.

### Traffic Analysis

Given the ability to perform packet or flow monitoring on network traffic, network operators can use systems such as Deepfield [88], Kentic [113], or Velocidata [118] to perform network analysis in support of network performance or security. For example, Deepfield Singularity performs joint analysis of packet captures and IPFIX records to help network operators understand questions such as the relationship between traffic overload and application performance as well as detect network attacks such as distributed denial of service attacks. For example, analyzing the average bitrate of Netflix streams traversing the network requires: (1) capturing the DNS queries (and responses) for DNS domains corresponding to Netflix streams; (2) joining the resulting

IP addresses to the corresponding traffic data (e.g., either IPFIX or packet capture) that can provide information about the rates that individual flows are seeing. Another example might be the detection of a DNS reflection and amplification attack, which involves compromised hosts sending large volumes of DNS queries with the spoofed source IP address of the victim. Detecting such attacks often involves detecting an abnormally high number of DNS queries from an IP address (in this case, the IP address of the victim), typically for DNS query types that elicit large responses (e.g., `TXT`, `RRSIG`); alternatively, one could look for an abnormally large number of such responses destined for a given IP address.

Although existing technologies developed by Deepfield and Kentik support certain aspects of this type of analysis, they do not use the query itself to drive collection of the traffic data, which often results in collecting, storing and analyzing large volumes of data that do not pertain to the specific queries. Specifically, these analysis tools rely on *separate* collection of DNS data (with packet monitoring) and traffic utilization information (e.g., with IPFIX), which the analysis tools subsequently joint *post hoc*. This approach to analysis also requires capturing a large amount of traffic that is not relevant to the analysis, which increases the overhead of the analysis, both in terms of the volume of data and the computation time. Furthermore, because IPFIX data is often based on sampled traffic traces with high sampling rates, many DNS queries and responses will not be captured in the IPFIX data at all, severely compromising accuracy. Finally, the *post hoc* nature of existing analysis tools also precludes real-time detection, since all data is collected and warehoused for subsequent joint analysis.

## 2.3 Unified Query Interface

This section presents Sonata's query interface and shows example queries to illustrate the types of queries that existing systems can and cannot support. Sonata provides a

query interface that is as flexible as modern stream processors but opportunistically achieves the scalability of data-plane execution. Although Sonata uses programmable switches to scale query execution, the computational limitations of these switches do *not* affect the flexibility of the query interface. Sonata can execute operations that are not supported in the switch, such as ones requiring payload processing or floating-point operations, in the stream processor.

### 2.3.1 Dataflow Queries on Tuples

**Extensible tuple abstraction.** Information in packet headers naturally constitute key-value tuples (e.g., source and destination IP address, and other header values). This structure lends itself to a tuple-based abstraction [38]. Of course, an operator may want to write queries based on information that is not in the IP packet header, such as the application protocol, or DNS query type. To facilitate a wider range of queries, Sonata allows an operator to extend the tuple interface to include other fields that could be extracted by programmable switch or stream processor. For example, they can specify customized packet-parsing operations for programmable switches in a language such as P4. Based on such a specification, the parser can extract all portions of the packet that pertain to the query. Sonata parses tuples on the switch whenever possible, shunting packets to the stream processor only when the query requires sophisticated parsing or join operations that the switch itself cannot support (e.g., parsing of the packet's payload), or information that requires a join with auxiliary information such as a routing table.

**Flexible dataflow operators.** Many network-monitoring queries require computing aggregate statistics over a subset of traffic and joining the results from multiple queries, which can be expressed as a sequential composition of dataflow operators (e.g., filter, map, reduce). Gigascope [23], Chimera [13], and Marple [76] all use such a programming model, which is both familiar and amenable for compilation to pro-

grammable switches [76]. Section 2.4 describes how Sonata compiles queries across the stream processor and switch. Table 2.1 summarizes Sonata's dataflow operators. Stateful dataflow operators are all executed with respect to a query-defined time interval, or window. For example, applying `reduce` over a sum operation will return a result at the end of each window. Each query can explicitly specify the interval's duration for stateful operations.

| **Operator** | **Description** |
|---|---|
| filter($p$) | Filter packets that satisfy predicate $p$. |
| map($f$) | Transform each tuple with function $f$. |
| distinct() | Emit tuples with unique combinations of fields. |
| reduce($k$, $f$) | Emit result of function $f$ applied on key $k$ over the input stream. |
| join($k$, $q$) | Join the output of query $q$ on key field $k$ |

Table 2.1: Sonata's Dataflow Operators. All stateful operators execute with respect to a window interval of $W$ seconds.

**Limitations.** Sonata queries operate at packet-level granularity, as in existing declarative monitoring systems [23, 13, 90, 76]; it cannot support queries that require reassembling a byte stream, as in Bro. Sonata also currently compiles each query to a single switch, *not* across multiple switches. The set of single-switch queries we present are still practical for many deployments, such as on a border switch or at an Internet exchange point (IXP). We leave compiling arbitrary queries to multiple switches as future work.

### 2.3.2 Example Network-Monitoring Queries

We now present three example queries: one that executes entirely in the data plane, a second that involves a join of two sub-queries, and a third that requires parsing packet payloads. Table 2.5 summarizes the queries that we have implemented and released publicly along with the Sonata software [117].

**Computing aggregate statistics over a subset of traffic.** Suppose that an operator wants to detect hosts that have too many recently opened TCP connections,

```
1  packetStream(W)
2  .filter(p => p.tcp.flags == 2)
3  .map(p => (p.dIP, 1))
4  .reduce(keys=(dIP,), f=sum)
5  .filter((dIP, count) => count > Th)
```

Query 2.1: Detect Newly Opened TCP Connections.

as in a SYN flood attack. Detection requires parsing each packet's TCP flags and destination IP address, as well as computing a sum over the destination IP address field. Query 2.1 first applies a `filter` operation (line 2) over the entire packet stream to select TCP packets with just the `SYN` flag set. It then counts the number of packets it observed for each host (lines 3–4) and reports the hosts for which this count exceeds threshold `Th` at the end of the window (line 5). This query can be executed entirely on the switch, so existing systems (e.g., Marple [76]) can also execute this type of query at scale.

**Joining the results of two queries.** A more complex query involves joining the results from two sub-queries. To detect a Slowloris attack [102], a network operator must identify hosts which use many TCP connections, each with low traffic volume. This query (Query 2.2) consists of two sub-queries: The first counts the number of unique connections by applying a `distinct`, followed by a `reduce` (lines 1–6). The second counts the total bytes transferred for each host (lines 8–11). The query then joins the two results (line 7) to compute the average connections per byte (line 12) and reports hosts whose average number of connections per byte exceeds a threshold `Th2` (line 13). This query, as presented, is equivalent to detecting hosts for which average bytes per connection is *less than* a threshold; in Section 2.5, we discuss why it is desirable to express the query as the former instead of the latter. Because this query involves a join operation after an aggregation operation (reduce), Marple cannot support it. Additionally, the query involves computing an average, which requires performing division; even state-of-the-art programmable switches do not support this operation in the data plane [112]. Some queries will naturally need to perform opera-

```
packetStream
.filter(p => p.proto == TCP)
.map(p => (p.dIP,p.sIP,p.tcp.sPort))
.distinct()
.map((dIP,sIP,sPort) =>(dIP,1))
.reduce(keys=(dIP,), f=sum)
.join(keys=(dIP,), packetStream
  .filter(p => p.proto == TCP)
  .map(p => (p.dIP,p.pktlen))
  .reduce(keys=(dIP,), f=sum)
  .filter((dIP, bytes) => bytes > Th1) )
.map((dIP,(byte,con)) => (dIP,(con/byte))
.filter((dIP, con/byte) => (con/byte > Th2)
```

Query 2.2: Detect Slowloris Attacks.

```
packetStream
.filter(p => p.tcp.dPort == 23)
.join(keys=(dIP,), packetStream
  .filter(p => p.tcp.dPort == 23)
  .map(p => ((p.dIP,p.nBytes/N), 1))
  .reduce(keys=(dIP, nBytes), f=sum)
  .filter(((dIP,nBytes),cnt1) => cnt1 > Th1))
.filter(p => p.payload.contains('zorro'))
.map(p => (p.dIP,1))
.reduce(keys=(dIP,), f=sum)
.filter((dIP, count2) => count2 > Th2)
```

Query 2.3: Detect Zorro Attacks.

tions that are more sophisticated than the switch can support; in such cases, existing approaches cannot execute these kinds of queries at all. In contrast, Sonata's query planner can partition the query for partial execution on the switch and perform more sophisticated computation not available in the switch at stream processor.

**Processing packet payloads.** Consider a query to detect the spread of malware via telnet [80], which is a common tactic targeting IoT devices [4]. Here, miscreants use brute force to gain shell access to vulnerable Internet-connected devices. Upon successful login, they issue a sequence of shell commands, one of which contains the keyword "zorro". The query to detect these attacks first looks for hosts that receive many similar-sized telnet packets followed by a telnet packet with a payload containing the keyword "zorro". The query for this task has two sub-queries (Query 2.3): the first part identifies hosts that receive more than `Th1` similar-sized telnet packets

rounded off by a factor of `N` (lines 4–7). The second part joins (line 3) the output of the first sub-query with the other and reports hosts that receive more than `Th2` packets and contain the keyword "zorro" in the payload (lines 8–11). Since this query requires parsing packet payloads, existing approaches cannot support it. In contrast, Sonata can support and scale these queries by performing as much computation as possible on the switch and then performing the rest at the stream processor.

## 2.4 Query Partitioning

Sonata partitions a given query across a stream processor and a protocol-independent switch that performs part of the query, ultimately reducing the load (Section 2.4.1) on the stream processor. Section 2.4.2 discusses the constraints of these switches that the Sonata query planner considers; the planner solves an optimization problem to partition the query, as described in Section 2.4.3.

### 2.4.1 Data Reduction on the Switch

A central contribution of Sonata is to use the capabilities of programmable switches to reduce the load on the stream processor. In contrast to conventional switches, protocol-independent switch architecture (PISA) switches (e.g., RMT [15], Barefoot Tofino [50], Netronome [120]) offer programmable parsing and customizable packet-processing pipelines, as well as general-purpose registers for stateful operations. These features provide opportunities for Sonata to perform more of the query on the switch, reducing the amount of data sent to the stream processor.

#### Abstract Packet Processing Model

Figure 2.2 shows how Query 2.1 naturally maps to the capabilities of the packet processing model of a PISA switch. On PISA switches, a reconfigurable parser constructs

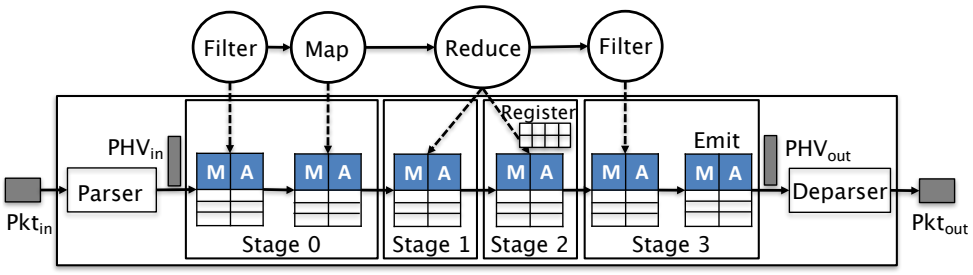



Figure 2.2: Compiling a dataflow query (Query 2.1) to a sequence of match-action tables for a PISA switch. Each query consists of an ordered sequence of dataflow operators, which are then mapped to match-action tables in the data plane.

a packet header vector (PHV) for each incoming packet. The PHV contains not only fixed-size standard packet headers but also custom metadata for additional information such as queue size. A fixed number of physical stages, each containing one match-action unit (MAU), then processes the PHVs. The packet processing pipeline is a sequence of custom match-action tables; MAUs implement these abstract tables in hardware. Each MAU performs a self-contained set of match-action operations, consuming PHVs as input and emitting transformed PHVs as output. If fields in the PHV match a given rule in the MAU, then a set of custom actions corresponding to that rule are applied to the PHV. These actions can be stateless or stateful; the stateful operations use register memory to maintain state. Finally, a deparser reassembles the modified PHV into a packet before sending it to an output port.

The PISA processing model aligns well with streaming analytics platforms such as Spark Streaming [130] or Apache Flink [86]. The processing pipelines for both can be represented as a directed, acyclic graph (DAG) where each node in the graph performs some computation on an incoming stream of structured data. For stream processors, the nodes in the DAG are dataflow operators and the structured-data stream consists of tuples. For PISA switches, the nodes in the DAG are match-action tables and the structured-data stream consists of packets. Given this inherent similarity, an ordered set of dataflow query operators could map to an ordered set of match-action tables

in the data plane. We now describe how Sonata takes advantage of this similarity to execute dataflow operators directly in the data plane.

### Compiling Individual Operators

Compiling dataflow queries to PISA switches requires translating the DAG of dataflow operators into an equivalent DAG of match-action tables. Prior work [76] also faced the challenge of compiling high-level queries to match-action tables, but limited the set of input queries to those that can be performed entirely on the switch. Rather than constraining the set of input queries, Sonata's query planner partitions all input queries into a set of dataflow operators than can be executed on the switch and a set that must be executed at the stream processor. Before Sonata's query planner can make this partitioning decision, we must first quantify which limited resources are consumed by executing individual dataflow operators on the switch.

**Filter** requires a single match-action table to match a set of fields in the PHV. For example, line 1 of Query 2.1 requires a single match-action table where the six-bit `tcp.flags` field is a column and the value 2 is a single rule (row), as shown in Figure 2.2. In general, the match-action table for a `filter` operation has a column for each field in the predicate. A filter predicate with multiple clauses connected by "and" leads to multiple rules, one per clause.

**Map** also requires a single match-action table. For example, line 2 of Query 2.1 transforms all incoming packets into a tuple consisting of the `ipv4.dIP` field from the packet's header and the value `1`. These values are stored in query-specific metadata for further processing. Although Sonata's query interface does not constrain the transformations that `map` might perform over a set of tuples, the operator cannot be compiled to the data plane if the switch cannot perform the corresponding transformation.

**Reduce** requires maintaining state across sequences of packets; Sonata uses registers, which are simply arrays of values indexed by some key, to do so. Query-specific metadata fields permit loading and storing values from the registers. As a result, stateful operations require two match-action tables: one for computing the index of the value stored in the register and the other for updating state using arithmetic operators supported by the switch, such as `add` and `bit_or`. A corresponding metadata field carries the updated state after applying the arithmetic operation. For example, executing the `reduce` operator for Query 2.1 in Figure 2.2 requires a match-action table to compute an index into the register using the `dIP` header field. A second table performs the stateful action that increments the indexed value in the register and stores the updated value. In Section 2.4.3, we describe how Sonata's query planner uses representative training data to configure the number of entries for each register.

**Distinct** operations are similar to a `reduce`, where the function `bit_or 1` is applied to a single bit.

**Join** operations are costly to execute in the data plane[1]. In the worst case, this operation maintains state that grows with the square of the number of packets. Sonata executes `join` operations at the stream processor by iteratively dividing the query into a set of sub-queries. For example, Sonata divides Query 2.2 into two sub-queries: one that computes the number of unique connections, and a second that computes the number of bytes transferred for each host. Sonata independently decides how to execute the two sub-queries and ultimately joins their results at the stream processor.

[1]Marple's `zip` operator [76], which can be executed in the data plane, is a restricted version of Sonata's `join` operator and unlike Sonata, cannot be used to join the results of two separate analyses together.

### Compiling Dataflow Queries

In addition to mapping individual dataflow operators to match-action tables, the resulting data-plane mapping must be synthesized in a way that respects the following additional considerations.

**Preserving packet forwarding decisions.** Sonata preserves packet forwarding decisions by transforming only query-specific metadata fields, rather than the packet contents that might affect forwarding decisions (e.g., destination address, application headers, or even payload). The switch extracts values from the packets' original header fields and copies them to auxiliary metadata fields before performing any additional processing. This process leaves the original packet unmodified.

**Reporting intermediate results to the stream processor.** When a query is partitioned across the stream processor and the switch, the stream processor may need either the original packet or just an intermediate result from the switch, so that it can perform its portion of the query. To facilitate this reporting, the switch maintains a one-bit `report` field in metadata for each packet. Each query partitioned to the switch marks this field whenever a query-specific condition is met that requires a packet be sent to the stream processor. If this field is set at the conclusion of the entire processing pipeline, the switch sends to the stream processor all intermediate results needed to complete processing the query, including the original packet if needed by the query. If the last operator is stateful (e.g., `reduce`), then the switch sends only one packet for each key to the stream processor. This informs the stream processor which register indices in the data plane must be polled at the end of each window to retrieve aggregated values stored in the switch (see Section 2.6 for details).

**Detecting and mitigating hash collisions.** Sonata must detect and mitigate hash collisions that may result at the switch. The probability of a hash collision is proportional to the number of hashes performed on unique keys and the size of the output hash as a consequence of the the pigeonhole principle. In theory, a 32-bit

hash has a 50% chance of a collision after hashing fewer than 80,000 keys. Since true hash-tables with collision resolution are not available on the switch, we instead use registers with hash-based indices. In practice, these registers contain far fewer rows than the number of unique values in the hash output, making collisions even more likely. To detect collisions, switches store the original key when performing `reduce` and `distinct` operations. To mitigate collisions, Sonata uses a sequence of up to $d$ registers, each using a different hash function for determining indices. If a key generates a collision, Sonata iterates through each of the $d$ registers, storing the key in the first register that does not result in a collision. If after iterating through all $d$ registers, the key still generates a collision, Sonata sends the packet to the stream processor. At the end of each window, the stream processor adjusts the results received from the switch with the additional packets processed due to collisions.

### 2.4.2 Data-Plane Resource Constraints

Sonata's query planner must consider the finite resource constraints of PISA switches for parsing packet header fields, performing actions on packets, storing state in register memory and performing all of these operations in a limited number of stages.

**Parser.** The cost of parsing increases with the number of fields to extract from the packet. This cost is quantified as the number of bits to extract and the depth of the parsing tree. The size of the PHV limits the number of fields that can be extracted for processing. Typically, PISA switches have PHVs about 0.5–8 Kb [15] in size. $M$ represents the maximum storage for metadata in the PHV.

**Actions.** Most stream processors execute multiple queries in parallel, where each query operates over its own logical copy of the input tuple. In contrast, PISA switches transform raw packets to PHVs and then concurrently apply multiple operations over the PHV in pipelined stages. These mechanisms suggest that PISA switches would

be amenable to parallel query execution. In practice, there is a limit on how many actions can be applied over a PHV in one stage, which limits the number of queries that can be supported in the data plane. Typically, PISA switches support 100–200 stateless and 1–32 stateful actions per stage [15]; we represent the maximum number of stateful actions per stage as $A$.

**Registers.** The amount of memory required to perform stateful operations grows with the number of packets and the number of queries. Stream processors scale by adding more nodes for maintaining additional state. In contrast, stateful operations in PISA switches can only access register memory locally available to their physical stage. This register memory is bounded for each stage, which affects the switch's ability to handle both increased traffic loads and additional queries. Within a stage, the amount of memory available to a single register is also bounded. Typically, PISA switches support 0.5–32 Mb memory for each stage [15]. $B$ represents the maximum number of register bits available in each stage.

**Stages.** Queries that lack available resources in a given stage must execute in a later stage. PISA switches typically support 1–32 physical stages [15]; we represent the maximum number of stages as $S$.

### 2.4.3 Computing Query-Partitioning Plans

Consider a switch with $S = 4$ stages, $B = 3,000$ Kb, and $A = 4$ stateful actions per stage. These constraints are more strict than Barefoot's Tofino switch [112], but they illustrate how the data-plane resource constraints affect query planning. Sonata runs Query 2.1 over a one-minute packet trace from CAIDA [24] to compute that the switch requires $2,500$ Kb to count the number of TCP SYN packets per host (Figure 2.5). Since $2,500$ Kb $< B$, Sonata can execute the entire query on the switch, sending only the 77 tuples that satisfy the query to the stream processor. If $B$ or $S$ were smaller, Sonata could not execute the `reduce` operator on the switch and would need

| | |
|---|---|
| **Switch Constraints** | |
| $M$ | Amount of metadata stored in switch. |
| $A$ | Number of stateful actions per stage. |
| $B$ | Register memory (in bits) per stage. |
| $S$ | Number of stages in match-action pipeline. |
| **Input from Queries** | |
| $O_q$ | Ordered set of dataflow operators for query $q$. |
| $T_q$ | Ordered set of match-action tables for query $q$. |
| $M_q$ | Amount of metadata required to perform query $q$. |
| $Z_t$ | Indicates whether table $t$ performs a stateful operation. |
| **Input from Workload** | |
| $N_{q,t}$ | Number of packets generated after table $t$ of query $q$. |
| $B_{q,t}$ | State (bits) required for executing table $t$ of query $q$. |
| **Output** | |
| $P_{q,t}$ | Indicates whether $t$ is the last table partitioned to the switch for query $q$. |
| $X_{q,t,s}$ | Indicates whether table $t$ of query $q$ executes at stage $s$ in the switch. |
| $S_{q,t}$ | Stage id for table $t$ for query $q$. |

Table 2.2: Summary of variables in the query-planning problem.

to partition the query. The rest of this section describes how Sonata computes such query plans.

Sonata's query planner solves an Integer Linear Program (ILP) that minimizes the number of packet tuples sent to the stream processor based on a partitioning plan, subject to switch constraints, as summarized in Table 2.3. Our approach is inspired by previous work on a different problem that partitions multiple logical tables across physical tables [54]. Table 2.2 summarizes the variables in the query planning problem. To select a partitioning plan, the query planner determines the capabilities of the underlying switch, estimates the data-plane resources needed to execute individual queries, and estimates the number of packets sent to the stream processor given a partitioning of operators on the switch.

**Input.** For the set of input queries ($Q$), Sonata interacts with the switch to compile the ordered set of dataflow operators in each query ($O_q$) to an ordered set of match-action tables ($T_q$) that implement the operators on the switch. $Z_t$ indicates to the

**Goal**

$$min(N = \sum_{q} \sum_{t} P_{q,t} \cdot N_{q,t})$$

**Constraints**

$$C1: \quad \forall s: \sum_{q} \sum_{T_q} X_{q,t,s} \cdot B_{q,t} \leq B$$

$$C2: \quad \forall s: \sum_{q} \sum_{T_q} Z_t \cdot X_{q,t,s} \leq A$$

$$C3: \quad \forall q, t: S_{q,t} < S$$

$$C4: \quad \forall q, i < j, i, j \in T_q: S_{q,j} > S_{q,i}$$

$$C5: \quad \forall q: \sum_{q} M_q \leq M$$

Table 2.3: ILP formulation for the query partitioning problem.

query planner whether the table has stateful or stateless operators. Using historical traffic traces, the query planner estimates the number of packet tuples ($N_{q,t}$) sent to the stream processor and the amount of state ($B_{q,t}$) required to execute table $t$ for query $q$ on the switch.

**Objective.** The objective of Sonata's query planning ILP is to minimize the number of packets processed by the stream processor. The query planner models this objective by introducing a binary decision variable $P_{q,t}$ that captures the partitioning decision for each query; $P_{q,t} = 1$ one if $t$ is the last table for query $q$ that is executed on the switch. For each query, only one table corresponding to one operator can be set as the last table on the switch: $\sum_{T_q} P_{q,t} \leq 1$. The total number of packets processed by the stream processor is then the sum of all packets emitted by the last table processed on the switch for all queries.

**Switch constraints.** To ensure that Sonata respects the constraints from Section 2.4.2, we introduce variables $X$ and $S$. $X_{q,t,s}$ is a binary variable that reflects stage assignment: $X_{q,t,s} = 1$ only if table $t$ for query $q$ executes at stage $s$ in the match-action pipeline. Similarly, $S_{q,t}$ returns the stage number where table $t$ for query $q$ is executed. These two variables are related: if $X_{q,t,s} = 1$, then $S_{q,t} = s$ for

a given stage. We will now summarize how Sonata's query planner models various data-plane constraints.

*C1: Register Memory per stage (B).* For each stage, the amount of state allocated for Sonata's packet processing cannot exceed $B$. Since PISA targets can only configure tables with stateful operations in a single stage, the amount of state required to execute query $q$ at stage $s$ is $\sum_{T_q} X_{q,t,s} \cdot B_{q,t}$. This sum over all queries captures the total memory required for each stage $s$.

*C2: Number of Actions per stage (A).* For each stage, the total number of stateful operations cannot exceed $A$. We can again use the $X$ variable to model this constraint. The expression $\sum_{T_q} Z_t \cdot X_{q,t,s}$ captures the number of stateful operations performed at stage $s$ for query $q$. This sum over all queries captures the total number of stateful actions for each stage $s$.

*C3: Number of Stages (S).* The total number of stages required to execute a query in the data plane cannot exceed $S$. The variable $S_{q,t}$ represents the stage where table $t$ for query $q$ is executed. For every table of each query, this variable should always be less than $S$ because the last stage is reserved to determine which packet needs to be reported to the stream processor.

*C4: Intra-Query Ordering.* We can also use $S$ to express intra-query ordering constraints. For example, in the Slowloris query (Query 2.2), the tables for the `reduce` operator can only be executed after the `distinct` operator has been applied in a previous stage. For each query $q$ and any two indices $(i, j)$ in the ordered set of tables $T_q$ where $(i < j)$, $S_{q,j}$ is always greater than $S_{q,i}$.

*C5: Total Metadata (M).* Finally, since the PHV consists of a fixed-size, $(M)$ represents the maximum space available in the PHV to add query-specific metadata fields. The total metadata used for all queries must then be less than $M$, *i.e.* $\sum_q M_q \leq M$.

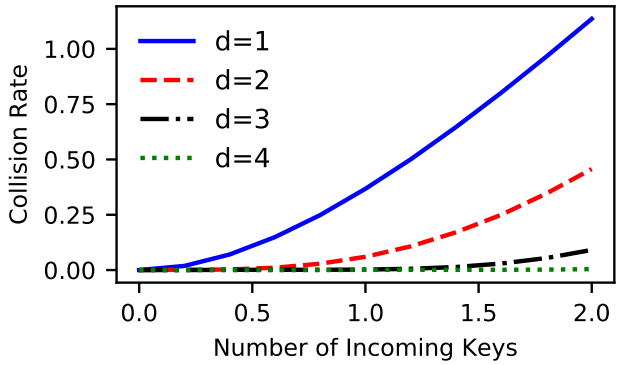



Figure 2.3: Relationship between collision rate and number of unique incoming keys.

**Monitoring traffic dynamics.** The query planner uses training data to decide how to configure the number of entries ($n$) for each register, and how many registers ($d$) to use for each stateful operation. It is possible that the training data might underestimate the number of expected keys ($k$) for a stateful operation due to variations in traffic patterns. In Figure 2.3, we show how the collision rates increase as the number of unique keys grows beyond the original estimate ($n$) for a sequence of ($d$) registers. Here, the x-axis is the number of incoming keys and y-axis is the collision rate—both normalized with respect to $n$. The collision rate increases as the number of incoming keys increases and decreases as $d$ increases.

Since collision rates are predictable, we choose values of ($n$) and ($d$) to keep collision rates low but still high enough to send a signal to Sonata's runtime when the switch is storing many more unique keys than originally expected. Both of the costs (packets to processor, and memory on the switch) required to handle collisions and monitor traffic dynamics are captured in the ILP formulation.

## 2.5 Algorithm: Dynamic Query Refinement

For certain queries and workloads, partitioning a subset of dataflow operators to the switch does not reduce the workload on the stream processor enough; in these situations, Sonata dynamically refines the query.

To do so, Sonata's query planner modifies the input queries to start at a coarser level of granularity than specified in the original query (Section 2.5.1). It then chooses a sequence of finer granularities that reduces the load on the stream processor. This process introduces additional delay in detecting the traffic that satisfies the input queries. The specific levels of granularity chosen and the sequence in which they are applied constitute a *refinement plan*. To compute an optimal refinement plan for the set of input queries, Sonata's query planner estimates the cost of executing different refinement plans based on historical training data. Sonata's query planner then solves an extended version of the ILP from Section 2.4.3 that determines both partitioning as well as refinement plans to minimize the workload on the stream processor (Section 2.5.2).

### 2.5.1 Modifying Queries for Refinement

**Identifying refinement keys.** A refinement key is a field that has a hierarchical structure and is used as a key in a stateful dataflow operation. The hierarchical structure allows Sonata to replace a more specific key with a less specific version without missing any traffic that satisfies the original query. This applies to all queries that filter on aggregated counts greater than a threshold. For example, `dIP` has a hierarchical structure and is used as a key for aggregation in Query 2.1. As a result, the query planner selects `dIP` as a refinement key for this query. Other fields that have hierarchical structure can also serve as refinement keys, such as `dns.rr.name` and `ipv6.dIP`. For example, a query for detecting malicious domains that requires

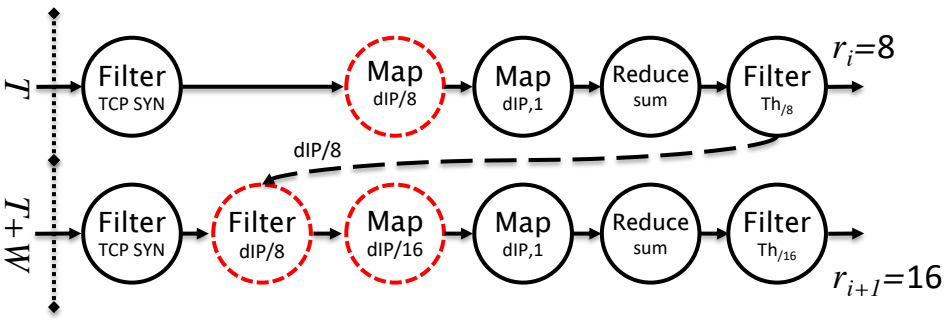



Figure 2.4: Query augmentation for Query 2.1. The query planner adds the operators shown in red to support refinement. Query 2.1 executes at refinement level $r_i$ = `/8` during window T and at level $r_{i+1}$ =`/16` during window $(T + W)$. The dashed arrow shows the output from level $r_i$ feeding a filter at level $r_{i+1}$.

counting the number of unique resolved IP address for each domain [11], can use the field `dns.rr.name` as a refinement key. Here, a fully-qualified domain name is the finest refinement level and the root domain (`.`) is the coarsest. A query can contain multiple candidate refinement keys and Sonata independently selects refinement keys for each query.

**Enumerating refinement levels.** After identifying candidate refinement keys, the query planner enumerates the possible levels of granularity for each key. Each refinement key consists of a set of levels $R = \{r_1 \ldots r_n\}$ where $r_1$ is the coarsest level and $r_n$ is the finest. The inequality $r_1 > r_n$ means that $r_1$ is coarser than $r_n$. The semantics of the $n^{th}$ refinement level is specific to each key; $n = 32$ would correspond to a `/32` IP prefix for the key `dIP` and $n = 2$ would correspond to second-level domain for the key `dns.rr.name`.

**Augmenting input queries.** To ensure that the finer refinement levels only consider the traffic that has already satisfied coarser ones, Sonata's query planner augments the input queries. For example, Figure 2.4 shows how it augments Query 2.1 with refinement key `dIP` and $R$ ={`8`, `16`, `32`} to execute the query at level $r_{i+1} = 16$ after executing it at level $r_i = 8$. The query planner first adds a `map` at each level to transform the original reduction key into a count bucket for the current refine-


| $r_i \longrightarrow r_{i+1}$ | $N_1$ | $B$ (Kb) | $N_2$ |
|---|---|---|---|
| $* \longrightarrow 32$ | 570K | 2,500 | 77 |
| $* \longrightarrow 16$ | | 180 | 99 |
| $* \longrightarrow 8$ | | 6 | 33 |
| $8 \longrightarrow 32$ | 526K | 1,900 | 77 |
| $8 \longrightarrow 16$ | | 50 | 98 |
| $16 \longrightarrow 32$ | 450K | 1,200 | 77 |

Figure 2.5: The $N$ and $B$ cost values for executing Query 2.1 at refinement level $r_{i+1}$ after executing it at level $r_i$.

ment level. For example, $r_i$ and $r_{i+1}$ rewrite `dIP` as `dIP/8` and `dIP/16`, respectively. By transforming the reduction key for each refinement level, the rest of the original query can remain unmodified. At refinement level $r_{i+1}$, the query planner also adds a `filter`. At the conclusion of the first time window, the runtime feeds as input to the `filter` operator the `dIP/8` addresses that satisfy the query at $r_i = 8$. This filtering ensures that refinement level $r_{i+1}$ only considers traffic that satisfies the query at $r_i$. Sonata's query planner also augments queries to increase the efficiency of executing refined queries. While using the original query's threshold values for coarser refinement levels would still be correct, Sonata's query planner instead uses training data to calculate relaxed threshold values for coarser refinement levels that do not sacrifice accuracy (e.g., $\texttt{Th}_{/8}$ and $\texttt{Th}_{/16}$ in Figure 2.4). Dynamic refinement is also appropriate for queries that require `join` operations (e.g., Query 2.2). The two sub-queries use the same refinement plan and their output at coarser levels determines which portion of traffic to process for the finer levels.

### 2.5.2 Computing Refinement Plans

**Dynamic query refinement example.** Sonata's query planner applies training data onto the the augmented queries to generate Figure 2.5 for Query 2.1. This figure shows the costs to execute Query 2.1 with refinement key `dIP` and refinement levels

$R$ ={8, 16, 32} over the training data. It shows the number of packets sent to the stream processor depending on which refinement level ($r_{i+1}$) is executed after level $r_i$. If only the `filter` operation is executed on the switch, then $N_1$ packets are sent to the stream processor. If the `reduce` operation is also executed on the switch, then $N_2$ packets are sent, but then $B$ bits of state must also be maintained in the data plane. For simplicity of exposition, we assume that these counts remain the same for three consecutive windows.

Consider an approach, *Fixed-Refinement*, that applies a fixed refinement plan for all input queries. In this example, the query planner augments the original queries to always run at refinement levels 8, 16, and 32. The runtime updates the filter for the query at level 16 with the output from level 8 and the filter of level 32 with the output from 16. The costs of this plan are shown in rows $* \rightarrow 8$, $8 \rightarrow 16$, and $16 \rightarrow 32$ of Figure 2.5. Because the switch only supports two stateful operations ($A = 2$), the `reduce` operator could only be performed on the switch for the first two refinement levels. This would result in sending 33 packets ($N_2$ for $* \rightarrow 8$) at the end of the first window, 98 packets ($N_2$ for $8 \rightarrow 16$) at the end of the second window, and $450,000$ ($N_1$ for $16 \rightarrow 32$) packets at the end of the third window to the stream processor. Compared to the solution without any refinement from beginning of Section 2.4.3, *Fixed-Refinement* reduces the number of tuples reported to the stream processor from 570 K to 450 K at the cost of delaying two additional time windows to detect traffic that satisfies the query.

In contrast, Sonata's query planner uses the costs in Figure 2.5 combined with the switch constraints to compute the refinement plan $* \rightarrow 8 \rightarrow 32$. Executing the query at refinement level $* \rightarrow 8$ requires only 6 Kb of state on the switch and sends 33 packet tuples to the stream processor at the end of the first window. Each packet represents an individual `dIP/8` prefix that satisfies the query in the first window. Sonata then applies the original input query (`dIP/32`) over these 33 `dIP/8` prefixes in the second

window interval, processing 526,000 packets ($N_1$ for `8` $\rightarrow$ `32`) and consuming only 1900 Kb on the switch. At the end of the second window, the switch reports 77 `dIP/32` addresses to the stream processor. This refinement plan sends 110 packet tuples to the stream processor over two window intervals, significantly reducing the workload on the stream processor while costing only one additional window of delay.

**ILP for dynamic refinement.** The ILP for jointly computing partitioning and refinement plans is an extension to the ILP from Section 2.4.3. Table 2.4 presents the full version of the extended ILP, including these new constraints. The objective is the same, but the query planner must also compute the cost of executing combinations of refined queries to estimate the total cost of candidate query plans. We add new decision variables $I_{q,r}$ and $F_{q,r_1,r_2}$ to model the workload on the stream processor in the presence of refined queries. $I_{q,r}$ is set to one if the refinement plan for query $q$ includes level $r$. $F_{q,r_1,r_2}$ is set to one if level $r_2$ is executed after $r_1$ for query $q$. These two variables are related by $\sum_{r_1} F_{q,r_1,r_2} = I_{q,r_2}$. We also augment $X$ and $S$ variables with subscripts to account for refinement levels.

**Additional constraints.** For queries containing `join` operators, the query planner can select refinement keys for each sub-query separately, but it must ensure that both sub-queries use the same refinement plan. We then add the constraint $\forall q, r$ and $\forall q_i, q_j \in q : I_{q_i,r} = I_{q_j,r}$. The variables $q_i$ and $q_j$ represent sub-queries of query $q$ containing a `join` operation. The query planner also limits the maximum detection delay for each query, $\forall q : \sum_r I_{q,r} \leq D_q$. Here, $D_q$ is the maximum delay query $q$ can tolerate expressed in number of time windows.

## 2.6 Implementation

Figure 2.6 illustrates the Sonata implementation. For each query, the *core* generates partitioned and refined queries; *drivers* compile the parts of each query to the

| | |
|---|---|
| **Goal** | $min(N = \sum_{q} \sum_{r_2} L_{q,t,r_2} \cdot N_{q,t,r_2})$ <br> $N_{q,t,r_2} = I_{q,r_2} \cdot \sum_{r_1} F_{q,r_1,r_2} \cdot N_{q,t,r_1,r_2}$ |
| **Constraints** | |
| $C1:$ | $\forall s : \sum_{q} X_{q,s,t} \cdot B_{q,t} \leq B_{max}$ <br> $B_{q,t} = \sum_{r_2} I_{q,r_2} \sum_{r_1} F_{q,r_1,r_2} \cdot B_{q,r_2,t}$ |
| $C2:$ | $\forall s : \sum_{q} \sum_{t} X_{q,t,s} \leq W_{max}$ <br> $X_{q,t,s} = \sum_{r} I_{q,r} \cdot X_{q,t,s,r}$ |
| $C3:$ | $\forall q, t, r : S_{q,t,r} \leq S_{max} - 1$ |
| $C4:$ | $\forall q, r, i < j : S_{q,j,r} < S_{q,i,r}$ |
| $C5:$ | $\forall q : \sum_{q} \sum_{r} I_{q,r} \cdot M_{q,r} \leq M$ |
| $C6:$ | $\forall q_i, q_j, r : I_{q_i,r} = I_{q_j,r}$ |
| $C7:$ | $\forall q : \sum_{r} I_{q,r} \geq D_q$ |

Table 2.4: Extension of ILP to support dynamic refinement.

appropriate component. When packets arrive at a PISA switch, it applies the packet-processing pipelines and mirrors the appropriate packets to a monitoring port, where a software *emitter* parses the packets and sends the corresponding tuples to the stream processor. The stream processor reports the results of the queries to the runtime, which then updates the switch, via the data-plane driver, to perform dynamic refinement.

**Core.** The core has two modules: (1) the query planner and (2) the runtime. Upon initialization or re-training, the runtime polls the data-plane driver over a network socket to determine which dataflow operators the switch is capable of executing, as well as the values of the data-plane constraints (*i.e.*, $M$, $A$, $B$, $S$). It then passes these values to the query planner which uses Gurobi [43] to solve the query planning ILP offline and to generate partitioned, refined queries. The runtime then sends parti-

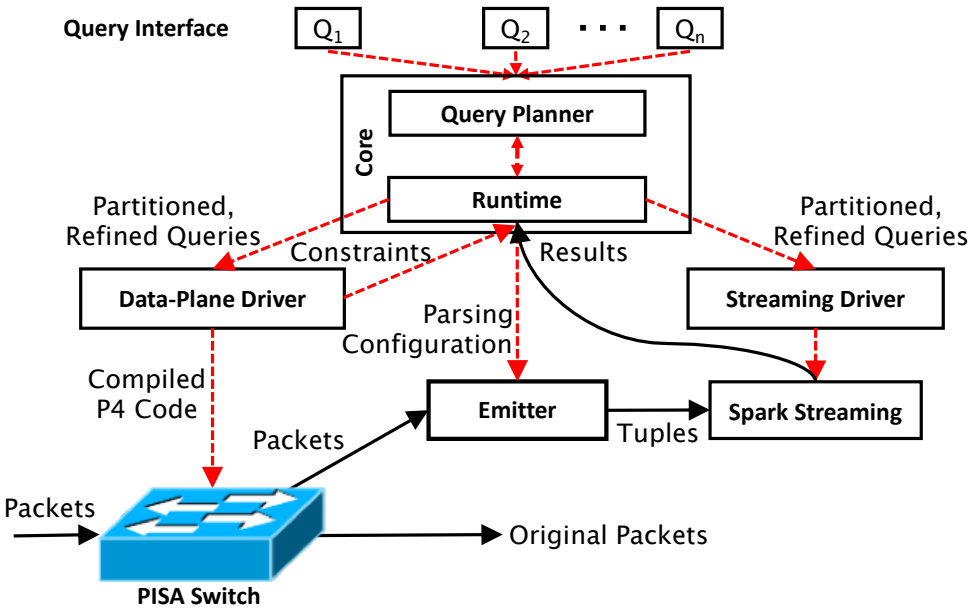



Figure 2.6: Sonata Implementation. Red arrows show compilation control flow and black ones show packet/tuple data flow

tioned and refined queries to the data-plane and streaming drivers. It also configures the emitter— specifying the fields to extract from each packet for each query; each query is identified by a corresponding query identifier (`qid`). When the switch begins processing packets, the runtime receives query outputs from the stream processor at the end of every window. It then sends updates to the data-plane driver, which in turn updates table entries in the switch according to the dynamic refinement plan. When it detects too many hash collisions, the runtime triggers the query planner to re-run the ILP with the new data.

**Drivers.** Data-plane and streaming drivers compile the queries from the runtime to target-specific code that can run on the switch and stream processor respectively. The data-plane drivers also interact with the switch to execute commands on behalf of the runtime, such as updating `filter` tables for iterative refinement at the end of every window. The Sonata implementation currently has drivers for two PISA switches: the BMV2 P4 software switch [114], which is the standard behavioral model for evaluating P4 code; and the Barefoot Wedge 100B-65X (Tofino) [112] which is a

| # | Query | Lines of Code | | |
|---|---|---|---|---|
| | | *Sonata* | *P4* | *Spark* |
| 1 | Newly opened TCP Conns. [129] | 6 | 367 | 4 |
| 2 | SSH Brute Force [53] | 7 | 561 | 14 |
| 3 | Superspreader [126] | 6 | 473 | 10 |
| 4 | Port Scan [56] | 6 | 714 | 8 |
| 5 | DDoS [126] | 9 | 691 | 8 |
| 6 | TCP SYN Flood [129] | 17 | 870 | 10 |
| 7 | TCP Incomplete Flows [129] | 12 | 633 | 4 |
| 8 | Slowloris Attacks [129] | 13 | 1,168 | 15 |
| 9 | DNS Tunneling [13] | 11 | 570 | 12 |
| 10 | Zorro Attack [80] | 13 | 561 | 14 |
| 11 | DNS Reflection Attack [62] | 14 | 773 | 12 |

Table 2.5: Implemented Sonata Queries. We report lines of code considering the same: (1) refinement plan; (2) partitioning plan, *i.e.*, executing as many dataflow operators in the switch as possible.

6.5 Tbps hardware switch. The data-plane driver communicates with these switches using a Thrift API [6].

**Emitter.** The emitter consumes raw packets from the data- plane's monitoring port, parses the query-specific fields in the packet, and sends the corresponding tuples to the stream processor. The emitter uses Scapy [115] to extract the unique (`qid`) from packets. It uses this identifier to determine how to parse the remainder of the query-specific fields embedded in the packet based on the configuration provided by the runtime. As discussed in Section 2.4.1, the emitter immediately sends the output of stateless operators to the stream processor, but it stores the output of stateful operators in a local key-value data store. At the end of each window interval, it reads the aggregated value for each key in the local data store from the data-plane registers before sending the output tuples to the stream processor.

## 2.7 Evaluation

In this section, we (1) demonstrate that Sonata is flexible (Table 2.5). We use real-world packet traces to (2) show that Sonata reduces the workload on the stream processor by 3–7 orders of magnitude (Figure 2.7) and that these results are robust to various switch resource constraints (Figure 2.8). We also (3) present a case study with a Tofino switch to demonstrate how Sonata operates end-to-end, discovering "needles" of interest without collecting the entire "haystack" (Figure 2.9).

### 2.7.1 Setup

**Monitoring applications.** To demonstrate the flexibility of Sonata's query interface, we implemented eleven different monitoring tasks, as shown in Table 2.5. We show how Sonata makes it easier to express queries for complex monitoring tasks by comparing the lines of code needed to express those tasks. For each query, Sonata required far fewer lines of code to express the same task than the code for the switch [14] and streaming [111] targets combined. Not only does Sonata reduce the lines of code, but also the queries expressed with Sonata are platform-agnostic and could execute unmodified with a different choice of hardware switch or stream processor, e.g., Apache Flink.

**Packet traces.** We use CAIDA's anonymized and unsampled packet traces [92], which were captured from a large ISP's backbone link between Seattle and Chicago. We evaluate over a subset of this data containing 600 million packets and transferring about 360 GB of data over 10 minutes. This data contains no layer-2 headers or packet payloads, and the layer-3 headers were anonymized with a prefix-preserving algorithm [32].

We replay the traffic at $20x$ speed to evaluate Sonata on a simulated 100 Gbps workload (*i.e.*, about 20 million packets per second) that might be experienced at

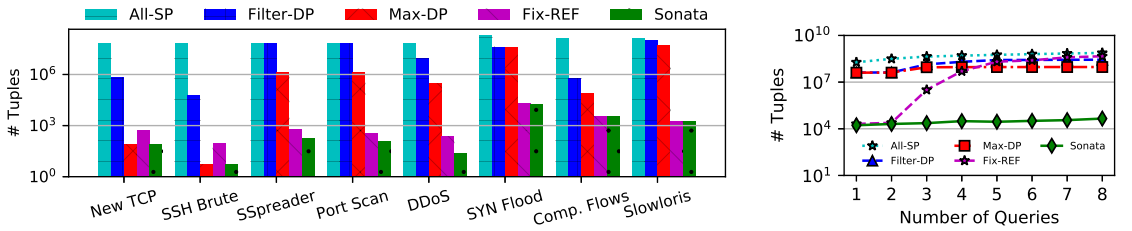
| Query Plan | Description | Monitoring Systems |
|---|---|---|
| **All-SP** | Mirror all incoming packets to the stream processor | Gigascope[23], OpenSOC[90], NetQRE[129] |
| **Filter-DP** | Apply only filter operations on the switch | EverFlow[132] |
| **Max-DP** | Execute as many dataflow operations as possible on the switch | Univmon[65], OpenSketch[126] |
| **Fix-REF** | Iteratively zoom-in one refinenement level at a time | DREAM[71] |

Table 2.6: Monitoring systems emulated for evaluation.

a border switch in a large network. We use a time window ($W$) of three seconds, resulting in 60 million packets per window. In the worst case, Sonata can only detect network events lasting at least $W \times |R|$ seconds. However, the network operator can force Sonata to consider fewer refinement levels with the maximum delay ($D_{max}$) constraint, which reduces the time required to detect network events.

**Targets.** Since switches have fixed resource constraints, we choose to evaluate Sonata's performance with simulated PISA switches. This approach allows us to parameterize the various resource constraints and to evaluate Sonata's performance over a variety of potential PISA switches. Unless otherwise specified, we present results for a simulated PISA switch with sixteen stages ($S = 16$), eight stateful operators per stage ($A = 8$), and eight Mb of register memory per stage ($B = 8$ Mb). Within each stage, a single stateful operator can use up to four Mb.

**Comparisons to existing systems.** Since the key performance metric for Sonata is the number of packet tuples processed by the stream processor, we do not need to instrument the several existing monitoring systems, such as Gigascope [23], OpenSOC [91], EverFlow [132], OpenSketch [126], and DREAM [71], for comparison. Instead, we compare Sonata's performance with four alternative query plans which are each representative of groups of existing systems as shown in Table 2.6. We modify the constraints on Sonata's query planning ILP to generate plans that




(a) Single-query performance (b) Multi-query performance

Figure 2.7: Reduction in workload on the stream processor running: (a) one query at a time, (2) concurrently running multiple queries.

emulate the performance for each of these solutions. For example with the *Fix-REF* plan, we add the constraint $\forall q, r : I_{q,r} = 1$.

### 2.7.2 Load on the Stream Processor

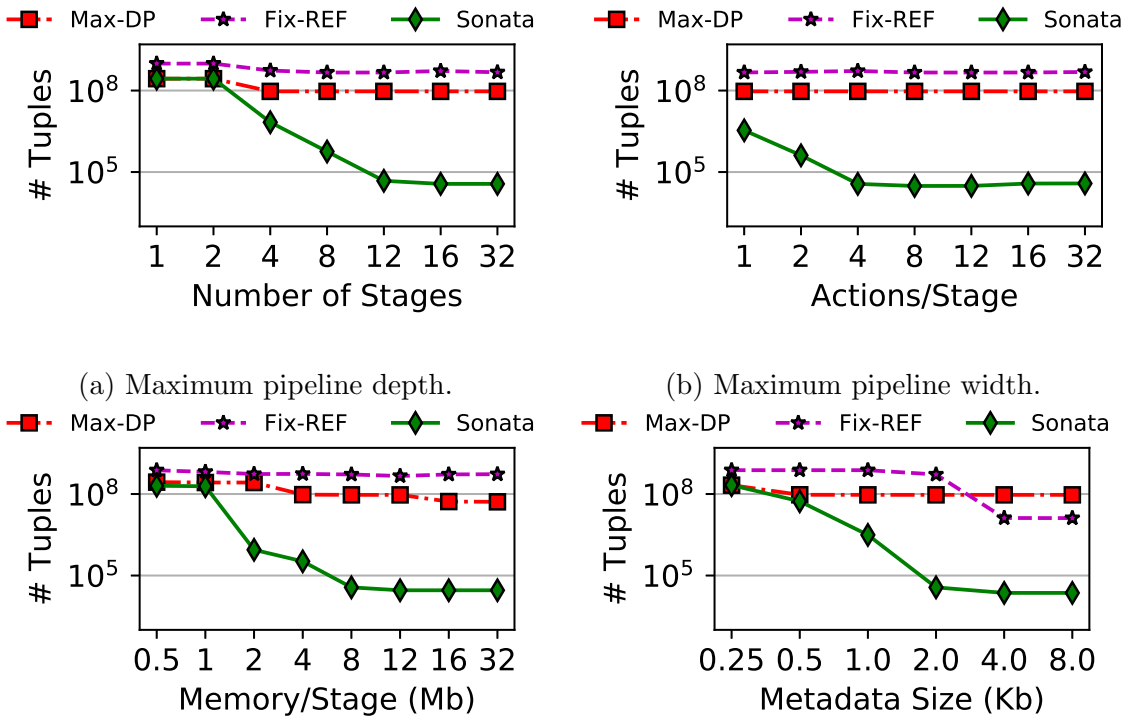
We perform trace-driven analysis to quantify how much Sonata reduces the workload on the stream processor. To enable comparison with prior work, we evaluate the top eight queries from Table 2.5; these queries process only layer 3 and 4 header fields. We consider a maximum of eight refinement levels for all queries (*i.e.*, $R = \{4, 8, ..., 32\}$); additional levels offered only marginal improvements. *Fix-REF* queries use all eight refinement levels, while Sonata may select a subset of all eight levels in its query plans.

**Single query performance.** Figure 2.7a shows that Sonata reduces the workload on the stream processor by as much as seven orders of magnitude. *Filter-DP* is efficient for the SSH brute-force attack query, because this query examines such a small fraction of the traffic. *Filter-DP*'s performance is similar to *All-SP* for queries that must process a larger fraction of traffic, such as detecting Superspreaders [126]. For some queries, such as the SSH brute-force attack, *Max-DP* matches Sonata's performance. In many other cases, large amounts of traffic are sent to the stream processor due to a lack of resources. For example, the Superspreader query exhausts stateful

processing resources. *Fix-REF*'s performance matches Sonata's for most cases, but uses up to seven additional windows to detect traffic that satisfies the query.

**Multi-query performance.** Figure 2.7b shows how the workload on the stream processor increases with the number of queries. When executing eight queries concurrently, Sonata reduces the workload by three orders of magnitude compared to other query plans. These gains come at the cost of up to three additional time windows to detect traffic that satisfies the query. The performance of *Fix-REF* degrades the most because the available switch resources, such as metadata and stages, are quickly exhausted when supporting a fixed refinement plan for several queries. We have also considered query plans with fewer refinement levels for *Fix-REF* and observed similar trends. For example, when considering just two refinement levels (`dIP/16` and `dIP/32`) for all eight queries, we observed that the load on the stream processor was two orders of magnitude greater than Sonata.

**Effect of switch constraints.** We study how switch constraints affect Sonata's ability to reduce the load on the stream processor. To quantify this relationship, we vary one switch constraint at a time for the simulated PISA switch. Figure 2.8a shows how the workload on the stream processor decreases as the number of stages increases. More stages allow Sonata to consider more levels for dynamic refinement. Additional stages slightly improve the performance of *Fix-REF* as it can now support stateful operations for the queries at finer refinement levels on the switch. We observe similar trends as the number of stateful actions per stage (Figure 2.8b), memory per stage (Figure 2.8c), and total metadata size (Figure 2.8d) increase. As expected, *Max-DP* slightly reduces the load on the stream processor when more memory per stage is available for stateful operations; increasing the total metadata size also allows *Fix-REF* to execute more queries in the switch—reducing the load on the stream processor.




(a) Maximum pipeline depth.

(b) Maximum pipeline width.

(c) Maximum bits per stage.

(d) Metadata Size.

Figure 2.8: Effect of switch constraints.

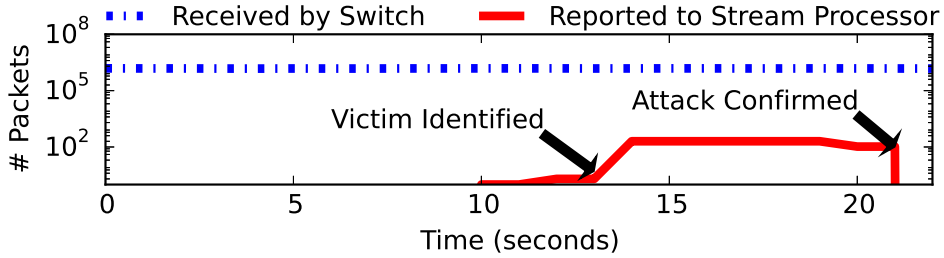



Figure 2.9: Detecting Zorro attacks using Tofino switch.

**Overhead of dynamic refinement.** When running all eight queries concurrently, as many as 200 `filter` table entries are updated after each time window during dynamic refinement. Micro-benchmarking experiments with the Tofino switch [112] show that updating 200 table entries takes about 127 ms, and resetting registers takes about 4 ms. The total update time took 131 ms which is about 5% of the specified window interval ($W = 3$s).

### 2.7.3 Case Study: Tofino Switch

We used Sonata to execute Query 2.3 with a Tofino switch [112]. We chose this query to highlight how Sonata handles `join` operators and operations over a packet's payload. For this experiment, we built a testbed containing four hosts and a Tofino switch [112]. Each host has two Intel Xeon E5-2630 v4 10-core processors running at 2.2 Ghz with 128 GB RAM and 10 Gbps NICs. We dedicate two hosts for traffic generation: one sender and one receiver. We assign a third host for the emitter component and a fourth for the remaining runtime, streaming driver, and Spark Streaming [111] components (see Figure 2.6). The data-plane driver runs on the CPU of the Tofino switch itself. The sender connects to the Tofino switch with two interfaces: one interface to replay CAIDA traces using the Moongen [27] traffic generator at about 1.5 Mpps and another to send attack traffic using Scapy [115]. If we were processing packets at Tofino's maximum rate of 6.5 Tbps, our setup would only need to replace the single instance of Spark Streaming with a cluster that supports the expected data rate.

The attacker starts sending similar-sized telnet packets to a single host (`99.7.0.25`) at time $t = 10$ s. Figure 2.9 shows the number of packets: (1) received by the switch, and (2) reported to the stream processor on a log scale. Sonata reports only two packet tuples, out of 1.5M pps, to the stream processor to detect the victim in three seconds using two refinement levels: $* \rightarrow 24$ and $24 \rightarrow 32$. At $t = 13$ s, the stream processor starts processing the payload of all telnet packets destined for the victim host, which is only around 100 pps. The attacker gains shell access at $t = 20$ s and sends five packets with the keyword "zorro" in it. Sonata detects the attack at $t = 21$ s, demonstrating its ability to perform real-time monitoring using state-of-the-art hardware switches.

## 2.8 Related Work

**Network monitoring.** Existing monitoring systems that process all packets at the stream processor such as Chimera [13], Gigascope [23], OpenSOC [90], and NetQRE [129] can express a range of queries but can only support lower packet rates because the stream processor ultimately processes all results. These systems also require deploying and configuring a collection infrastructure to capture packets from the data plane for analysis, incurring significant bandwidth overhead. These systems can benefit from horizontally scalable stream processors such as Spark Streaming [130] and Flink [86], but they also face scaling limitations due to packet parsing and cluster coordination [91].

Everflow [132], UnivMon [65], OpenSketch [126], and Marple [76] rely on programmable switches to execute queries entirely in the data plane. These systems can process queries at line rate but can only support queries that can be implemented on switches. Trumpet [73] and Pathdump [108] offload query processing to end-hosts (VMs in data center networks) but not to switches. Gupta et al. [38] proposed a monitoring system that could coordinate queries across a stream processor and switch, but the work considered only switches with fixed-function chipsets for single queries, and required network operators to explicitly specify the refinement and partitioning plans. In contrast, Sonata supports programmable switches and employs a sophisticated query planner to automatically partition and refine multiple queries. We also quantify the performance gains and overhead with realistic packet traces and a programmable hardware switch.

**Query planning.** Database research has explored query planning and optimization extensively [81, 74, 8]. Gigascope performs query partitioning to minimize the data transfer from the capture card to the stream processor [23]. Sensor networks have explored the query partitioning problems that are similar to those that Sonata faces [81, 74, 8, 66, 67, 105]. However, these systems face different optimization prob-

lems because they typically involve lower traffic rates and involve special-purpose queries. Path Queries [75] and SNAP [7] facilitate network-wide queries that execute across multiple switches; in contrast, Sonata currently only compiles queries to a single switch, but it addresses a complementary set of problems, such as unifying data-plane and stream processing platforms to support richer queries and partitioning sets of queries across a data-plane switch and a stream processor.

**Query-driven dynamic refinement.** Autofocus [30], ProgME [128], and DREAM [71], SCREAM [72], MULTOPS [36], and HHH [55] all iteratively zoom in on traffic of interest. These systems either do not apply to streaming data (e.g., ProgME requires multiple passes over the data [128]) they use a static refinement plan for all queries (e.g., HHH zooms in one bit at a time), or they do not satisfy general queries on network traffic (e.g., MULTOPS is specifically designed for bandwidth attack detection). These approaches all rely on general-purpose CPUs to process the data-plane output, but none of them permit additional parsing, joining, or aggregation at the stream processor, as Sonata does.

# Chapter 3

# Network Control with SDX

In this chapter, we present the design, implementation, and evaluation of network-control system, SDX, that supports flexible control programs at Internet exchange points (IXPs). It utilizes both the programmable switch and participants' fixed-function border routers at the IXP to scale these control programs with the number of IXP participants.

## 3.1 Overview

Internet routing is unreliable, inflexible, and difficult to manage. Network operators must rely on arcane mechanisms to perform traffic engineering, prevent attacks, and realize peering agreements. Most of the Internet's routing problems result from three characteristics of the Border Gateway Protocol (BGP), the Internet's interdomain routing protocol:

- **Routing only on destination IP prefix.** BGP selects and exports routes for destination prefixes. Networks cannot make more fine-grained decisions based on the type of application or the sender.

- **Influence only over direct neighbors.** A network selects among BGP routes learned from its direct neighbors and exports selected routes to these neighbors. Networks have little control over end-to-end paths.
- **Indirect expression of policy.** Networks rely on indirect, obscure mechanisms (e.g., "local preference", "AS Path Prepending") to influence path selection. Networks cannot directly express preferred inbound and outbound paths.

These problems are well-known, yet incremental deployment of alternative solutions is a perennial problem in a global Internet with more than $50,000$ independently operated networks and a huge installed base of BGP-speaking routers.

In this chapter, we present a way forward that improves our existing routing system by allowing a network to execute a far wider range of decisions concerning end-to-end traffic delivery. Our approach builds on recent technology trends and also recognizes the need for incremental deployment. First, we argue that Software Defined Networking (SDN) shows great promise for simplifying network management and enabling new networked services. SDN switches match on a variety of header fields (not just destination prefix), perform a range of actions (not just forwarding), and offer direct control over the data plane. Yet, SDN currently only applies to *intra*-domain settings, such as individual data-center, enterprise, or backbone networks. By design, a conventional SDN controller has purview over the switches within a single administrative (and trust) domain.

Second, we recognize the recent resurgence of interest in Internet exchange points (IXPs), which are physical locations where multiple networks meet to exchange traffic and BGP routes. An IXP is a layer-two network that, in the simplest case, consists of a single switch. Each participating network exchanges BGP routes (often with a BGP route server) and directs traffic to other participants over the layer-two fabric. The Internet has more than 300 IXPs worldwide—with more than 80 in North America alone—and some IXPs carry as much traffic as some of the tier-1 ISPs. For example,

the Open-IX effort seeks to develop new North American IXPs with open peering and governance, similar to the models already taking root in Europe. As video traffic continues to increase, tensions grow between content providers and access networks, and IXPs are on the front line of today's peering disputes. In short, not only are IXPs the right place to begin a revolution in wide-area traffic delivery, but the organizations running these IXPs have strong incentives to innovate.

We aim to change wide-area traffic delivery by designing, prototyping, and deploying a software defined exchange (SDX). Contrary to how it may seem, however, merely operating SDN switches and a controller at an IXP does not automatically present a turnkey solution. SDN is merely a tool for solving problems, not the solution. In fact, running an SDN-enabled exchange point introduces many problems, ranging from correctness to scalability. To realize the SDX in practice, we must address the following four challenges:

- **Compelling applications.** The success of the SDX depends on identifying compelling wide-area traffic-delivery applications that are difficult to deploy today. We present five motivating applications: application-specific peering, inbound traffic engineering, server load balancing, and traffic redirection through middleboxes (Section 3.2).
- **Programming abstractions.** Participating networks need a way to create and run applications, without conflicting with each other or with the global routing system. Our SDX design presents each participating network with the illusion of its own virtual SDN switch that extends the footprint of its legacy network and enables flexible policies that interact safely with today's BGP (Section 3.3).
- **Scalable operation.** An SDX needs to support hundreds of participants, hundreds of thousands of IP prefixes, and policies that match on multiple packet-header fields—all while using conventional SDN switches. We show how to combine the policies of multiple participants and join them with the current

BGP routes, while limiting rule-table size and computational overhead (Section 3.4).

- **Realistic deployment.** We have built a prototype and created two example applications (Section 3.5). Experiments demonstrate that our prototype scales (in terms of rule-table size and CPU time) to many participants, policies, and prefixes (Section 3.6).

We survey the related work in Section 3.7.

## 3.2 Background and Motivation

In this section, we first present the conventional IXP architecture, followed by how SDX enables various new applications for flexible wide-area traffic delivery.

### 3.2.1 Conventional IXP Architecture

An IXP is a physical location where multiple networks meet to exchange traffic. An IXP is a layer-two network that, in the simplest case, consists of a single switch. Each member network physically connects one or more edge routers to the IXP, where each router port has a unique MAC address as well as a dedicated IP address from the IXP's own address block. Since all router ports belong to the same IP subnet, one member can direct traffic to another simply by sending a packet with the appropriate destination MAC address.

Two members can peer by establishing a Border Gateway Protocol (BGP) session between their respective edge routers, and applying local policies for selecting and exporting routes. Each BGP route has various attributes, including the IP prefix, the Autonomous System (AS) path, and the next-hop IP address of the neighboring router. Upon choosing a route, the member's router creates a forwarding-table entry that maps the destination IP prefix to (1) its output port connected to the IXP and

(2) the destination MAC address of the chosen next-hop (resolved from the next-hop IP address using ARP). As such, the IXP switch does *not* need any information about IP prefixes, and simply forwards the packet based on the destination MAC address.

Rather than having a BGP session between each pair of members, IXPs often host a Route Server (RS) that acts as a sort of BGP multiplexer [52, 37]. Each member establishes two BGP sessions to the RS, and the RS applies all of the selection and export policies on each member's behalf, based on policies provided by the members. The RS sends each member one "best" BGP route (if any) for each destination IP prefix, subject to the export policy of the member that announced the route. The next-hop attribute corresponds to the IP address of this member's router port, rather than the RS itself, so the members can exchange data traffic directly. That is, the RS is purely a control-plane entity, with no participation in packet forwarding. In practice, IXP members use the RS for most BGP peering relationships, but may have dedicated BGP sessions for their most important peers [3].

### 3.2.2 Wide-Area Traffic Delivery

We present five different applications that the SDX enables. We describe how operators tackle these problems today, focusing in particular on the "pain points" for implementing these functions in today's infrastructure. We also describe how these applications would be easier to implement with the SDX. We revisit several of these examples throughout the paper, both demonstrating how the SDX makes them possible and, in some cases, deploying them in the wide area.

**Application-specific peering.** High-bandwidth video services like YouTube and Netflix constitute a significant fraction of overall traffic volume, so ISPs are increasingly interested in application-specific peering, where two neighboring AS exchange traffic only for certain applications. BGP does not make it easy to support such an

arrangement. An ISP could configure its edge routers to make different forwarding decisions for different application packet classifiers (to identify the relevant traffic) and policy-based routing (to direct that traffic over a special path). For example, an ISP could configure its border routers to have multiple VRFs (virtual routing and forwarding tables), one for each traffic class, and direct video traffic via one VRF and non-video traffic through another. Still, such an approach forces the ISPs to incur additional routing and forwarding state, in proportion to the number of traffic classes, and configure these mechanisms correctly. SDX could instead install custom rules for groups of flows corresponding to specific parts of flow space.

**Inbound traffic engineering.** Because BGP performs *destination*-based routing, ASes have little control over how traffic *enters* their networks and must use indirect, obscure techniques (e.g., AS path prepending, communities, selective advertisements) to influence how ASes reach them. Each of these existing approaches is limited: prepending cannot override another AS's local preference for outbound traffic control, communities typically only affect decisions of an immediate neighbor network, and selective advertisements pollute the global routing tables with extra prefixes. By installing forwarding rules in SDN-enabled switches at an exchange point, an AS can directly control inbound traffic according to source IP addresses or port numbers.

**Wide-area server load balancing.** Content providers balance client requests across clusters of servers by manipulating the domain name system (DNS). Each service has a single domain name (e.g., `http://www.example.com/`) which resolves to multiple IP addresses for different backend servers. When a client's local DNS server issues a DNS request, the service's authoritative DNS server returns an IP address that appropriately balances load. Unfortunately, using DNS for server selection has several limitations. First, DNS caching (by the local DNS server, and by the user's browser) results in slower responses to failures and shifts in load. To (partially)

address this problem, content providers use low "time to live" values, leading to more frequent DNS cache misses, adding critical milliseconds to request latency. Instead, a content provider could assign a single anycast IP address for a service and rewrite the destination addresses of client requests in the middle of the network (e.g., at exchange points). SDX could announce anycast prefixes and rewrite the destination IP address to match the chosen hosting location based on any fields in the packet header.

**Blocking of denial-of-service attacks.** Denial-of-service attacks are a persistent security problem. Enterprises and content providers devote substantial resources to detecting and blocking attacks. However, most techniques block attacks close to the victim, by measuring and classifying the attacks, and dynamically installing packet filters to block the offending traffic. Blocking attacks further upstream, closer to the attacker, avoid wasting network resources near the victim. However, existing techniques for blocking traffic upstream are quite clumsy. For example, a network can use BGP "route poisoning" to block an entire sending AS (and any of its single-homed customers) from directing traffic to a destination prefix. This is done by placing the offending network's AS number in the BGP AS-PATH, with the goal of triggering BGP loop detection that causes this AS to filter the route. However, the technique relies on "attacking" BGP by sending erroneous routing information, and only works at the level of destination IP prefixes—leading to substantial "collateral damage" by blocking substantial benign traffic.

Rather than manipulate BGP, an SDN-enabled exchange point can allow a victim to remotely specify a fine-grained access-control rule (e.g., matching on source and destination IP addresses, protocol, and TCP/UDP port numbers) to drop unwanted traffic close to the sender.

**Redirection through middleboxes.** Networks increasingly rely on middleboxes to perform a wide range of functions (e.g., firewalls, network address translators, load

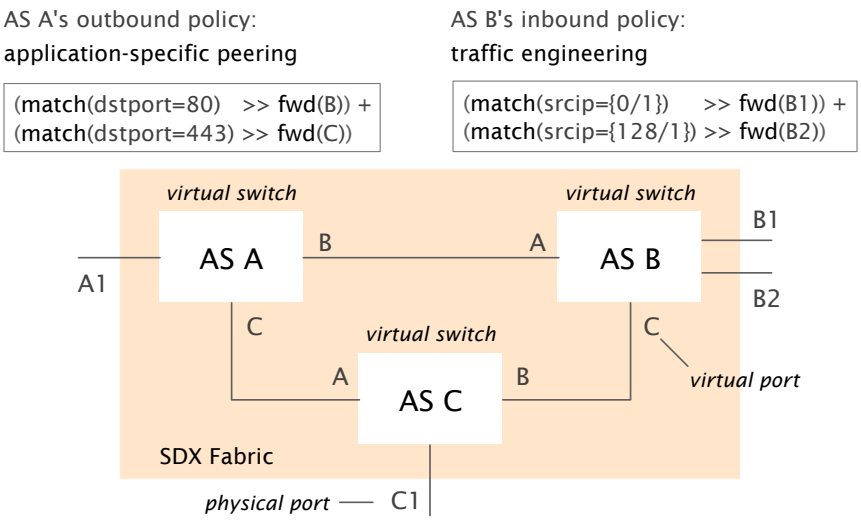
balancers). Enterprise networks at the edge of the Internet typically place middleboxes at key junctions, such as the boundary between the enterprise and its upstream ISPs, but large ISPs are often geographically expansive, making it prohibitively expensive to place middleboxes at every location. Instead, they manipulate the routing protocols to "steer" traffic through a fixed set of middleboxes. For example, when traffic measurements suggest a possible denial-of-service attack, an ISP can use internal BGP to "hijack" the offending traffic and forward it through a traffic scrubber. Some broadband access ISPs perform similar steering of a home user's traffic by routing all home network traffic through a scrubber via a VPN. Such steering requires ISPs to "hijack" much more normal traffic than necessary, and the mechanisms are not well-suited to steering traffic through a *sequence* of middleboxes. Instead, an SDN-enabled exchange point can redirect targeted subsets of traffic through one or more middleboxes.

## 3.3 Abstraction: Virtual SDX Switch

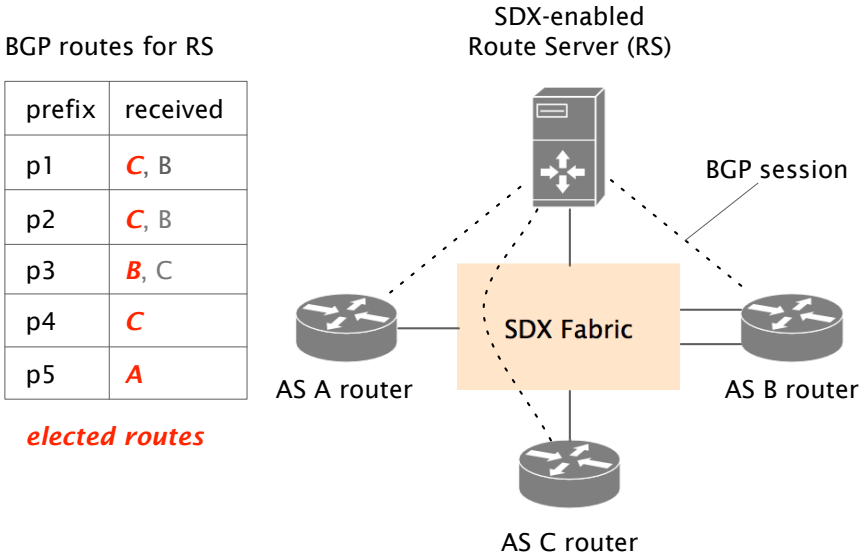
The SDX enables the operators of participating ASes to run novel applications that control the flow of traffic entering and leaving their border routers, or, in the case of remote participants, the flow of traffic destined for their AS. By giving each AS the illusion of its own virtual SDN switch, the SDX enables flexible specification of forwarding policies while ensuring isolation between different participants. SDX applications can base decisions on the currently available BGP routes, which offers greater flexibility while ensuring that traffic follows valid interdomain paths.

### 3.3.1 Virtual SDX Switch Abstraction

In a traditional exchange point, each participating AS typically connects a BGP-speaking border router to a *shared layer-two network* (a data plane for forwarding




(a) Virtual switch abstraction.




(b) Integration with interdomain routes.

Figure 3.1: SDX programming abstractions.

packets) and a *BGP route server* (a control plane for exchanging routing information). At an SDX, each AS can run SDN applications that specify flexible policies for dropping, modifying, and forwarding the traffic. The SDX must then combine the policies of multiple ASes into a single coherent policy for the physical switch(es). To balance the desire for flexibility with the need for isolation, we give each AS the illusion of its own virtual SDN switch connecting its border router to each of its peer

ASes, as shown in Figure 3.1a. AS A has a virtual switch connecting to the virtual switches of ASes B and C, where each AS can write forwarding policies as if it is the only participant at the SDX. Yet, AS A *cannot* influence how ASes B and C forward packets on their own virtual switches.

For writing policies, we adopt the Pyretic language [70] that supports declarative programming based on boolean predicates (that each match a subset of the packets) and a small set of actions (that modify a packet's header fields or location). A Pyretic policy maps a located packet (*i.e.*, a packet and its location) to a set of located packets. Returning the empty set drops the packet. Returning a set with a single packet forwards the packet to its new location. Finally, returning a set with multiple packets multicasts the packets. In contrast to vanilla Pyretic policies, we require participants to specify whether a policy is an *inbound* or an *outbound* policy. Inbound policies apply to the traffic entering a virtual switch on a virtual port from another SDX participant; outbound policies apply to the traffic entering a virtual switch on a physical port from the participant's own border router. In the rest of the paper, we omit this distinction whenever it is clear from context. We now present several simple examples inspired by Section 3.2.2.

**Application-specific peering.** In Figure 3.1a, AS A has an outbound policy that forwards HTTP traffic (destination port 80) and HTTPS traffic (destination port 443) to AS B and AS C, respectively:

```
(match(dstport = 80) >> fwd(B)) +
(match(dstport = 443) >> fwd(C))
```

The `match()` statement is a filter that returns all packets with a transport port number of 80 or 443, and the `>>` is the sequential composition operator that sends the resulting packets to the `fwd(B)` (or, respectively, `fwd(C)`) policy, which in turn modifies the packets' location to the corresponding virtual switch. The `+` operator corresponds to

parallel composition which, given two policies, applies them both to each packet and combines their outputs. If neither of the two policies matches, the packet is dropped.

**Inbound traffic engineering.** AS B has an inbound policy that performs inbound traffic engineering over packets coming from ASes A and C:

```
(match(srcip = {0.0.0.0/1}) >> fwd(B1)) +
(match(srcip = {128.0.0.0/1}) >> fwd(B2))
```

AS B directs traffic with source IP addresses starting with 0 to B's top output port, and the remaining traffic (with source IP addresses starting with 1) to B's bottom output port. Under the hood, the SDX runtime system "compiles" A's outbound policy with B's inbound policy to construct a single policy for the underlying physical switch, such as:

```
(match(port=A1, dstport=80,
       srcip={0.0.0.0/1}) >> fwd(B1)) +
(match(port=A1, dstport=80,
       srcip={128.0.0.0/1}) >> fwd(B2))
```

that achieves the same outcome as directing traffic through multiple virtual switches (here, A and B's switches). This policy has a straightforward mapping to low-level rules on OpenFlow switches [70].

**Wide-area server load balancing.** An AS can have a virtual switch at the SDX *without* having any physical presence at the exchange point, in order to influence the end-to-end flow of traffic. For example, a content provider can perform server load balancing by dividing request traffic based on client IP prefixes and ensuring connection affinity across changes in the load-balancing policy [122]. The content provider might host a service at IP address `74.125.1.1` and direct specific customer prefixes to specific replicas based on their request load and geographic location:

```
match(dstip=74.125.1.1) >>
    (match(srcip=96.25.160.0/24) >>
        mod(dstip=74.125.224.161)) +
    (match(srcip=128.125.163.0/24) >>
        mod(dstip=74.125.137.139))
```

Manipulating packet forwarding at the SDX gives a content provider fast and direct control over the traffic, in contrast to existing indirect mechanisms like DNS-based load balancing. The content provider issuing this policy would first need to demonstrate to the SDX that it owns the corresponding IP address blocks.

### 3.3.2 Integration with Interdomain Routing

The ASes must define SDX policies in relation to the advertised routes in the global routing system. To do so, the SDX allows participating ASes to define forwarding policies relative to the current BGP routes. To learn BGP routes, the SDX controller integrates a route server, as shown in Figure 3.1b. Participants interact with the SDX route server in the same way that they do with a conventional route server. The SDX route server collects the routes advertised by each participant BGP router and selects one best route for each prefix on behalf of each participant, and re-advertises the best BGP route on the appropriate BGP session(s). In contrast to today's route servers, where each participant learns and uses one route per prefix, the SDX route server allows each participant to forward traffic to all feasible routes for a prefix, even if it learns only one.

**Overriding default BGP routes.** Many ASes may be happy with how BGP computes routes for most of the traffic. Rather than requiring each AS to fully specify the forwarding policy for all traffic, the SDX allows each AS to rely on a default forwarding policy computed by BGP, overriding the policy as needed. In the

example in Figure 3.1a, AS A's outbound policy for Web traffic (forwarding to AS B) applies only to Web traffic; all of the remaining traffic implicitly follows whatever best route AS A selects in BGP. This greatly simplifies the task of writing an SDX application: the simplest application specifies nothing, resulting in all traffic following the BGP-selected routes announced by the route server. The programmer need only specify the handling of any "non-default" traffic. For example in Figure 3.1b, AS A would forward any non-Web traffic destined to IP prefix $p_1$ or $p_2$ to next-hop AS C, rather than to AS B.

**Forwarding only along BGP-advertised paths.** The SDX should not direct traffic to a next-hop AS that does not want to receive it. In Figure 3.1b, AS B does *not* export a BGP route for destination prefix $p_4$ to AS A, so AS A should not forward any traffic (including Web traffic) for $p_4$ through AS B. To prevent ASes from violating these restrictions, and to simplify the writing of applications, the SDX only applies a `match()` predicate to the portion of traffic that is eligible for forwarding to the specified next-hop AS. In Figure 3.1, AS A can forward Web traffic for destination prefixes $p_1$, $p_2$, and $p_3$ to AS B, but not for $p_4$. Note that, despite not selecting AS B as the best route for destination prefix $p_1$ and $p_2$, AS A can still direct the corresponding Web traffic through AS B, since AS B does export a BGP route for these prefixes to AS A.

**Grouping traffic based on BGP attributes.** ASes may wish to express policies based on higher levels of abstraction than IP prefixes. Instead, an AS could handle traffic based on the organization managing the IP address (e.g., "all flows sent by YouTube") or the current AS-PATH for each destination prefix. The SDX allows a policy to specify a match *indirectly* based on regular expressions on BGP route attributes. For example, an AS could specify that all traffic sent by YouTube servers traverses a video-transcoding middlebox hosted at a particular port (E1) at the SDX:

```
YouTubePrefixes =
          RIB.filter('as_path', .*43515$)
match(srcip={YouTubePrefixes}) >> fwd(E1)
```

The regular expression matches all BGP-announced routes ending in AS 43515 (YouTube's AS number), and generates the list of associated IP prefixes. The `match()` statement matches any traffic sent by one of these IP addresses and forwards it to the output port connected to the middlebox.

**Originating BGP routes from the SDX.** In addition to forwarding traffic along BGP-advertised paths, ASes may want the SDX to originate routes for their IP prefixes. In the wide-area load-balancing application, a remote AS D instructs the SDX to match request traffic destined to an anycast service with IP address 74.125.1.1. To ensure the SDX receives the request traffic, AS D needs to trigger a BGP route announcement for the associated IP prefix (`announce(74.125.1.0/24)`), and withdraw the prefix when it is no longer needed (`withdraw(74.125.1.0/24)`). AS D could announce the anycast prefix at multiple SDXs that each run the load-balancing application, to ensure that all client requests flow through a nearby SDX. Before originating the route announcement in BGP, the SDX would verify that AS D indeed owns the IP prefix (e.g., using the RPKI).

**Integrating SDX with existing infrastructure.** Integrating SDX with existing IXP infrastructure and conventional BGP-speaking ASes is straightforward. Any participant that is physically connected to a SDN-enabled switch exchanges BGP routes with the SDX route server can write SDX policies; furthermore, an AS can benefit from an SDX deployment at a single location, even if the rest of the ASes run only conventional BGP routing. A participant can implement SDX policies for any route that it learns via the SDX route server, independently of whether the AS that

originated the prefix is an SDX participant. Participants who are physically present at the IXP but do not want to implement SDX policies see the same layer-2 abstractions that they would at any other IXP. The SDX controller can run a conventional spanning tree protocol to ensure seamless operation between SDN-enabled participants and conventional participants.

## 3.4 Efficient Compilation

In this section, we describe how the SDX runtime system compiles the policies of all participants into low-level forwarding rules (Section 3.4.1). We then describe how we made that process efficient. We consider *data-plane* efficiency (Section 3.4.2), to minimize the number of rules in the switches, and *control-plane* efficiency (Section 3.4.3), to minimize the computation time under realistic workloads.

### 3.4.1 Compilation by Policy Transformation

The policies written by SDX participants are abstract policies that need to be joined with the BGP routes, combined, and translated to equivalent forwarding rules for the physical switch(es). We compile the policies through a sequence of syntactic transformations: (1) restricting policies according to the virtual topology; (2) augmenting the policies with BGP-learned information; (3) extending policies to default to using the best BGP route; and (4) composing the policies of all the participants into one main SDX policy by emulating multiple hops in the virtual topology. Then, we rely on the underlying Pyretic runtime to translate the SDX policy into forwarding rules for the physical switch.

**Enforcing isolation between participants.** The first transformation restricts the participant's policy so that each participant can only act on its own virtual

switch. Each port on a virtual switch corresponds either to a physical port at the SDX (e.g., $A1$ in Figure 3.1a) or a virtual connection to another participant's virtual switch (e.g., port $B$ on AS A's virtual switch in Figure 3.1a). The SDX runtime must ensure that a participant's outbound policies only apply to the traffic that it sends. Likewise, its inbound policies should only apply to the traffic that it receives. For example, in Figure 3.1a, AS A's outbound policy should only apply to traffic that it originates, not to the traffic that AS B sends to it. To enforce this constraint, the SDX runtime automatically augments each participant policy with an explicit `match()` on the participant's port; the port for the match statement depends on whether the policy is an inbound or outbound policy. For an inbound policy, the `match()` it refers to the participant's virtual port; for an outbound policy, it refers to the participant's physical ports. After this step, AS A's outbound and AS B's inbound policies in Figure 3.1(a) become:

```
PA = (match(port=A1) && match(dstport=80)
            >> fwd(B)) +
     (match(port=A1) && match(dstport=443)
            >> fwd(C))

PB = (match(port=B) && match(srcip={0/1})
            >> fwd(B1)) +
     (match(port=B) && match(srcip={128/1})
            >> fwd(B2))
```

For convenience, we use `match(port=B)` as shorthand for matching on any of B's internal virtual port.

**Enforcing consistency with BGP advertisements.** The second transformation restricts each participant's policy based on the BGP routes exported to the participant. For instance, in Figure 3.1, AS A can only direct traffic with destination prefixes

$p_1$, $p_2$, and $p_3$ to AS B, since AS B did not export a BGP route for $p_4$ or $p_5$ to AS A. The SDX runtime generates a BGP filter policy for each participant based on the exported routes, as seen by the BGP route server. The SDX runtime then inserts these filters inside each participant's outbound policy, according to the forwarding action. If a participant AS A is forwarding to AS B (or C), the runtime inserts B's (or, respectively, C's) BGP filter before the corresponding forwarding action. After this step, AS A's policy becomes:

```
PA' = (match(port=A1) && match(dstport=80) &&
         (match(dstip=p1) || match(dstip=p2) ||
          match(dstip=p3))
       >> fwd(B)) +

       (match(port=A1) && match(dstport=443) &&
         (match(dstip=p1) || match(dstip=p2) ||
          match(dstip=p3) || match(dstip=p4))
       >> fwd(C))
```

AS B does not specify special handling for traffic entering its physical ports, so its policy `PB'` remains the same as `PB`.

**Enforcing default forwarding using the best BGP route.** Each participant's policy overrides the default routing decision for a select portion of the traffic, with the remaining traffic forwarded as usual. Each data packet enters the physical switch with a destination MAC address that corresponds to the BGP next-hop of the participant's best BGP route for the destination prefix. To implement default forwarding, the SDX runtime computes simple MAC-learning policies for each virtual switch. These policies forward packets from one virtual switch to another based on the destination MAC address and forward packets for local destinations on the appropriate physical ports. The default policy for AS A in Figure 3.1(a) is:

```
defA = (match(dstmac=MAC_B1) >> fwd(B)) +
       (match(dstmac=MAC_B2) >> fwd(B)) +
       (match(dstmac=MAC_C1) >> fwd(C)) +

       (match(port=A)  >>
         modify(dstmac=MAC_A1) >> fwd(A1))
```

The first part of the policy handles traffic arriving on A's physical port and forwards traffic to the participant with the corresponding destination MAC address. The second part of the policy handles traffic arriving from other participants and forwards to A's physical port. The runtime also rewrites the traffic's destination MAC address to correspond to the physical port of the intended recipient. For example, in Figure 3.1, A's diverted HTTP traffic for $p_1$ and $p_2$ reaches B with C1 as the MAC address, since C is the designated BGP next-hop for $p_1$ and $p_2$. Without rewriting, AS B would drop the traffic. The runtime then combines the default policy with the corresponding participant policy. The goal is to apply `PA'` on all matching packets and `defA` on all other packets. The SDX controller analyzes `PA'` to compute the union of all match predicates in `PA'` and applies Pyretic's `if_()` operator to combine `PA'` and `defA`, resulting in policy `PA''`.

**Moving packets through the virtual topology.** The SDX runtime finally composes all of the augmented policies into one main SDX policy. Intuitively, when a participant $A$ sends traffic in the SDX fabric destined to participant $B$, $A$'s outbound policy must be applied first, followed by $B$'s inbound policy, which translates to the sequential composition of both policies, (*i.e.*, `PA'' >> PB''`). Since any of the participant can originate or receive traffic, the SDX runtime sequentially composes the *combined* policies of all participants:

```
SDX = (PA'' + PB'' + PC'') >> (PA'' + PB'' + PC'')
```

When the SDX applies this policy, any packet that enters the SDX fabric either reaches the physical port of another participant or is dropped. In any case, the resulting forwarding policy within the fabric will never have loops. Taking BGP policies into account also prevent forwarding loops between edge routers. The SDX enforces two BGP-related invariants to prevent forwarding loops between edge routers. First, a participant router can only receive traffic destined to an IP prefix for which it has announced a corresponding BGP route. Second, if a participant router announces a BGP route for an IP prefix $p$, it will never forward traffic destined to $p$ back to the SDX fabric.

Finally, the SDX runtime relies on the underlying Pyretic runtime to translate the SDX policy to the forwarding rules to install in the physical switch. More generally, the SDX may consist of *multiple* physical switches, each connected to a subset of the participants. Fortunately, we can rely on Pyretic's existing support for topology abstraction to combine a policy written for a single SDX switch with another policy for routing across multiple physical switches, to generate the forwarding rules for multiple physical switches.

### 3.4.2 Reducing Data-Plane State

Augmenting each participant's policy with the BGP-learned prefixes could cause an explosion in the size of the final policy. Today's global routing system has more than 500,000 IPv4 prefixes (and growing!), and large IXPs host several hundred participants (e.g., AMS-IX has more than 600). The participants may have different policies, directing traffic to different forwarding neighbors. Moreover, composing these policies might also generate a "cross-product" of their predicates if the participants' policies match on different fields. For instance, in Figure 3.1a, AS A matches on `dstport`, and B on `srcip`. As a result, a naive compilation algorithm could easily lead to millions

of forwarding rules, while even the most high-end SDN switch hardware can barely hold half a million rules [77].

Existing layer-two IXPs do not face such challenges because they forward packets based only on the destination MAC address, rather than the IP and TCP/UDP header fields. To minimize the number of rules in the SDX switch, the SDX (1) groups prefixes with the same forwarding behavior into an equivalence class and (2) implicitly tags the packets sent by each participant's border router using a virtual MAC address. This technique substantially reduces the number of forwarding rules, and works with unmodified BGP routers.

**Grouping prefixes into equivalence classes.** Fortunately, a participant's policy would typically treat a large number of IP prefixes the same way. For instance, in Figure 3.1, AS A has the same forwarding behavior for $p_1$ and $p_2$ (*i.e.*, send Web traffic via AS B, and send the rest via AS C). By grouping $p_1$ and $p_2$, we could implement the policy with only two forwarding rules, directing traffic to AS B and C, instead of the four currently required. We say that $p_1$ and $p_2$ belong to the same Forwarding Equivalence Class (FEC). An FEC is a set of IP prefixes that share the same forwarding behavior throughout the SDX fabric. Ideally, we would install the minimum set of forwarding rules for each FEC, which is equivalent to the number of forwarding actions associated with the FEC. Doing so requires a new way to combine prefixes; conventional IP prefix aggregation does not work because prefixes $p_1$ and $p_2$ might not be contiguous IP address blocks.

**Offloading tagging to the participants' border routers.** To group non-adjacent prefixes belonging to the same FEC, we introduce the abstraction of a multi-stage Forwarding Information Base (FIB) for each participant, as shown in Figure 3.2. The first table matches on the destination IP prefix and tags packets with the associated FEC. Then, a second table simply matches on the tag and

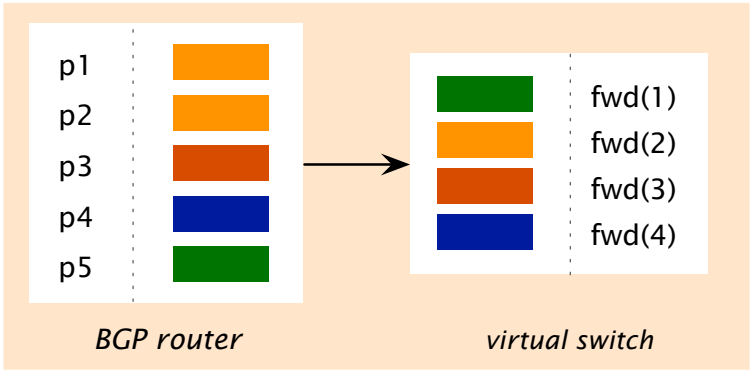



Figure 3.2: Multi-stage FIB for each participant, where the first stage corresponds to the participant's border router and the second stage corresponds to the participant's virtual switch at the SDX.

performs the forwarding actions associated with the FEC. Using a multi-staged FIB substantially reduces the number of rules in the second table. The first table remains quite large because of the many IP prefixes. To address this challenge, we implement the first table using the participant's own border router. Each border router already maintains a forwarding table with an entry for each destination prefix, so we can realize our abstraction without any additional table space! Still, we need (1) a data-plane mechanism for tagging the packets and (2) a control-plane mechanism for the SDX to instruct the border router about which tag to use for each prefix. Ideally, the solution to both problems would be completely transparent to the participants, rather than requiring them to run or configure an additional protocol (e.g., MPLS) for this purpose.

**Using the MAC address as data-plane tag and the BGP next-hop IP address for control-plane signaling.** The SDX runtime capitalizes on how BGP-speaking routers compute forwarding-table entries. Upon choosing a BGP route for a prefix $p$, a router (1) extracts the next-hop IP address from the BGP route announcement, (2) consults its ARP table to translate the IP address to the corresponding MAC address, and (3) installs a forwarding-table entry that sets the destination MAC address before directing the packet to the output port. Usually, this MAC

address corresponds to the physical address of the next-hop interface. In the SDX though, we have the MAC address correspond to a virtual MAC address (VMAC)—the tag—which identifies the FEC for prefix $p$. The SDX fabric can then just match on the VMAC and perform the forwarding actions associated with the FEC. We refer to the BGP next-hop IP address sent to the border router as the *Virtual Next-Hop* (VNH). Finally, observe that we can assign the same VNH (and, hence, the same VMAC) to disjoint IP prefixes—the address blocks need not be contiguous.

In practice, the SDX runtime first pre-computes the FEC according to participant policies and assigns a distinct (VNH, VMAC) pair to each of them. It then transforms the SDX policies to match on the VMAC instead of the destination prefixes. Finally, it instructs the SDX route server to set the next-hop IP address (VNH) in the BGP messages and directs its own ARP server to respond to requests for the VNH IP address with the corresponding VMAC.

**Computing the virtual next hops.** Computing the virtual next-hop IP addresses requires identifying all groups of prefixes that share the same forwarding behavior, considering both default BGP forwarding and specific SDX policies. To ensure optimality, we want the groups of prefixes to be of maximal size; in other words, any two prefixes sharing the same behavior should always belong to the same group. The SDX runtime computes the FECs in three passes.

In the first pass, the SDX runtime extracts the groups of IP prefixes for which the default behavior is affected in the same way by at least one SDX outbound policy. Figure 3.1 shows that the group $\{p_1, p_2, p_3\}$ has its default behavior overridden by AS A's outbound policies, which forward its Web traffic to AS B. Similarly, the group $\{p_1, p_2, p_3, p_4\}$ has its default behavior overridden by AS A's outbound policies, which forward its HTTPS traffic to AS C. All of the prefixes except $p_5$ have their default behavior overridden.

In the second pass, the SDX runtime groups all the prefixes that had their default behavior overridden according to the default next-hop selected by the route server. In the previous example, prefixes $p_1, p_2, p_3, p_4$ will be divided into two groups: $\{p_1, p_2, p_4\}$ whose default next-hop is $C$ and $\{p_3\}$ whose default next-hop is $B$.

In the third pass, the SDX runtime combines the groups from the first two passes into one group $C = \{\{p_1, p_2, p_3\}, \{p_1, p_2, p_3, p_4\}, \{p_1, p_2, p_4\}, \{p_3\}\}\}$. It then computes $C'$ such that each element of $C'$ is the largest possible subset of elements of $C$ with a non-empty intersection. In the example above, $C' = \{\{p_1, p_2\}, \{p_3\}, \{p_4\}\}$ and is the only valid solution. Intuitively, $n$ prefixes belonging to the same group $C_i \in C$ either always appear altogether in a policy $P$, or do not appear at all—they share the same forwarding behavior. We omit the description of a polynomial-time algorithm that computes the Minimum Disjoint Subset (MDS).

Finally, observe that we do not need to consider BGP prefixes that retain their default behavior, such as $p_5$ in Figure 3.1. For these prefixes, the SDX runtime does not have to do any processing and simply behaves like a normal route server, which transmits BGP announcements with the next-hop IP address unchanged.

### 3.4.3 Reducing Control-Plane Computation

In this section, we describe how to reduce the time required for control-plane computation. Many of these operations have a default computation time that is exponential in the number of participants and thus does not scale as the number of participants grows. At a high level, the control plane performs three computation-intensive operations: (1) computing the VNHs; (2) augmenting participants' SDX policies; and (3) compiling the policies into forwarding rules. The controller performs these operations both during initialization and whenever SDX's operational state changes. We focus primarily on optimizing policy compilation, as this step is the most computationally intensive. We first describe optimizations that accelerate the initial computation. We

then describe optimizations that accelerate incremental computation in response to updates (*i.e.*, due to changes in the available BGP routes or the SDX policies). We describe each optimization along with the insight that enables it.

### Optimizing initial compilation

SDX compilation requires composing the policies of every participant AS with every other participant's policy using a combination of sequential and parallel composition. Performing such compositions is time-consuming, as it requires inspecting each pair of policies involved to identify overlaps. As illustration, consider the final policy computed in Section 3.3, without considering default forwarding (for simplicity):

```
policy_composed =
  (PA'' + PB'' + PC'') >> (PA'' + PB'' + PC'')
```

Since the parallel-composition operator is distributive, the compiler can translate the policy into many pairs of sequential composition, combined together using parallel composition. Removing terms that apply the same policy in succession (*i.e.*, `PA'' >> PA''`) yields:

```
policy_composed =
  ((PA'' >> PB'')+(PA'' >> PC''))+
  ((PB'' >> PA'')+(PB'' >> PC''))+
  ((PC'' >> PA'')+(PC'' >> PB''))
```

Compiling this policy requires executing eleven composition operations—six sequential (two per line) and five in parallel—to combine the intermediate results together. Fortunately, a lot of these sequential and parallel composition can be avoided by exploiting three observations: (1) participant policies tend to involve only a subset of the participants; (2) participant policies are disjoint by design; and (3) many policy idioms appear multiple times in the final policy. The first observation reduces the number of sequential composition operations, and the second reduces the number of

parallel composition operations. The third observation prevents compilation of the same policy more than once. With these optimizations, the SDX can achieve policy compilation with only three sequential compositions and no parallel compositions.

**Most SDX policies only concern a subset of the participants.** In the IXP traffic patterns we observe, a few IXP participants carry most of the traffic. Previous work has shown that about 95% of all IXP traffic is exchanged between about 5% of the participants [2]. We thus assume that most SDX policies involve these few large networks rather than all of the IXP participants. The SDX controller avoids all unnecessary compositions by only composing policies among participants that exchange traffic. In this example, AS B has no outbound policy, so compositions `(PB'' >> PA'')` and `(PB'' >> PC'')` are unnecessary. The same reasoning applies for AS C. The SDX controller therefore reduces the policy as follows:

```
policy_composed =
  (PA'' >> PB'') + (PA'' >> PC'') + (PC'' >> PB'')
```

which only involves three sequential composition operations.

**Most SDX policies are disjoint.** Parallel composition is a costly operation that should be used only for combining policies that apply to overlapping flow space. For policies that apply to disjoint flow spaces, the SDX controller can simply apply the policies independently, as no packet ever matches both policies. The policies are disjoint by design because they differ with respect to the virtual switch and port after the first syntactic transformation (*i.e.*, isolation). Also, the same observation applies within the policies of a single participant. We assume that the vast majority of participants would write unicast policies in which each packet is forwarded to one other participant. We do not prevent participants from expressing multicast policies, but we optimize for the common case. As a result, SDX policies that forward to

different participants always differ with respect to the forwarding `port` and are also disjoint by construction.

Returning to the previous example, none of the parallel compositions between `(PA'' >> PC'')`, `(PA'' >> PC'')`, and `(PC'' >> PB'')` are necessary, since each of them always applies on strictly disjoint portions of the flow space.

**Many policy idioms appear more than once in the global policy.** The reuse of various policy idioms results from the fact that participants exchange traffic with each other (and, more often than not, with the same participant). For instance, in an IXP where every participant sends to AS X, AS X's policies would be sequentially composed with all policies. Currently, the Pyretic compiler would recompile the same sub-policy multiple times. It would therefore compile `PA''`, `PB''`, and `PC''` twice. To accelerate compilation, the SDX controller memoizes all the intermediate compilation results before composing the final policy.

### Optimizing incremental updates

SDX compilation occurs not only at initialization time, but also whenever a change occurs in the set of available BGP routes after one or more BGP updates. Efficiently coping with these changes is important. The SDX runtime supports fast recompilation by exploiting three characteristics BGP update patterns: (1) prefixes that are likely to appear in SDX policies tend to be stable; (2) most BGP route changes only affect a small portion of the forwarding table; and (3) BGP route changes occur in bursts and are separated by large periods with no change at all. We draw these observations from a week-long analysis of BGP updates collected at BGP collectors in three of the largest IXPs in the world. Table 3.1 summarizes the data that we used for this analysis.

| | AMS-IX | DE-CIX | LINX |
|---|---|---|---|
| **collector peers/-total peers** | 116/639 | 92/580 | 71/496 |
| **prefixes** | 518,082 | 518,391 | 503,392 |
| **BGP updates** | 11,161,624 | 30,934,525 | 16,658,819 |
| **prefixes seeing updates** | 9.88% | 13.64% | 12.67% |

Table 3.1: IXP datasets. We use BGP update traces from RIPE collectors [96] in the three largest IXPs—AMS-IX, DE-CIX, and LINX—for January 1–6, 2014, from which we discarded updates caused by BGP session resets [131].

Based on these observations, we augmented the basic SDX compilation with an additional compilation stage that is invoked immediately whenever BGP routes change. The main recompilation algorithm is then executed in the background between subsequent bursts of updates. We tune the optimization to handle changes that result from BGP updates, because BGP updates are significantly more frequent than changes to the participants' SDX policies.

**Prefixes that are likely to appear in SDX policies tend to be stable.** Only about 10–14% of prefixes saw any BGP updates at all for an entire week, suggesting that most prefixes are stable. Furthermore, previous work suggests that the stable prefixes are also the same ones that carry the most traffic [93]. Hence, those stable prefixes are also the ones that are likely to be associated with SDX policies.

**Most BGP update bursts affect a small number of prefix groups.** Updates and best path changes tend to occur in bursts. In 75% of the cases, these update bursts affected no more than three prefixes. Over one week, we observed only one update burst that triggered updates for more than 1,000 prefixes. In the common case, the SDX thus only needs to recompute flow table entries for a few affected prefix groups. Even in cases where bursts are large, there is a linear relationship between

the burst size and recompilation time and, as we explain next, this recompilation can occur in the background.

**BGP bursts are separated by large periods with no changes, enabling quick, suboptimal reactions followed by background re-optimization.** We observed that the inter-arrival time between BGP update bursts is at least 10 seconds 75% of the time; half of the time, the inter-arrival time between bursts is more than one minute. Such large inter-arrival times enable the SDX runtime to adopt a two-stage compilation approach, whereby time is traded for space by combining: (1) a fast, but suboptimal recompilation technique, that quickly reacts to the updates; and (2) an optimal recompilation that runs periodically in the background.

The fast stage works as follows. Whenever there is a change in the BGP best path pertaining to a prefix $p$, the SDX immediately creates a new VNH for $p$ and recompiles the policy, considering only the parts related to $p$. It then pushes the resulting forwarding rules into the data plane with a higher priority. The computation is particularly fast because: (1) it bypasses the actual computation of the VNH entirely by simply assuming a new VNH is needed; (2) it restricts compilation to the parts of the policy related to $p$. In Section 3.6, we show that sub-second recompilation is achievable for the majority of the updates. Although the first stage is fast, it can also produce more rules than needed, since it essentially bypasses VNH optimization.

## 3.5 Implementation and Deployment

We now describe the implementation of the SDX controller, as well as our current deployment. We then describe several applications that we have implemented with the SDX. We describe one application with outbound traffic control (application-specific peering) and one with inbound traffic control (wide-area load balance).

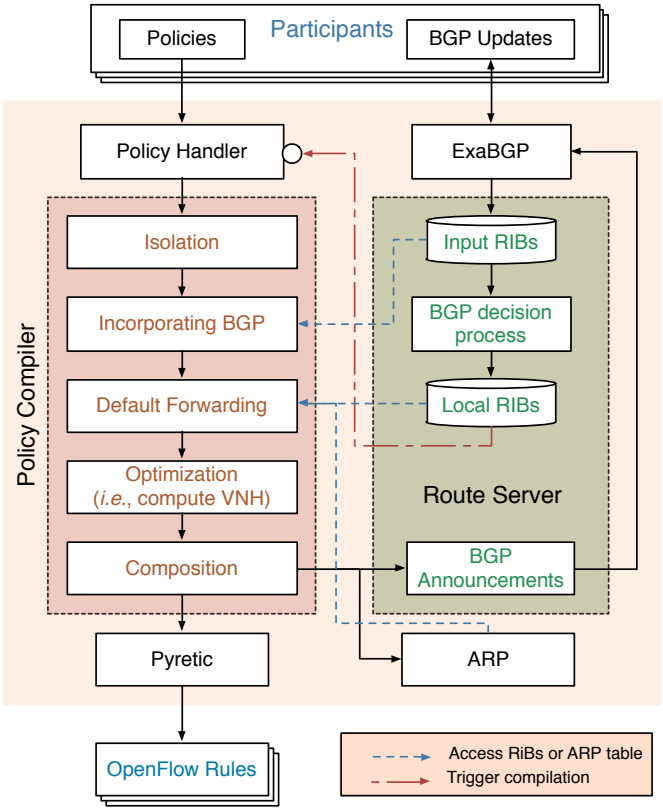



Figure 3.3: The SDX controller implementation, which has two pipelines: a policy compiler and a route server.

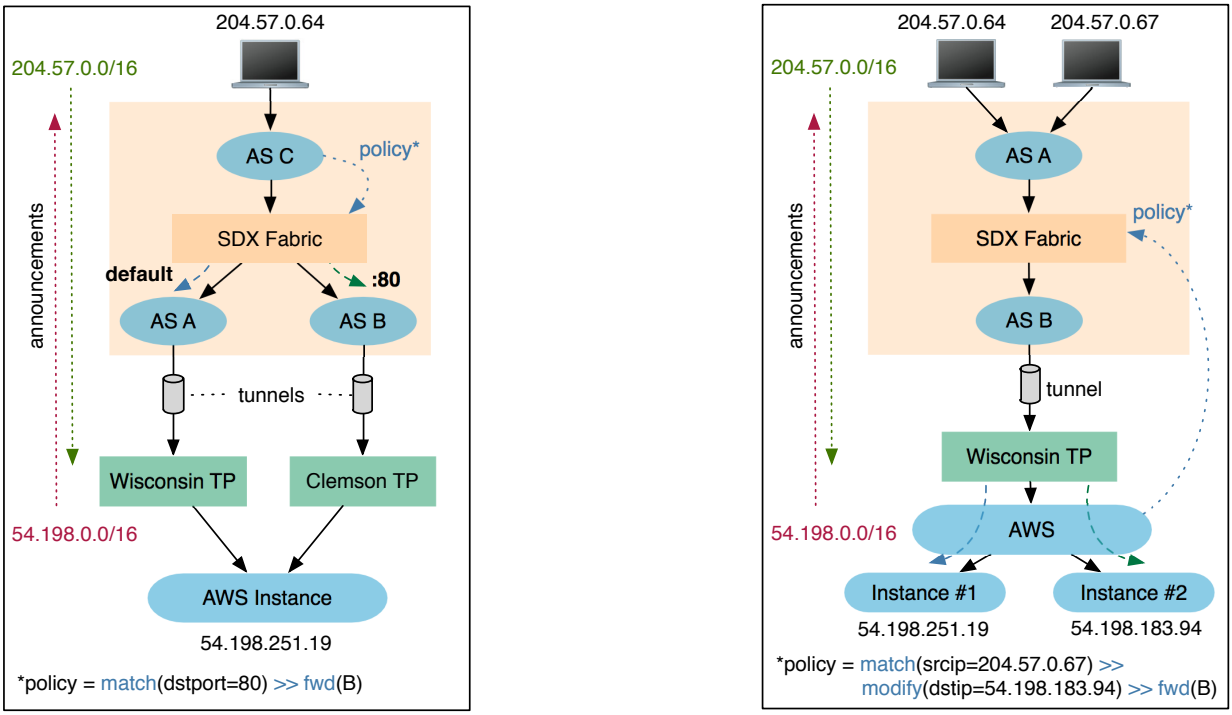
### 3.5.1 Implementation

Figure 3.3 shows the SDX controller implementation, which has two main pipelines: a *policy compiler*, which is based on Pyretic; and a *route server*, which is based on ExaBGP. The policy compiler takes as input policies from individual participants that are written in Pyretic—which may include custom route advertisements from the participants—as well as BGP routes from the route server, and it produces forwarding rules that implement the policies. The route server processes BGP updates from participating ASes and provides them to the policy compiler and re-advertises BGP routes to participants based on the computed routes. We briefly describe the steps of each of these functions below.

**SDX policy compiler.** The policy compiler is a Pyretic process that compiles participant policies to forwarding rules. Based on the virtual SDX abstraction from the SDX configuration (*i.e.*, the static configuration of which ASes are connected to each other at layer two), the policy compiler isolates the policies that each AS writes by augmenting each policy with a `match` statement based on the participant's `port`. The compiler then restricts each participant's outbound policies according to the current BGP routing information from the route server and rewrites the participant policies so that the switch can forward traffic according to the default BGP policies. After augmenting the policies, the compiler then computes VNH assignments for the advertised prefixes. Finally, the compiler writes the participant policies where necessary, taking care to avoid unnecessary composition of policies that are disjoint and performing other optimizations such as caching of partial compilations, as described in Section 3.4.3. It then passes the policies to the Pyretic compiler, which generates the corresponding forwarding rules.

Because VNHs are virtual IP addresses, the controller also implements an ARP responder that responds to ARP queries for VNHs with the appropriate VMAC addresses.

**SDX route server.** We implemented the SDX route server by extending ExaBGP [31], an existing route server that is implemented in Python. As in other traditional route servers [82, 12], the SDX route server receives BGP advertisements from all participants and computes the best path for each destination prefix on behalf of each participant. The SDX route server also (1) enables integration of the participant's policy with interdomain routing by providing advertised route information to the compiler pipeline; and (2) reduces data-plane state by advertising virtual next hops for the prefixes advertised by SDX participants. The SDX route server recompiles the participants' policies whenever a BGP update results in changes to




(a) Application-Specific Peering.

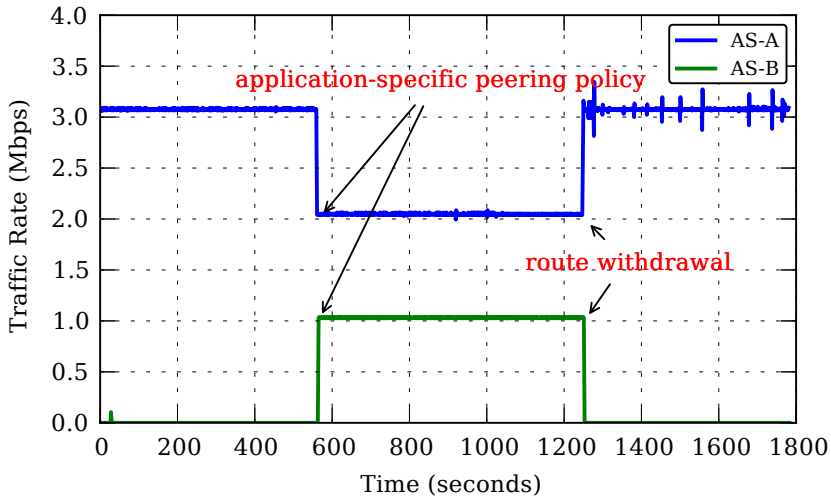
(b) Wide-Area Load Balance.

Figure 3.4: Setup for deployment experiments.

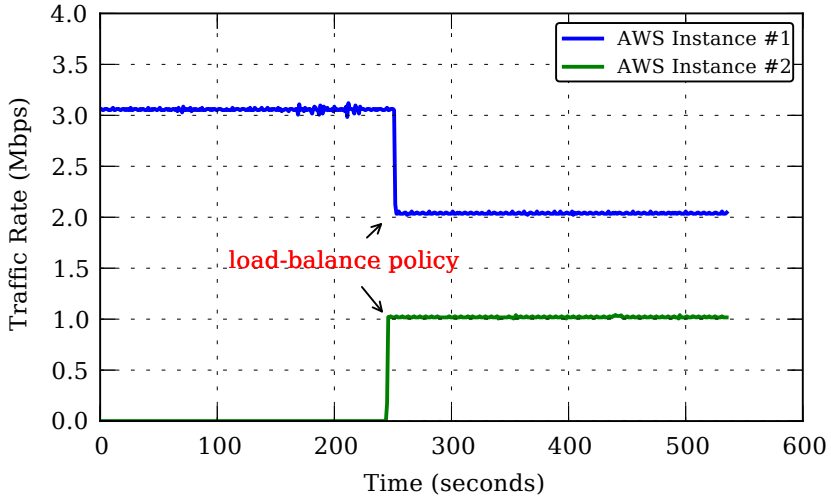
best routes for a prefix. When such an update occurs, the route server sends an event to the policy handler, which recompiles policies associated with the affected routing updates. The compiler installs new rules corresponding to the BGP update while performing the optimizations described in Section 3.4.3 in the background. After compiling the new forwarding rules, the policy compiler then sends the updated next-hop information to the route server, which marshals the corresponding BGP updates and sends them to the appropriate participant ASes.

### 3.5.2 Deployment

We have developed a prototype of the SDX [100] and a version that can be deployed using virtual containers in Mininet [45]. Figure 3.4 shows two setups that we have created in these environments for the purposes of demonstrating two applications: application-specific peering and wide-area load balance. For each use case, we explain




(a) Application-specific peering.




(b) Wide-area load balance.

Figure 3.5: Traffic patterns for the two "live" SDX applications. (a) At 565 seconds, the AS C installs an application-specific peering policy, causing port 80 traffic to arrive via AS B. At 1253 seconds, AS B withdraws its route to AWS, causing all traffic to shift back to the path via AS A. (b) At 246 seconds, the AWS network installs a wide-area load balance policy to shift the traffic for source 204.57.0.67 to arrive at AWS instance #2.

the deployment setup and demonstrate the outcome of the running application. For both use cases, we have deployed an SDX controller (including route server) that is connected to an Open vSwitch software switch. The ASes that we have connected

to the Open vSwitch at the exchange point are currently virtual (as our deployment has no peers that carry real Internet traffic), and these virtual ASes in turn establish BGP connectivity to the Internet via the Transit Portal [119]. The client generates three 1 Mbps UDP flows, varying the source and destination IP addresses and ports as required for the demonstrations below.

**Application-specific peering.** Figure 3.4a shows an SDX setup where we test the application-specific peering use-case described in Section 3.3. The example demonstrates several features of the SDX controller, including (1) the ability for a participant to control traffic flows based on portions of flow space other than destination IP prefix (e.g., port number); and (2) the SDX controller's ability to guarantee correct forwarding that is in sync with the advertised BGP routes.

Transit Portal deployments at the University of Wisconsin and at Clemson University both receive a route to the Amazon prefix hosting our Amazon Web Services (AWS) instance. They distribute their routes to AS A and AS B, respectively. These ASes in turn send announcements to the SDX controller, which then selects a best route for the prefix, which it re-advertises to AS C. AS C's outbound traffic then flows through either AS A or AS B, depending on the policies installed at the SDX controller.

AS C, the ISP hosting the client, installs a policy at the SDX that directs all traffic to the Amazon /16 IP prefix via AS A, except for port 80 traffic, which travels via AS B. To demonstrate that the SDX controller ensures that the switch data plane stays in sync with the BGP control plane messages, we induce a withdrawal of the route announcement at AS B (emulating, for example, a failure). At this point, all traffic from the SDX to AWS travels via AS A. Figure 3.5a shows the traffic patterns resulting from this experiment and the resulting traffic patterns as a result

of (1) installation of the application-specific peering policy; (2) the subsequent BGP route withdrawal.

**Wide-area load balancer.** The wide-area load balancer application also demonstrates the ability for a remote network to install a policy at the SDX, even if it is not physically present at the exchange. Figure 3.4b shows an SDX setup where an AWS tenant hosts destinations in two distinct AWS instances and wishes to balance load across those two destinations. The AWS tenant remotely installs a policy that rewrites the destination IP address for traffic depending on the source IP address of the sender. Initially, traffic from the clients of AS A directed towards the AWS tenant's instances traverses the SDX fabric unchanged and routed out to the Internet via AS B. After the AWS tenant installs the load-balance policy at the SDX, traffic that was initially destined only for AWS instance #1 is now balanced across both of the AWS instances. Figure 3.5b shows the traffic rates from the resulting experiment and how they evolve when the load balance policy is installed at the SDX. Although this deployment has only one SDX location, in practice the AWS tenant could advertise the same IP prefix via multiple SDX locations as an anycast announcement, thus achieving more control over wide-area load balance from a distributed set of locations.

## 3.6 Performance Evaluation

We now demonstrate that, under realistic scenarios, the SDX platform scales—in terms of forwarding-table size and compilation time—to hundreds of participants and policies.

### 3.6.1 Experimental Setup

To evaluate the SDX runtime, we provide realistic inputs to our compiler. We instantiate the SDX runtime with no underlying physical switches because we are not

concerned with evaluating forwarding performance. We then install policies for hypothetical SDX participants, varying both their numbers and their policies. We derive policies and topologies from the characteristics of three large IXPs: AMS-IX, LINX, and DEC-IX. We repeat each experiment ten times.

**Emulating real-world IXP topologies.** Based on the characteristics of existing IXPs, we define a few static parameters, including the fraction of participants with multiple ports at the exchange, and the number of prefixes that each participant advertises. For example, at AMS-IX, approximately 1% of the participating ASes announce more than 50% of the total prefixes, and 90% of the ASes combined announce less than 1% of the prefixes. We vary the number of participants and prefixes at the exchange.

**Emulating realistic AS policies at the IXP.** We construct an exchange point with a realistic set of participants and policies, where each participant has a mix of inbound and outbound policies. Inbound policies include inbound traffic engineering, WAN load balancing, and redirection through middleboxes. Outbound policies include application-specific peering, as well as policies that are intended to balance transit costs. Different types of participants may use different types of policies. To approximate this policy assignment, we classify ASes as eyeball, transit, or content, and we sort the ASes in each category by the number of prefixes that they advertise. Since we do not have traffic characteristics, we use advertised prefixes as a rough proxy. Only a subset of participants exchange most of the traffic at the IXPs, and we assume that most policies involve the participants who carry significant amounts of traffic. We assume that the top 15% of eyeball ASes, the top 5% of transit ASes, and a random set of 5% of content ASes install custom policies:

*Content providers.* We assume that content providers tune outbound traffic policies for the top eyeball networks, which serve as destinations for the majority of

traffic flows. Thus, for each content provider, we install outbound policies for three randomly chosen top eyeball networks. Occasionally, content providers may wish to redirect incoming requests (e.g., for load balance), so each content provider installs one inbound policy matching on one header field.

*Eyeballs.* We assume that eyeball networks generally tune inbound traffic, and, as a result, most of their policies involve controlling inbound traffic coming from the large content providers. The eyeball networks install inbound policies and match on one randomly selected header field; they do not install any outbound policies. For each eyeball network, we install inbound policies for half of the content providers.

*Transit providers.* Finally, we assume that transit networks have a mix of inbound and outbound traffic-engineering policies to balance load by tuning the entry point. In our experiment, each transit network installs outbound policies for one prefix group for half of the top eyeball networks and installs inbound policies proportional to the number of top content providers. Again, the inbound policies match on one header field that we select at random, and outbound policies match on destination prefix group plus one additional header field.

In the following subsections, we show that the required forwarding rules and compilation time scale proportionally with the total number of policies for each participant.

### 3.6.2 Forwarding-Table Space

We first evaluate the number of prefix groups to implement a particular SDX policy, given a certain number of participants and prefixes. We then quantify the number of flow rules that result from a given number of prefix groups.

**Number of prefix groups.** We estimate the number of prefix groups (and hence, VNHs) that result when the participant ASes at the SDX apply policies to a certain

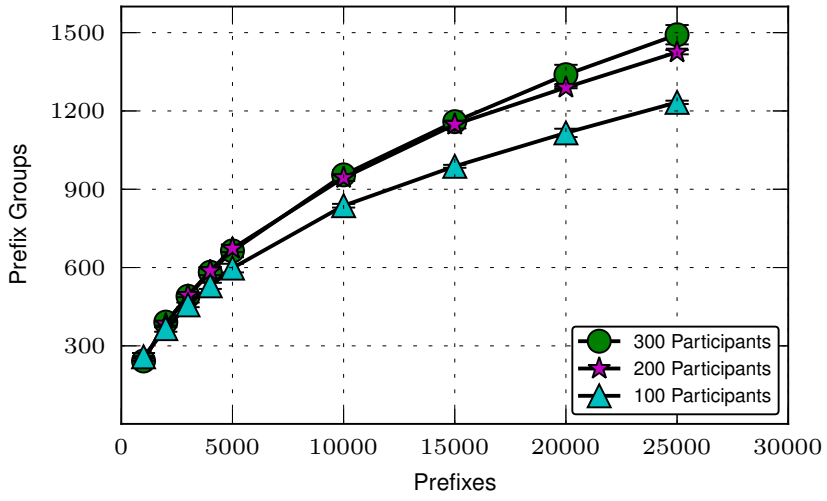



Figure 3.6: Number of prefix groups as a function of the number of prefixes, for different numbers of participants.

number of prefixes. When policies involve portions of flow space other than destination IP address, the number of prefix groups can be larger than the number of participants times the number of next-hop IP addresses at the exchange, since the resulting policies can create more forwarding equivalence classes.

To study the relationship between the number of prefixes and the number of prefix groups, we consider the approximately 300 ASes at AMS-IX which announce more than one prefix (about half of all ASes at the exchange). The results are similar for other large IXPs. Each experiment has two parameters, $N$ and $x$, defining the set of ASes that participate (the top $N$ by prefix count, for $N \in \{100, 200, 300\}$) and the set of prefixes with SDX policies ($|p_x| = x \in [0, 25000]$, selected at random from the default-free routing table). In a given experiment, for AS $i \in [1, \ldots, N]$, let $p_i$ be the set of prefixes announced by AS $i$, and let $p'_i = p_i \cap p_x$. We then run the minimum disjoint subset algorithm over the collection $P' = \{p'_1, \ldots, p'_N\}$, yielding the set of prefix groups.

Figure 3.6 shows that the number of prefix groups is sub-linear in the number of prefixes. As the number of prefixes to which SDX policies are applied increases, more

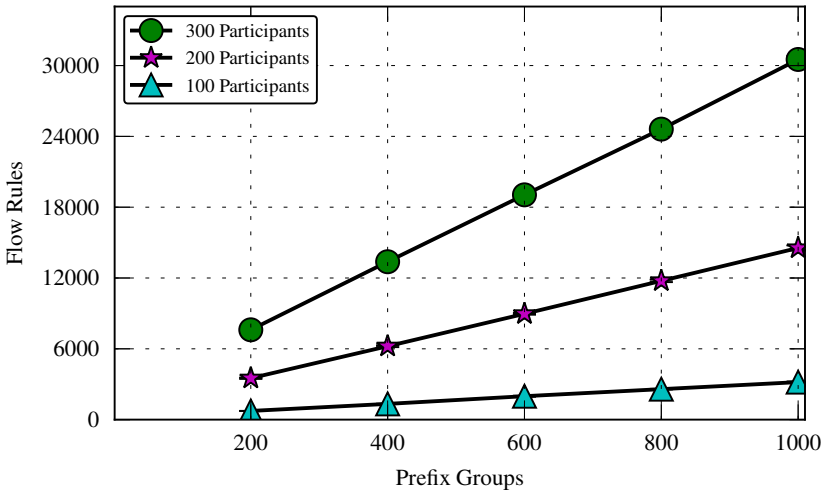



Figure 3.7: The number of forwarding rules as a function of the number of prefix groups for different number of participants.

prefixes are advertised by the same number of participants, thereby increasing the likelihood that the advertised prefixes are part of the same forwarding equivalence class. We also note that the number of prefix groups is significantly smaller than the number of prefixes, and that the ratio of prefix groups to prefixes decreases as the number of prefixes increases, indicating good scaling properties.

**Number of forwarding rules.** Figure 3.7 shows how the number of forwarding rules varies as we increase the number of prefix groups, for different numbers of participants. We select the number of prefix groups based on our analysis of the prefix groups that might appear in a typical IXP (Figure 3.6). We run the experiment as described above, selecting participant ASes according to common policies at IXPs. The number of forwarding rules increases roughly linearly with the number of prefix groups. Because each prefix group operates on a disjoint portion of the flow space, the increase in forwarding rules is linear in the number of prefix groups.

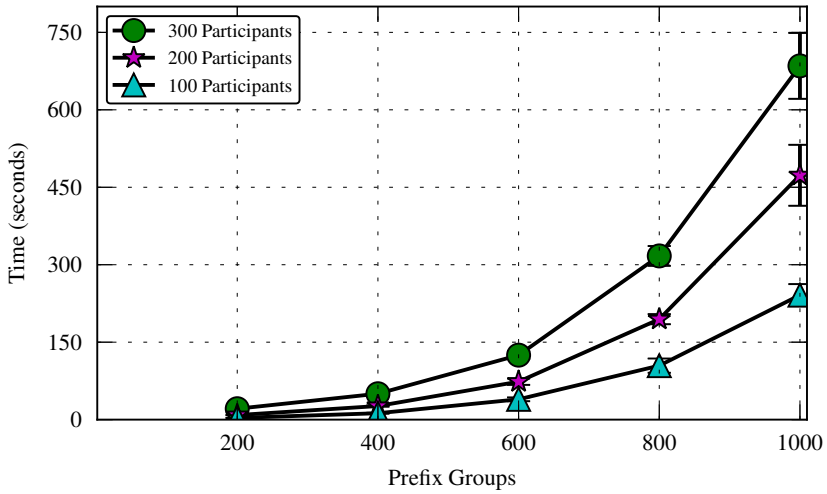



Figure 3.8: Compilation time as a function of the number of prefix groups, for different numbers of participants.

### 3.6.3 Compilation Time

We measure the compilation time for two scenarios: (1) *initial compilation time*, which measures the time to compile the initial set of policies to the resulting forwarding rules; and (2) *incremental compilation time*, which measures how long it takes to recompute when changes occur.

**Initial compilation time.** Figure 3.8 shows how the time to compute low-level forwarding rules from higher-level policies varies as we increase both the number of prefix groups and IXP participants. The time to compute the forwarding rules is on the order of several minutes for typical numbers of prefix groups and participants. The results also show that compilation time increases roughly quadratically with the number of prefix groups. The compilation time increases more quickly than linearly because, as the number of prefix groups increases, the interactions between policies of pairs of participants at the SDX also increases. The time for the SDX to compute VNHs increases non-linearly as the number of participants and prefix groups increases.

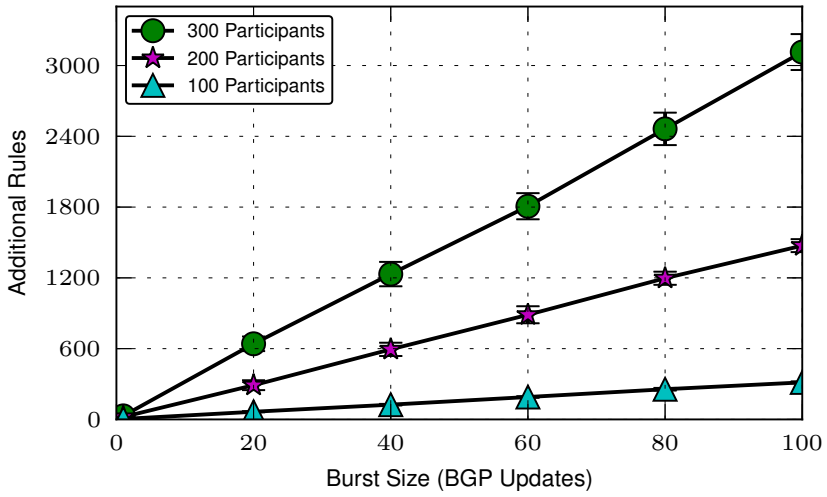



Figure 3.9: Number of additional forwarding rules.

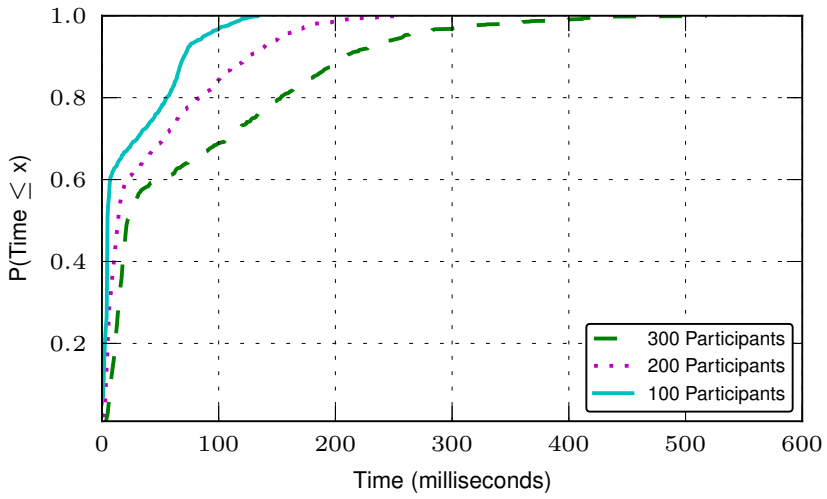



Figure 3.10: Time to process a single BGP update for various participants.

We observed that for 1,000 prefix groups and 100 participants, VNH computation took about five minutes.

As discussed in Section 3.4.3, the SDX controller achieves faster compilation by memoizing the results of partial policy compilations. Supporting caching for 300 participants at the SDX and 1,000 prefix groups could require a cache of about 4.5 GB. Although this requirement may seem large, it is on the order of the amount of memory required for a route server in a large operational IXP today.

**Incremental compilation time.** Recall that in addition to computing an initial set of forwarding table rules, the SDX controller must recompile them whenever the best BGP route for a prefix changes or when any participant updates its policy. We now evaluate the benefits of the optimizations that we discussed in Section 3.4.3 in terms of the savings in compilation time. When new BGP updates arrive at the controller, the controller must recompute VNH IP addresses for the affected routes to establish new prefix groups.

Figure 3.9 shows the number of additional rules that are generated when a "burst" of BGP updates of a certain size arrives. These rules must reside in the forwarding table until the SDX controller recomputes the minimum disjoint set. The figure represents a worst-case scenario, whereby each BGP update results in a change to the best path and, hence, an additional VNH in the table, causing a number of additional forwarding rules that depends on the number of participants with policies installed. In practice, as we discussed in Section 3.4.3, not every BGP update induces changes in forwarding table entries. When a BGP update arrives, the SDX controller installs additional flow table rules for the affected flows and computes a new optimized table *in the background* to ultimately coalesce these flows into the smaller, minimal forwarding tables. As shown in Figure 3.10, re-computing the tables takes less than 100 milliseconds most of the time.

## 3.7 Related Work

We briefly describe related work in SDN exchange points, interdomain route control, and policy languages for SDNs.

**SDN-based exchange points.** The most closely related work is Google's Cardigan project [123], which shares our broad goal of using SDN to enable innovation at IXPs. Cardigan runs a route server based on RouteFlow [97] and uses an OpenFlow switch

to enforce security and routing policies. The Cardigan project is developing a logical SDN-based exchange point that is physically distributed across multiple locations. Unlike the SDX in this paper, Cardigan does not provide a general controller for composing participant policies, offer a framework that allows IXP participants to write policies in a high-level language, or introduce techniques for scaling to handle a large number of participants and policies.

**Interdomain route control.** Previous work on applying SDN to interdomain routing has focused on how to use the separation of data and control planes to improve the manageability of routing within a single AS [51, 47]. Similarly, earlier work such as the Routing Control Platform (RCP) developed a BGP route controller for influencing route selection within a single AS and enabled various functions, such as re-routing traffic within an AS in the event of attack or traffic surge [17]. These systems apply SDN to help operators route interdomain traffic more efficiently within an AS, but they do not provide a way for *multiple* ASes to independently define policies which can then be composed into a single coherent forwarding policy for forwarding traffic *between* ASes. Previous work has also proposed outsourcing end-to-end path selection to third parties with an SDN controller [60, 63], but unlike SDX, these systems require ASes to modify their existing routing infrastructure.

**Policy languages for SDNs.** SDX takes advantage of recent advances in programming languages for SDNs that allow operators to express policies at a higher level of abstraction than flow rules [121, 70, 35]. In particular, Pyretic provides both topology abstraction and composition operators that we take advantage of when implementing the SDX policy compiler. It is worth pointing out, of course, that these languages only make it possible to implement something like the SDX—as discussed in Section 3.5, Pyretic is merely the language that we use to encode SDX policies, but the controller must first perform syntactic transformation and incorporate BGP

routing information to ensure forwarding according to AS policies that is congruent with the BGP routes that the SDX participants advertise.

# Chapter 4

# Network Control with iSDX

In this chapter, we present the design, implementation, and evaluation of the network-control system, iSDX, that improves the scalability of the SDX system further. More specifically, this system build up on the SDX system to leverage the arbitrary bitmask matching capability of the programmable switches and develops a new attribute-encoding algorithm, enabling iSDX to scale participants' flexible control programs with the number of IXP participants.

## 4.1 Overview

In Chapter 3, we presented an initial design of SDX and showed how introducing SDN functionality at even a single IXP could catalyze new traffic-management capabilities, ranging from better inbound traffic engineering to application-specific peering and server load balancing.

The earlier deployments [106, 68, 10, 64] of SDX, including the one we presented in Chapter 3, remain relatively small-scale or limited in scope because the currently available switch hardware cannot support large forwarding tables and because efficiently combining the policies of independently operated networks as routes and policies change presents a significant scaling challenge.

In this chapter, we tackle these scalability challenges with the design and implementation of iSDX, an industrial-scale SDX that can support interconnection for the largest IXPs on the Internet today. We design mechanisms that allow the number of participants, BGP routes, and SDN policies to scale, even for the limited table sizes of today's switches. We develop algorithms for compiling traffic-control policies at the scale and speed that these IXP would require. We have implemented these algorithms in Ryu [98], a widely used SDN controller. We have released our implementation to the public with documentation and tutorials.

In the design and implementation of iSDX, we address two scalability challenges that are fundamental to *any* SDX design. The first challenge relates to how the control plane combines the policies of individual networks into forwarding entries in the data plane. Compiling traffic control policies expressed in a higher-level policy language to forwarding table entries can be slow since this process involves composing the policies of multiple participants into a single coherent set of forwarding-table entries. This slow process is exacerbated by the fact that any change to BGP routing may change forwarding behavior; SDX designs, including the one presented in Chapter 3, trigger recompilation every time a BGP best route changes, which is not tractable in practice. The main scalability challenge thus involves efficiently composing the policies of individual participants and ensuring that the need to recompile the forwarding table entries is wholly decoupled from (frequent) BGP routing changes.

To scale the control plane, we introduce a new design that exploits the fact that each participant expresses its SDN policy independently, which implies that each participant can also compile its SDN policies independently, as well. This change enables more aggressive compression of the forwarding tables than is possible when all of the policies are compressed together and also allows for participant policies to be compiled in parallel. As a result, iSDX compiles the forwarding tables two orders

of magnitude faster than the existing approaches; the tables are also two orders of magnitude smaller, making them suitable for practical hardware-switch deployments.

The second challenge relates to the data plane: the number of forwarding table entries that might go into the forwarding table at an IXP switch can quickly grow unacceptably large. Part of the challenge results from the fact that the policies that each network writes have to be consistent with the BGP routes that each participant advertises, to ensure that an SDN policy cannot cause the switch to forward traffic on a path that was never advertised in BGP. This process significantly inflates the number of forwarding table entries in the switch and is a considerable deployment hurdle. Large industrial-scale IXPs can have over 700 participants exchanging traffic for hundreds of thousands of prefixes; combined with the fact that each of these participants may now introduce policies for specific traffic flows, the number of forwarding table entries quickly becomes intractable. Although the design presented in Chapter 3 reduced the size of the forwarding tables, we show that the size of these tables remained prohibitively large for industrial-scale deployments.

To address the data-plane challenge, we introduce an efficient encoding mechanism where the IXP fabric forwards the packet based on an opaque tag that resides in the packet's destination MAC field. This tag explicitly encodes both the next-hop for the packet and the set of ASes that advertise BGP routes for the packet's destination, thus making it possible to remove this information from the switch tables entirely. This separation prevents BGP routing updates from triggering recomputation and recompilation of the forwarding table entries. Using features in OpenFlow 1.3 that support matching on fields with arbitrary bitmasks, we significantly reduce the size of this table by grouping tags with common bitmasks.

In summary, we present the following contributions in this chapter:

- The design and implementation of iSDX, the first SDX controller that scales to large industrial-scale IXPs. We devised new mechanisms for distributing

control-plane computation, compressing the forwarding tables, and responding to BGP routing changes, reducing the compilation time and forwarding table size by several orders magnitude. (Sections 4.3–4.5)

- A public, open-source implementation of iSDX, publicly available on Github [48]; the system is based on Ryu, a widely used SDN controller, and is accompanied with tutorials and instructions that have already helped spur early adoption. (Section 4.6)
- An extensive evaluation of iSDX's scalability characteristics using a trace-driven evaluation from one of the largest IXPs in the world. Our evaluation both demonstrates that iSDX can scale to the largest IXPs and provides insight into specifically how (and to what extent) each of our optimizations and algorithms helps iSDX scale. (Section 4.7)

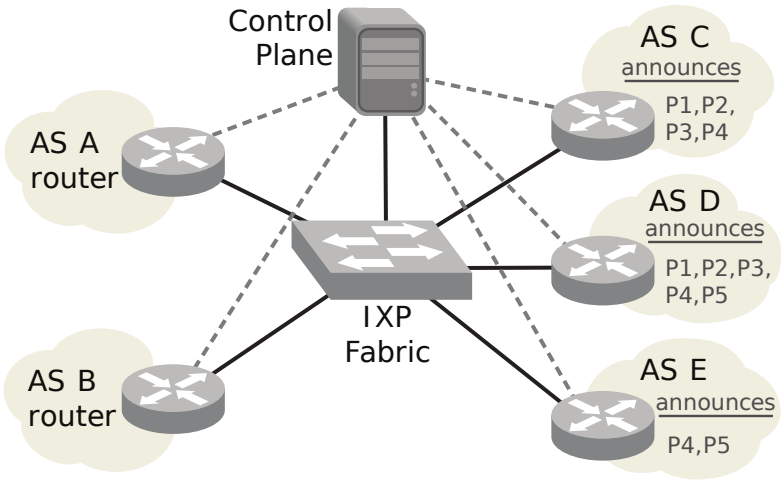
We survey the related work in Section 4.8.

## 4.2 SDX: Scaling Challenges

We begin with a demonstration that the designs presented in Chapter 3 cannot scale to the production settings of a large IXP.

### 4.2.1 Example Operation

Figure 4.1a shows an example topology with five participants; Figure 4.1b shows the routes advertised to $A$ and $B$ and the BGP routes that they select for each prefix (in bold). Both $A$ and $B$ express outbound policies. To ensure that SDN policies cause the IXP to forward traffic in a way that is consistent with the advertised BGP routes, the SDX controller *augments* each outbound policy with the reachability information. Intuitively, augmentation restricts forwarding policies so that traffic is forwarded only on paths that correspond to BGP routes that the participant has learned.




(a) Example Topology

| | A | B |
|---|---|---|
| P1 | C, **D** | **C**, D |
| P2 | C, **D** | **C**, D |
| P3 | C, **D** | **C**, D |
| P4 | C, **D**, E | C, D, **E** |
| P5 | D, **E** | D, **E** |

(b) Reachability and Next Hops (in bold) for AS A and AS B

Figure 4.1: An example with five IXP participants. Two participants AS $A$ and AS $B$ have outbound policies. The other three advertise five IP prefixes to both these participants.

For example, suppose that $A$ has the following outbound policies:

`dPort=443` $\rightarrow$ `fwd`$(C)$

`dPort=22` $\rightarrow$ `fwd`$(C)$

`dPort=80` $\wedge$ `sIp=10/24` $\rightarrow$ `fwd`$(D)$

`dPort=80` $\wedge$ `sIp=40/24` $\rightarrow$ `fwd`$(D)$

These policies forward traffic based on values of packet header fields, overriding BGP behavior. For instance, the first policy specifies HTTPS traffic (`dPort=443`) should be forwarded to $C$. Without augmentation, $A$ would also forward the HTTPS traffic destined for prefix $P5$ to $C$, even though $C$ never advertised a path for $P5$ to $A$. In our example, $A$'s policies are then augmented as follows:

`dIp` $\in \{\mathbf{P1}, \mathbf{P2}, \mathbf{P3}, \mathbf{P4}\} \wedge$ `dPort=443` $\rightarrow$ `fwd`$(C)$

`dIp` $\in \{\mathbf{P1}, \mathbf{P2}, \mathbf{P3}, \mathbf{P4}\} \wedge$ `dPort=22` $\rightarrow$ `fwd`$(C)$

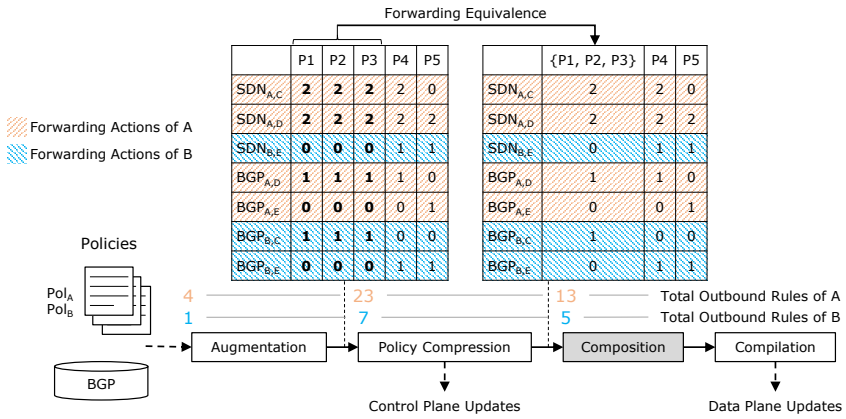



Figure 4.2: Matrix representation of AS A and AS B's outbound policies after augmentation and policy compression, as well as the stages of compression and composition in the original SDX design; the composition stage is grey to indicate that the Sonata eliminates this stage entirely.

$\texttt{dIp} \in \{\mathbf{P1}, \mathbf{P2}, \mathbf{P3}, \mathbf{P4}, \mathbf{P5}\} \wedge \texttt{dPort=80} \wedge \texttt{sIp=10/24} \rightarrow \texttt{fwd}(D)$

$\texttt{dIp} \in \{\mathbf{P1}, \mathbf{P2}, \mathbf{P3}, \mathbf{P4}, \mathbf{P5}\} \wedge \texttt{dPort=80} \wedge \texttt{sIp=40/24} \rightarrow \texttt{fwd}(D)$ Augmentation enforces that the destination IP (`dIp`) matches one of the prefixes that either $C$ or $D$ announces to $A$, therefore ensuring congruence with BGP routing. Observe that a straightforward realization of this policy requires one distinct match-action rule for each of the five prefixes. Hence, the augmented policies would result in 18 forwarding rules instead of the four rules necessary to implement the original policy.

Similarly, if $B$'s outbound policy is:

$\texttt{dPort=443} \rightarrow \texttt{fwd}(E)$

the SDX controller augments the policy, doubling the number of necessary rules, as follows:

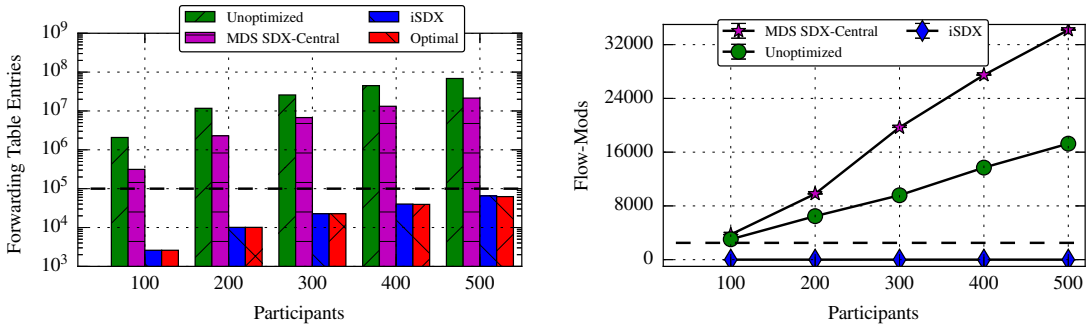
$\texttt{dIp} \in \{\mathbf{P4}, \mathbf{P5}\} \wedge \texttt{dPort=443} \rightarrow \texttt{fwd}(E)$

To better illustrate the scalability challenge, we capture the expansion of the switch forwarding tables using an *augmentation matrix* (Figure 4.2, left matrix). In this matrix, a row labeled as $\text{SDN}_{X,Y}$ refers to an SDN policy written by $X$ that results in traffic being forwarded to $Y$, while columns refer to IP prefixes. The value of an element $(i, j)$ indicates the number of forwarding table entries (*i.e.*, match-action

rules) in participant $i$'s policy where prefix $j$ appears. Similarly, $\text{BGP}_{X,Y}$ indicates whether $X$ selects $Y$ as the next hop for some BGP-advertised prefix, and element $(i, j)$ is 1 if participant $A$ selects the route advertised by $B$ for the prefix corresponding to column $j$.

For example, the element in row $\text{SDN}_{A,C}$ and column $P1$ reflects the fact there are two forwarding table entries that correspond to prefix $P1$: one for traffic with `dPort=443` and one for traffic with `dPort=22`. The same applies for columns $P2$, $P3$, and $P4$. We can determine the total number of forwarding table entries (and the number contributed by each participant) by summing up the corresponding elements in the matrix. We will use this notation to describe compression techniques (and their effects) throughout the paper.

**Previously developed compression techniques.** Intuitively, the number of forwarding rules increases as the number of SDX participants with outbound policies increases (more rows) and as forwarding policies are defined for additional prefixes (more columns). To limit the number of forwarding rules, the original SDX design (Chapter 3) identified the Minimum Disjoint Set (MDS) of prefixes (columns) with the same SDN policies and grouped each equivalent set into a Forwarding Equivalence Class (FEC). In the rest of this paper, we refer to this algorithm as *MDS compression*. For instance, in the preceding example, prefixes $P1, P2, P3$ belong to the same FEC, as indicated by the boldface entries in the left matrix in Figure 4.2. MDS compression reduces the number of forwarding table entries by assigning a virtual next-hop to each FEC, rather than to each individual prefix. Figure 4.2 also depicts the number of forwarding table entries before and after MDS compression. In particular, MDS compression reduces the number of columns from the total number of prefixes (5) to the number of FECs (3).




(a) Number of Forwarding Table Entries.

(b) Data-Plane Update Rate.

Figure 4.3: Existing SDX designs can require to maintain millions of forwarding entries (left) and update 10,000s of updates per second (right). Such numbers are far from current hardware capabilities. As an illustration, the dashed line highlights the hardware capabilities of state-of-the-art SDN switches [78].

| | **Unoptimized** | **Centralized MDS-SDX (Chapter 3)** | **iSDX** |
|---|---|---|---|
| Time (s) | 4572.15 | 1740.93 | 2.82 |

Table 4.1: Median time (for 60 trials) to compute forwarding table entries for an IXP with 500 participants. The iSDX column shows the results for this paper.

### 4.2.2 Existing SDX Designs Do Not Scale

In this section, we show that existing SDX designs do not scale to the demands of industrial-scale IXPs. We explore two different state-of-the-art SDX designs: (1) an *unoptimized SDX* that does not compress policies, such as that used by Google's Cardigan SDX [123]; (2) a simple, centralized SDX controller that applies MDS compression, as in Chapter 3. We also preview the results from this paper, showing that our new architecture, iSDX, reduces the compilation time, number of forwarding table entries, and data-plane update rate by more than two orders of magnitude, thus making operation in an industrial-scale IXP practical. In each case, we evaluate the time to compute the forwarding table entries, the number of forwarding table entries, and the rate at which changes in BGP routing information induce changes in the forwarding table entries. We use a real BGP trace from one the largest IXPs in the world for this evaluation. Section 4.7 provides details about our experiment setup.

**Existing SDX designs can take minutes to compute forwarding table entries.** Table 4.1 shows the median time over 60 trials to compute forwarding table entries for an IXP with 500 participants for two state-of-the-art SDX designs, as well as for iSDX, the design that we present in this paper. iSDX reduces the average time to compute forwarding table entries from 30 minutes to *less than three seconds*.

**Existing SDX designs can require millions of forwarding table entries.** Figure 4.3a shows how the number of forwarding table entries increases as the number of participants increases from 100 to 500. MDS compression reduces the number of entries by an order of magnitude, but the forwarding table is still too large for even the most high-end hardware switches, which have about $100,000$ TCAM entries [78]. The iSDX design ensures that the number of forwarding table entries is approximately the number of SDN policies that each participant expresses (shown as "optimal" in Figure 4.3a), thus allowing the number of forwarding table entries to be in the tens of thousands, rather than tens of millions.

**Existing SDX designs require hundreds of thousands updates per second to the data plane.** Figure 4.3b shows the worst-case data-plane update rate that an SDX controller must sustain to remain consistent with existing BGP updates. The update rates of existing designs are several orders of magnitude above what even top-of-the-line hardware switches can support [78] (*i.e.*, about 2,500 updates per second). In constrast, iSDX usually eliminates forwarding table updates in response to BGP updates.

## 4.3 Design of an Industrial-Scale SDX

We introduce the design of an industrial-scale SDX (iSDX), which relies on two principles to reduce compilation time, the number of forwarding table entries, and forwarding table update rates.

### 4.3.1 Partition Control-Plane Computation

**Problem: Considering all policies together reduces opportunities for compression.** Centralized SDX controllers perform control-plane computations for all IXP participants. Doing so not only forces the controller to process a large single combined policy, it also creates dependencies between the policies of individual IXP participants. For example, a change to any participant's inbound policy triggers the recompilation of the policies of *all* participants who forward traffic to that participant. This process requires significant computation and also involves many (and frequent) updates to the forwarding table entries at the IXP switch.

**Solution: Partition computation across participants.** We solve this problem by partitioning the control-plane computation across participants. Doing so ensures that participant policies stay independent from each other. In addition, partitioning the computation enables more efficient policy compression by operating on smaller state, reducing both computation time and data plane state. Partitioning the control-plane computation among participants also enables policy compilation to scale out as the number of IXP participants and routes grows. Section 4.4 details this approach.

### 4.3.2 Decouple BGP and SDN Forwarding

**Problem: Frequent BGP updates trigger recompilation.** Coupling BGP and SDN policies during compilation inflates the number of resulting forwarding table

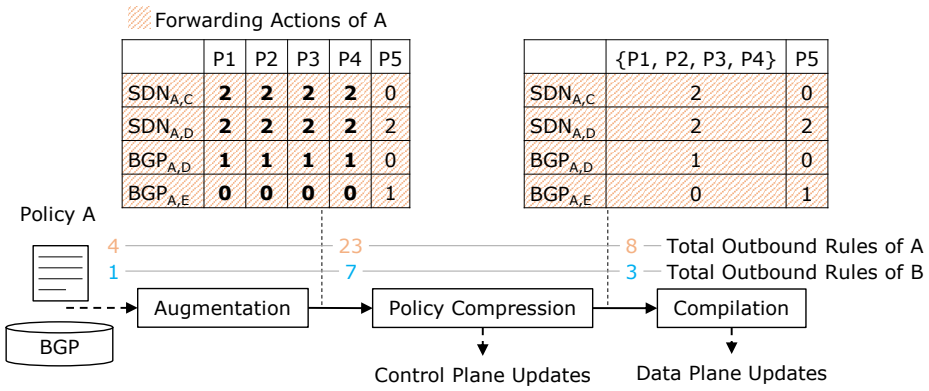



Figure 4.4: Partitioning the Control-Plane Computation.

entries and also implies that any change to BGP routing triggers recompilation of the forwarding table entries, which is costly. Our previous design partially addressed this problem, but this design still requires millions of flow rules in the data plane as shown in Figure 4.3a. Additionally, our previous approach to reduce the number of forwarding table entries *increases* the forwarding table update rates, since any change in BGP routing may affect how entries are compressed.

**Solution: Encode BGP reachability information in a separate tag.** We address this problem by encoding all information about BGP best routes (and corresponding next hops) into the destination MAC addresses, which reduces the number of forwarding table entries, as well as the number of changes to the forwarding table when BGP routes change. Section 4.5 discusses our approach in detail.

## 4.4 Partitioning Control-Plane Computation

To achieve greater compression of the rule matrix, we need to reduce the constraints that determine which prefixes belong to the same FEC. Rather than computing one set of equivalence classes for the entire SDX, iSDX computes *separate FECs for each participant*. We first discuss how partitioning by participant reduces the size of the rule matrices and, as a side benefit, allows for faster computation. We then describe

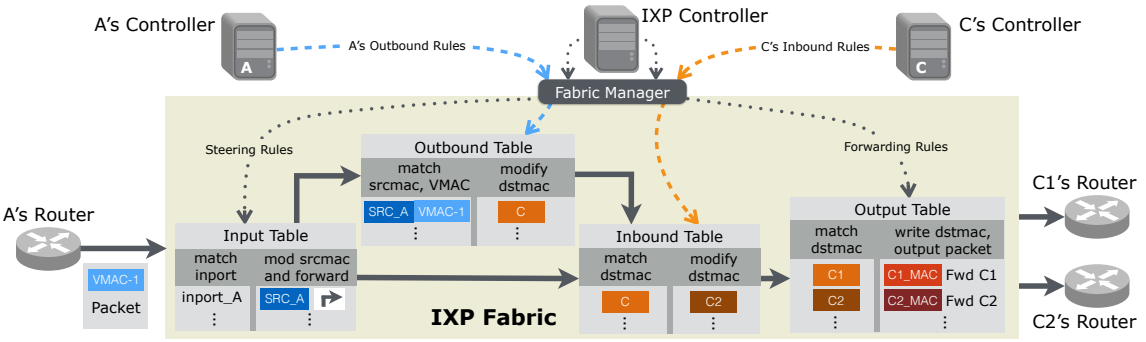



Figure 4.5: Distributing forwarding rules and tags.

how we use multiple match-action tables and ARP relays to further improve scalability, setting the stage for further optimizations in Section 4.5.

### 4.4.1 Partitioning the FEC Computation

Figure 4.4 shows similar compression and compilation steps as the ones done in Figure 4.2, with the important distinction that it takes place on behalf of participant *A* only; similar operations take place on behalf of other participants. Figure 4.4 highlights two important benefits of partitioning the computation of FEC across participants:

- Computing separately for each participant reduces the number of next-hops, leading to a smaller number of larger forwarding equivalence classes. In Figure 4.4, the number of columns reduces from five to two.
- The computational complexity of computing FECs is proportional to the number of rows times the number of columns in the rule matrix. Now, each rule matrix is smaller, and the computation for different participants can be performed in parallel.

In practice, the SDX controller could compute the FECs for each participant, or each participant could run its own controller for computing its own FECs. In the rest

of the paper, we assume each participant runs its own controller for computing its FECs.

### 4.4.2 Distributing Forwarding Rules and Tags

In addition to computing the FECs for each participant, the iSDX must realize these policies in the data plane.

**Decomposing the IXP fabric into four tables:** To forward traffic correctly, an SDX must combine the inbound and outbound policies for all of the participants. Representing the combination of policies in a single forwarding table, as in an OpenFlow 1.0 switch, would be extremely expensive. Some existing SDN controllers perform this type of composition [70, 103]—essentially computing a cross product of the constituent policies—and, in fact, earlier we followed followed this approach as presented in Chapter 3. Computing the cross product leads to an explosion in the number of rules, and significant recomputation whenever one of the participant policies changes.

Fortunately, modern switches have multiple stages of match-action tables, and modern IXPs consist of multiple switches. The iSDX design capitalizes on this trend. The main challenge is to determine *how* to most effectively map policies to the underlying tables.

A strawman solution would be to use a two-table pipeline, where packets first enter an outbound table implementing outbound policies for the participant where the traffic originates, followed by an inbound table that applies inbound policies for the participant that receives the traffic as it leaves the IXP fabric. Using only two tables, however, would mean that some of these tables would need to be much larger; for example, the outbound table would need to represent the cross product of all input ports and outbound policies. Additionally, using only two tables makes it more difficult to scale-out the iSDX as the number of participants grows.

As such, our design incorporates an input table, which handles all the incoming traffic and tags it with a new source MAC address based on the packet's incoming port, so that packets can be multiplexed to the outbound table. As the packet leaves the iSDX, it passes through an output table, which looks up the packet's tag in the destination MAC field and both performs the appropriate action and rewrites the packet's destination MAC address. Separate input and output tables provide a cleaner separation of function between the modules that write to each table, avoid cross-product explosion of policies, and facilitates scale-out by allowing the inbound and outbound tables to reside on multiple physical switches in the IXP infrastructure. (Such scale-out techniques are beyond of the scope of this paper.)

Figure 4.5 shows how the IXP fabric forwards a packet, while distributing the compilation and compression of policies across separate tables. Based on the destination IP address of the packet, suppose that AS $A$'s controller selects a route to the packet's destination via AS $D$; this route will correspond to a next-hop IP address. AS $A$'s controller will make a BGP announcement advertising this path. AS $A$'s router will issue an ARP query for the advertised next-hop IP address, and then AS $A$'s controller will respond via the ARP relay setting a virtual MAC address (in Figure 4.5, "VMAC-1") as the packet's destination MAC address.

When the packet enters the IXP fabric, the input table matches on the packet's incoming port and rewrites the source MAC address to indicate that the packet arrived from AS $A$ ("SRC_A"). If $A$ has an outbound policy, the packet will match on ("SRC_A"), and the outbound table will apply an outbound policy. If $A$ has no outbound policy for this packet, the input table forwards the packet directly to the inbound table without changing the destination MAC. This bypass is not strictly necessary but avoids an additional lookup for packets that do not have a corresponding outbound policy. $A$'s outbound policy thus overwrites default BGP forwarding decision and modifies the destination MAC address to "C". The inbound

table rewrites the tag to correspond to the final disposition of the packet ("C1" or "C2"), which is implemented in the output table. The output table also rewrites the tag to the receiver's physical MAC address before forwarding.

**Reducing ARP traffic overhead.** Partitioning the FEC computation reduces the number of FECs per participant, but may increase the *total* number of FECs across all participants (*i.e.*, the number of columns across all rule matrices). To reduce the size of the forwarding tables, each data packet carries a tag (*i.e.*, a virtual MAC address) that identifies its FEC. The participant's border router learns the virtual MAC address through an ARP query on the BGP next-hop IP address of the associated routes. The use of broadcast for ARP traffic, combined with the larger number of next-hop IP addresses, could overwhelm the border routers and the IXP fabric. In fact, today's IXPs are already vulnerable to high ARP overheads [16].

Fortunately, we can easily reduce the overhead of ARP queries and responses, because each participant needs to learn about only the virtual MAC addresses for its own FECs. As such, the SDX can turn ARP traffic into *unicast* traffic by installing the appropriate rules for handling ARP traffic in switches. In particular, each participant's controller broadcasts a gratuitous ARP response for every virtual next-hop IP address it uses; rules in the IXP's fabric recognize the gratuitous ARP broadcasts and ensure that they are forwarded only to the relevant participant's routers. Participants' routers can still issue ARP queries to map IP addresses to virtual MAC addresses, but the fabric intercepts these queries and redirects them to an ARP relay to avoid overwhelming other routers.

## 4.5 Decoupling SDN Policies from Routing

To ensure correctness, any SDX platform must combine SDN policies with dynamic BGP state: which participants have routes to each prefix (*i.e.*, valid next-hop ASes

for a packet with a given destination prefix), as well as the next-hop AS to use for each prefix (*i.e.*, the outcome of BGP decision process). The large number of prefixes and participants creates scalability challenges with respect to forwarding table sizes and update rates, before SDN policies even enter the equation.

### 4.5.1 Idea: Statically Encode Routing

To reduce the number of rules and updates, we develop a new encoding scheme that is analogous to source routing: The IXP fabric matches on a tag that is provisioned by a participant's SDX controller. To implement this approach, we optimize the tag that the fabric uses to forward traffic (as described in Section 4.4) to carry information about both the next-hop AS for the packet (as determined by the best BGP route) and the ASes who have advertised routes to the packet's destination prefix. If no SDN policy matches a packet, iSDX can simply match on the next-hop AS bits of the tag to make a default forwarding decision. As before, the sender discovers this tag via ARP.

To implement default forwarding, the IXP fabric maintains static entries for each next-hop AS which forward to participants based upon the next-hop AS bits of the tag. When the best BGP routes change, the entries need not change, rather the next-hop AS bits of the tags change.

To account for changes in available routes, SDN policies that reroute to some participant $X$ confirm whether $X$ has advertised a route before forwarding. The method of checking for $X$ in the tags is static, meaning that in contrast to our previous design (Chapter 3), BGP updates induce zero updates in the IXP switch data plane. Instead, BGP updates result in tag changes, and the participant's border router learns these dynamic tags via ARP.

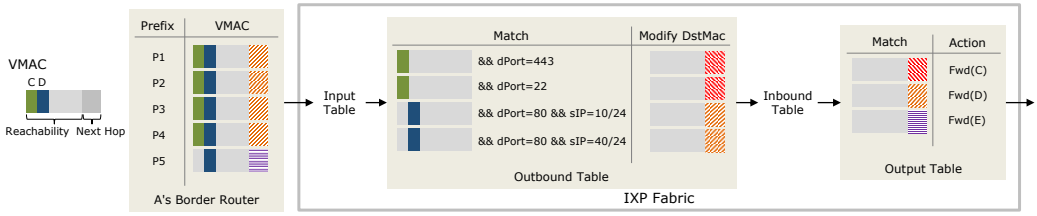



Figure 4.6: How AS $A$'s controller uses reachability encoding to reduce the number of flow rules.

### 4.5.2 Encoding Next-Hop and Reachability

We now describe how iSDX embeds both the next-hop AS (*i.e.*, from the best BGP route) and the reachability information (*i.e.*, the set of ASes that advertise routes to some prefix) into this tag.

#### Next-hop encoding

The next-hop information denotes the default next-hop AS for a packet, as determined by BGP. In the example from Section 4.2.1, $A$'s next-hop AS for traffic to $P1$ as determined by the best BGP route is $D$. iSDX allocates bits from the tag (*i.e.*, the virtual MAC, which is written into the destination MAC of the packet's header) to denote this next-hop. If no SDN policy overrides this default, iSDX applies a default priority prefix-based match on these bits to direct traffic to the corresponding next-hop.[1] This approach reduces the forwarding table entries in a participant's outbound table, since additional entries for default BGP forwarding no longer need to be represented as distinct entries in the forwarding table. Encoding the next hop information in this way requires $\lg(N)$ bits, where $N$ is the number of IXP participants. At a large IXP with up to 1024 participants, ten bits can encode information about default next-hop ASes, leaving 37 bits.[2]

[1]The OpenFlow 1.3 standard supports this feature [79], which is already implemented in many hardware switches (e.g., [78, 83]).

[2]One of the 48 bits in the MAC header is reserved for multicast.

### Reachability encoding

We now explain how to encode reachability information into the remaining 37 bits of the destination MAC address. We first present a strawman approach that illustrates the intuition before describing the scalable encoding.

**Strawman encoding.** Suppose that for a given tag, the $i$-th bit is 1 if that participant learns a BGP route to the corresponding prefix (or prefixes) via next-hop AS $i$. Such an encoding would allow the IXP fabric to efficiently determine whether some participant could forward traffic to some next-hop AS $i$, for any $i$ at the IXP. Considering the example in Section 4.2.1, $A$'s outbound policies are:

`dMac = XX1X...X` $\wedge$ `dPort=443` $\rightarrow$ `fwd`$(C)$

`dMac = XX1X...X` $\wedge$ `dPort=22` $\rightarrow$ `fwd`$(C)$

`dMac = XXX1X..X` $\wedge$ `dPort=80` $\wedge$ `sIp=10/24` $\rightarrow$ `fwd`$(D)$

`dMac = XXX1X..X` $\wedge$ `dPort=80` $\wedge$ `sIp=40/24` $\rightarrow$ `fwd`$(D)$

where `X` stands for a wildcard match (0 or 1). This encoding ensures correct interoperation with BGP, yet we use just four forwarding table entries, which is fewer than the 18 required using augmentation (from the original example in Section 4.2).

Figure 4.6 explains how this approach reduces the number of forwarding table entries in the switch fabric. When a packet arrives, its virtual MAC encodes both (1) which ASes have advertised a BGP route for the packet's destination ("reachability") and (2) the next-hop participant corresponding to the best BGP route ("next hop"). Suppose that a packet is destined for $P1$ from $A$; in this case, $A$'s border router will affix the virtual MAC as shown. If that virtual MAC does not match any forwarding table entries in the outbound table, the packet will simply be forwarded to the appropriate default next hop (in this case, $D$) based on the next-hop encoding. This process makes it possible for the switch to forward default BGP traffic without installing any rules in the outbound table, significantly reducing the size of this table.

**Hierarchical encoding.** The approach consumes one bit per IXP participant, allowing at most for only 37 IXP participants. To encode more participant ASes in these 37 bits, we divide this bitspace hierarchically. Suppose that an IXP participant has SDN policies that refer to $N$ other IXP participants (*i.e.*, possible next-hop ASes). Then, all of these $N$ participants need to be efficiently encoded in the 37-bit space, $B$. We aim to create $W$ bitmasks $\{B_1, B_2, \ldots, B_W\}$ that minimize the total number of forwarding table entries, subject to the limitations of the total length of the bitmask.

Given $M$ prefixes and $N$ IXP participants, we begin with $M$ bitmasks, where each bitmask encodes some *set* of participants that advertise routes to some prefix $p_i$. We greedily merge pairs of sets that have at least one common participant, and we always merge two sets if one is a subset of the other. Iterating over all feasible merges has worst-case complexity $O(M^2)$; and there may be as many as $M-1$ merge actions in the worst case. Each merge has complexity $O(N)$, which gives us an overall worst-case running time complexity of $O(M^3N)$.

Given 37 spare bits in the destination MAC for reachability encoding, if a participant has defined SDN policies for more that 37 participants who advertise the same prefix, then the number of bits required to encode the reachability information will exceed 37. Our analysis using a dataset from one of the largest IXPs in the world found that the maximum number of participants advertising the same prefix was only 27, implying that largest bitmask that this encoding scheme would require is 27 bits. There were 62 total bitmasks, meaning 6 bits are required to encode the ID of a bitmask, requiring a total of 33 bits for the encoding. Using a different (or custom) field in a packet header might also be possible if these numbers grow in the future.

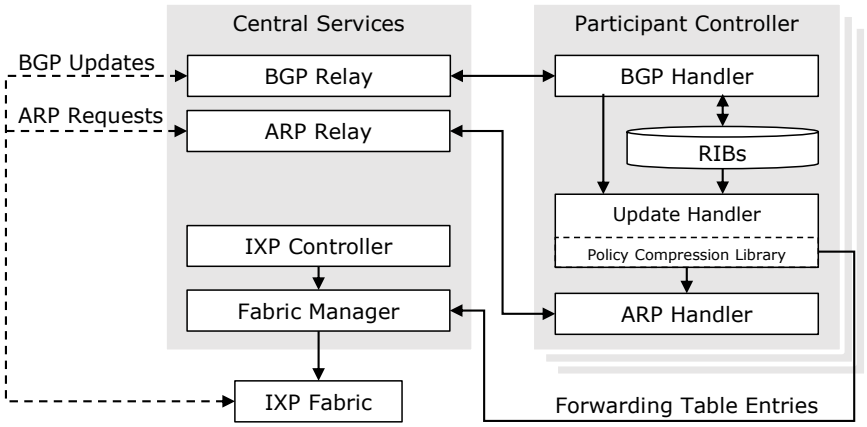



Figure 4.7: Implementation of iSDX. It has five main modules: (1) IXP controller, (2) participant SDN controller, (3) ARP relay, (4) BGP relay, and (5) fabric manager.

## 4.6 Implementation

We now describe an implementation of iSDX, as shown in Figure 4.7. Our Python-based implementation has about 5,000 lines of code. The source code is publicly available on Github [48] along with tutorials describing how to run it. We have also provided instructions describing how to deploy and test iSDX over hardware switches [49]. About 300 students used an earlier version of iSDX in the Coursera SDN course from Summer 2015 [22].

The **fabric manager** is based on Ryu [98]. It listens for forwarding table modification instructions from the participant controllers and the IXP controller and installs the changes in the switch fabric. The fabric manager abstracts the details of the underlying switch hardware and OpenFlow messages from the participant and the IXP controllers and also ensures isolation between participants.

The **IXP controller** installs forwarding table entries in the input and output tables in the switch fabric via the fabric manager. Because all of these rules are static, they are computed only at initialization. Moreover, the IXP controller handles ARP queries and replies in the fabric and ensures that these messages are forwarded to the respective participants' controllers via **ARP relay**.

The **BGP relay** is based on ExaBGP [31] and is similar to a BGP route server in terms of establishing peering sessions with the border routers. Unlike a route server, it does not perform any route selection. Instead, it multiplexes all BGP routes to the participant controllers.

Each **participant SDN controller** computes a compressed set of forwarding table entries, which are installed into the inbound and outbound tables via the fabric manager, and continuously updates the entries in response to the changes in SDN policies and BGP updates. The participant controller receives BGP updates from the BGP relay. It processes the incoming BGP updates by selecting the best route and updating the RIBs. We developed APIs to use either of MongoDB [69], Cassandra [5] and SQLite [104] for storing participants' RIBs. We used the MongoDB (in-memory) for the evaluation in Section 4.7. The participant controller also generates BGP announcements destined to the border routers of this participant, which are sent to the routers via the BGP relay.

Each participant controller's **update handler** determines whether the inbound and outbound tables need to be updated, as well as whether new gratuitous ARP messages must be sent to the participant's border routers to update any virtual destination MAC addresses. The controller receives ARP requests from the participant's border routers via the **ARP handler** and determines the corresponding ARP reply. The controller also receives SDN policy updates from the network operators in the form of addition and removal lists. Both the update handler and the ARP handler use a policy compression library that we implemented, which provides the mapping between IP prefixes and virtual next-hop IPs (corresponding to best BGP routes), and between virtual next-hop IPs and virtual destination MAC addresses (*i.e.*, an ARP table).

| | MDS | NH Encoding | Reachability Encoding |
|---|---|---|---|
| **iSDX-D** | ✓ | ✗ | ✗ |
| **iSDX-N** | ✓ | ✓ | ✗ |
| **iSDX-R** | ✗ | ✓ | ✓ |

Table 4.2: Three distributed SDX Controllers.

| | | | iSDX | | |
|---|---|---|---|---|---|
| | Unoptimized | Centralized MDS-SDX [42] | iSDX-D | iSDX-N | iSDX-R |
| Number of Forwarding Table Entries | 68,476,528 | 21,439,540 | 763,000 | 155,000 | 65,250 |
| Policy Compression Time (s) | N/A | 297.493 | 0.0629 | 0.111 | 2.810 |

Table 4.3: Summary of evaluation results for iSDX with 500 IXP participants. Note that compression times for iSDX are per-participant, since each participant can compile policies in parallel; even normalizing by this parallelization still yields significant gains.

## 4.7 Evaluation

We now demonstrate that iSDX can scale to the forwarding table size, data plane update rate, and control plane computation requirements of an industrial-scale IXP. Table 4.2 summarizes the three different iSDX designs that we compare to previous approaches: iSDX-D applies the same MDS compression technique as in our previous work [42], but with tables distributed across participants; iSDX-N additionally encodes the next-hop AS in the tag; and iSDX-R encodes both the next-hop AS and BGP reachability information in the tag.

Table 4.3 summarizes our results: *iSDX reduces the number of forwarding table entries for an industrial-scale IXP by three orders of magnitude as compared to an unoptimized, centralized SDX design; and by more than two orders of magnitude over the state-of-the-art SDX design [42].* This section explains these results in detail.

### 4.7.1 Experiment Setup

We use data sets from one of the largest IXPs worldwide, which interconnects more than 600 participants, who peer with one another via a BGP route server[94]. We had access to the RIB table dump collected from the IXP's route server on August 8, 2015

for 511 IXP participants. These datasets contain a total of 96.6 million peering (*i.e.*, non-transit) routes for over 300,000 distinct prefixes. We also use a trace of 25,676 BGP update messages from these participants to the route server for the two hours following the collection of this RIB table dump (the participants' RIBs are naturally not perfectly aligned, since dumping a BGP table of about 36 GB from the router takes about fifteen minutes). Our data set does not contain any user data or any personal information that identifies individual users. We run our experiments on a server installed at this IXP configured with 16 physical cores at 3.4 GHz and 128 GB of RAM.

This IXP does not use a programmable IXP fabric, so we assume how participants *might* specify SDN policies, as described in Section 4.2.1. Specifically, *each participant* has between one to four outbound policies for each of 10% of the total participants. The number of policies and set of participants are chosen uniformly at random. Our sensitivity analysis on this percentage shows that our results are influenced in magnitude but the underlying trends remain. Note that this setup is more taxing than the one in our previous work [42] where only 20% of the total participants had any SDN policies at all. We also evaluate iSDX's performance for smaller IXPs by selecting random subsets of IXP participants (ranging from 100 to 500 ASes) and considering only the RIB information and BGP updates for those participants. We also repeated experiments using public RIB dumps and BGP updates collected by RIPE's RIS servers from 12 other IXPs [95]. As the observed workload was much smaller in this case, we omit these results for brevity.

### 4.7.2 Steady-State Performance

We first evaluate the steady-state performance of iSDX. To do so, we use the RIB dumps to initialize the SDX controller (multiple of them for the distributed case) and

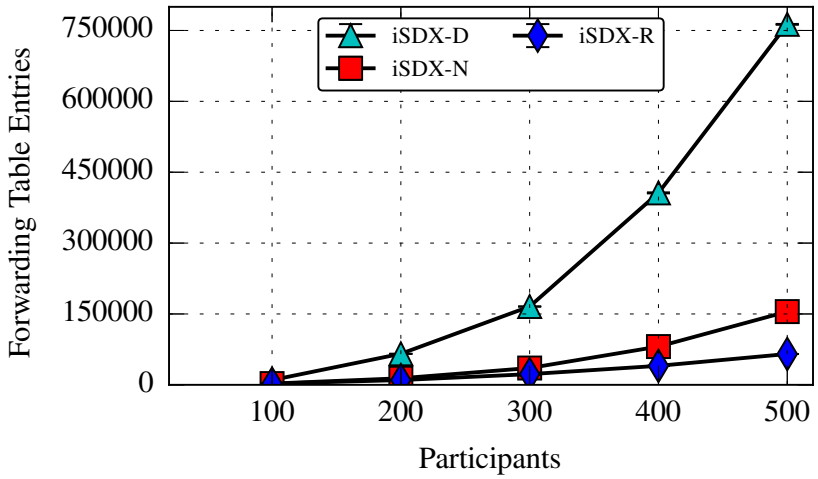



Figure 4.8: Number of forwarding table entries.

evaluate the overall performance in terms of the efficiency of data-plane compression, and the time to compile policies and compress them into smaller forwarding tables.

**Efficiency of compression.** Figure 4.8 shows the number of forwarding table entries for the three distributed controllers: iSDX-D, iSDX-N, and iSDX-R. The number of forwarding table entries increases with the increasing number of IXP participants. Each of our techniques progressively improves scalability. We observe that the number of forwarding table entries for iSDX-R is very close to the lower bound (*i.e.*, best case), where the number of forwarding table entries is equal to the number of SDN policies.

We also explore the effects of distributing the control plane computation on the ability of iSDX to perform MDS compression. The results are shown in Figure 4.9. Given 500 participants, partitioning the control plane reduces the number of next hop entries for the border router from 25,000 to 360. This reduction mitigates the load on the border routers, since the number of virtual next hop IP addresses reflects the number of ARP entries each participant's border router must maintain.

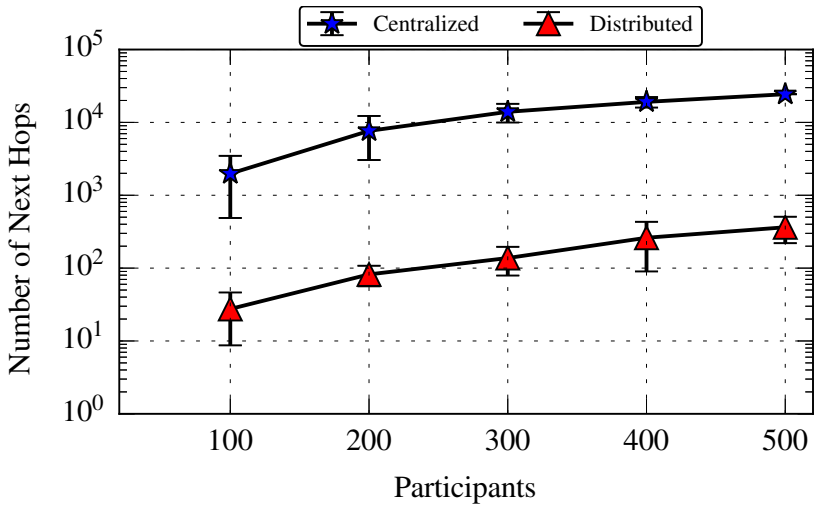



Figure 4.9: Number of virtual next-hop IP addresses for centralized and distributed control planes. Results for distributed iSDX do not depend on encoding or compression approach.

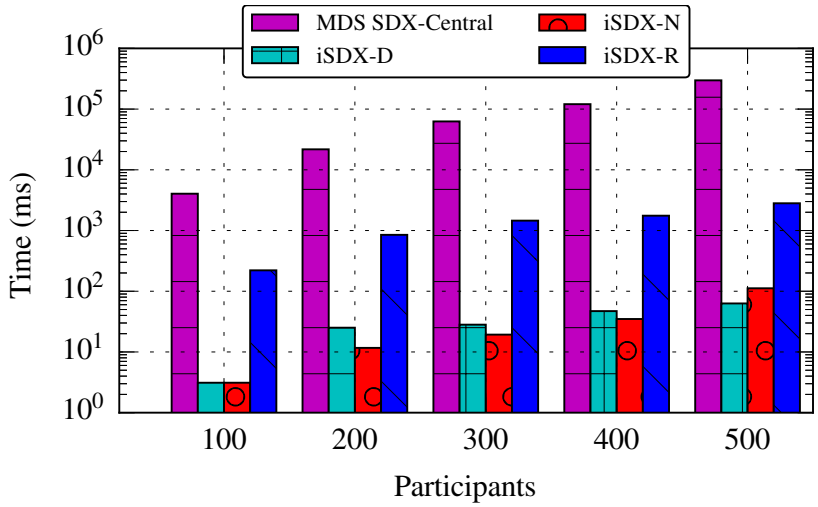



Figure 4.10: Time to perform policy compression.

**Time to perform policy compression.** Figure 4.10 shows the compression time for each controller; this time dominates control-plane computation but only occurs at initialization. The Centralized MDS-SDX operates on a large input rule matrix, and thus requires nearly five minutes to compress policies. iSDX-D distributes the computation across participants, reducing compression time by three orders of magnitude.

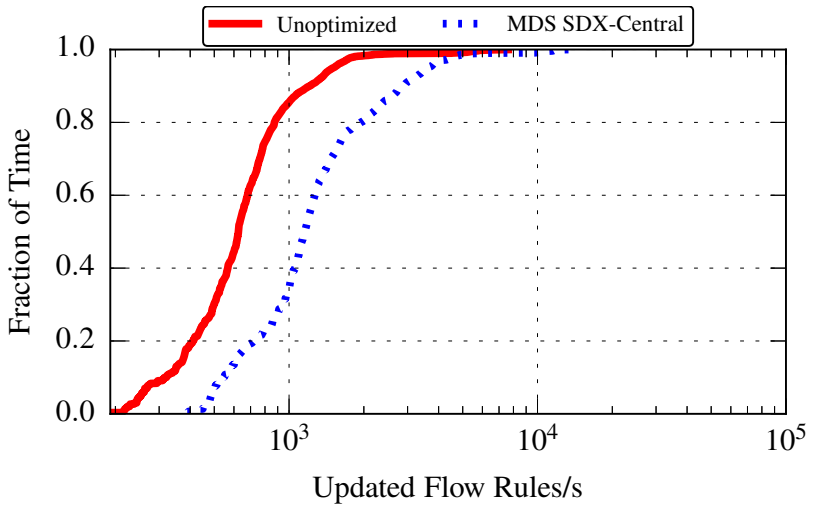


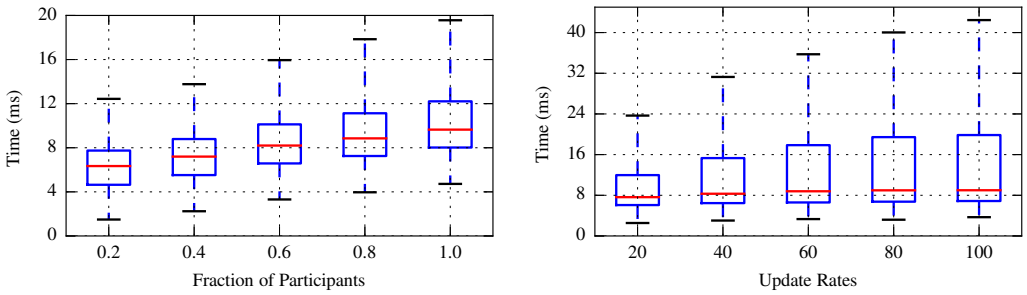

Figure 4.11: Rate at which forwarding table entries are updated.

iSDX-R takes longer than iSDX-D and iSDX-N controllers. For 500 participants, policy compression takes about three seconds.

### 4.7.3 Runtime Performance

After iSDX initializes, we replay a two-hour trace of BGP updates from one of the largest IXPs in the world to evaluate the runtime performance of iSDX compared to other SDX designs. We focus on how iSDX reduces the number of forwarding table updates induced by BGP updates and policy changes, as well as the corresponding increase in gratuitous ARP traffic, which is the cost we pay for increased forwarding table stability.

**Forwarding table updates in response to routing.** Figure 4.11 shows the cumulative distribution of the number of updated forwarding table entries per second the SDX must process for a BGP update stream coming from all 511 participants at the IXP. MDS compression, which is used in iSDX-D and iSDX-N, significantly increases the rate of updates to the forwarding table in comparison to an unoptimized SDX; this result makes sense because any change to forwarding is more likely to trig-




(a) Compute time for increasing forwarding actions.

(b) Compute time for increasing sustained rates of BGP updates.

Figure 4.12: Latency of iSDX-R updates in response to BGP update streams.

ger a change to one of the encoded forwarding table entries. *With iSDX-R, there are never updates to the forwarding table entries in response to BGP updates.*

**Update latency in response to BGP updates.** We aim to understand how quickly iSDX-R can update forwarding information when BGP updates arrive. For iSDX-R, this update time effectively amounts to computing updated virtual next-hop IP and MAC addresses, since iSDX-R never needs to update the IXP fabric forwarding table entries in response to BGP updates. We evaluate update latency with two experiments. First, we vary the fraction of IXP participants to which each IXP participant forwards with SDN policies. For example, if the fraction is 1, each participant has between one and four SDN forwarding policies (at random) for every other SDN participant. Figure 4.12a shows this result; in all cases, the median update latency in response to a BGP update is less than 10 ms, and the 95th percentile in the worst case is less than 20 ms. Even when we perform simultaneous compilation of all 511 participants on just three servers at the IXP, the median update time is only 52 ms, well within practical requirements.

To understand how iSDX-R behaves when it receives larger update bursts, we evaluate the update latency for increasing sizes of BGP update bursts. We vary the number of BGP updates per second from 20 to 100 and send a constant stream of

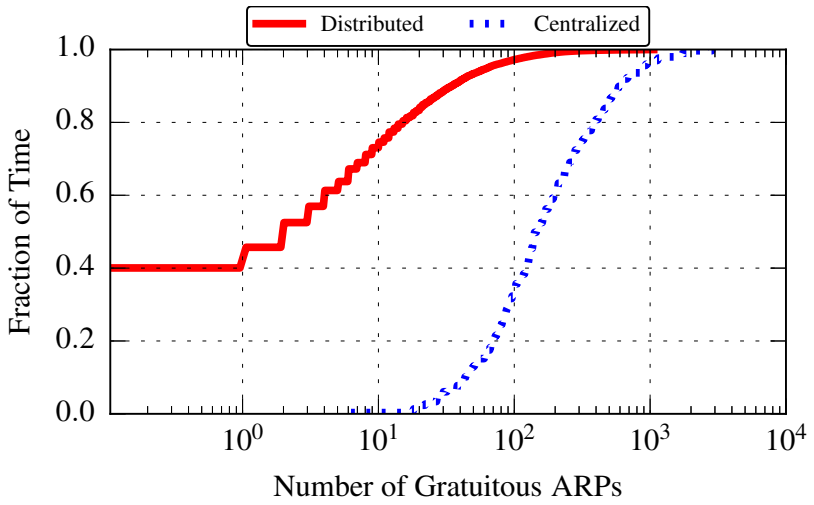



Figure 4.13: Rate at which a participant's border router receives gratuitous ARPs.

updates at this rate for five minutes, tracking the latency that the iSDX requires to process the updates. (Although a table reset would presumably cause a very large update burst, the fastest sustained BGP update rate we observed in the trace was only about 35 BGP updates per second.) Figure 4.12b shows this result. For example, for a rate of 100 BGP updates per second, the median update latency is about 8 ms and the 95th percentile is percentile is about 45 ms.

**Gratuitous ARP overhead.** Recall that SDX relies on gratuitous ARP to update virtual destination MAC addresses when forwarding behavior changes, often in lieu of updating the forwarding table itself. A centralized SDX control plane sends this ARP response to all IXP participants, but a distributed SDX can send this response only to the border router whose route changed. Figure 4.13 shows the distribution of the rate at which a participant's border router receives gratuitous ARP messages from the IXP controller in response to BGP routing changes, for both the centralized design (*i.e.*, centralized MDS) and the distributed one (*i.e.*, iSDX); these rates are independent of which encoding the iSDX uses.

## 4.8 Related Work

**SDX Projects.** Software-defined IXPs have been gaining momentum in the past few years [124, 9, 64], and limited real-world deployments are beginning to emerge. Yet, these existing deployments have focused on either smaller IXPs or on forwarding traffic for a partial routing table. Our original SDX [42] work introduced mechanisms for applying SDN policies to control interdomain traffic flow at an IXP and introduced some simple mechanisms for forwarding table compression; yet its capability for compressing and updating forwarding tables cannot meet either the scale or speed demands of the largest industrial IXP. Google's Cardigan SDX controller has been deployed in a live Internet exchange in New Zealand [106, 10]. Cardigan does not use any of the compression techniques that we use in either SDX or Sonata. As a result, we expect that the size of Cardigan's forwarding tables would be similar to the "unoptimized" results that we present in Section 4.2—orders of magnitude too large for use with hardware switches in large IXPs. Control Exchange Points [61] propose to interconnect multiple SDN IXPs to provide QoS services to the participants and is less concerned with the design of an individual SDN-based IXP.

**Distributed SDN controllers.** HyperFlow [110], Onix [59], and Devolved Controllers [107] implement distributed SDN controllers that maintain eventually consistent network state partitioning computation across multiple controllers such that each operates on less state. Kandoo [44] distributes the control plane for scalability, processing frequent events in highly replicated local control applications and rare events in a central location. Several distributed controllers focus on fault-tolerance [58, 18, 26]. In contrast to these systems, each participant controller in Sonata operates independently and requires no state synchronization. Sonata's partitioning is first and foremost intended to achieve more efficient compression of for-

warding table entries; other benefits, such as parallel computation and fault tolerance, are incidental benefits.

**Techniques for data-plane scalability.** Other work seeks to address the problem of small forwarding tables in hardware. Data-plane scaling involves (1) rule partitioning [127], where data plane rules are partitioned across multiple switches and incoming traffic is steered to load balance across these switches; and (2) caching [99, 57], which stores forwarding table entries for only a small number of flows in the data plane. These techniques are orthogonal to the compression that Sonata uses. Labeling packets for FIB compression has been applied in various contexts, such as MPLS [25], Fabric [19], LISP[33], and Shadow Macs [1]. These techniques all reduce the number of forwarding table entries in certain routers, often by pushing complex policies to the edge of the network. These techniques generally apply in the wide area, and cannot be directly applied to an IXP topology, although some of the techniques are analogous.

# Chapter 5

# Conclusion

## 5.1 Filling the Scalability and Flexibility Gap

The gap between flexibility and scalability exists for network-management systems as there is a mismatch between the required and available compute and storage resources. This gap widens as we try to support the limitless creativity of network operators with the available network resources.

This dissertation shows how we can harness the power of programmable switches to fill this gap. More concretely, it focuses on the design and implementation of two systems, *Sonata* and *SDX*. Sonata is a network-monitoring system that opportunistically uses stream processors and PISA switches to scale the execution of flexible dataflow queries over packet stream for network monitoring. On the other hand, SDX and iSDX, are network-control systems that use both the programmable switch and fixed-function border routers to scale the execution of flexible control programs expressed by IXP participants for flexible wide-area traffic delivery.

| | Sonata | SDX |
|---|---|---|
| **Abstractions** | Dataflow queries over packet fields | Virtual switch abstraction |
| **Algorithms** | Query partitioning and refinement algorithms | Compilation and attribute encoding algorithms |
| **Systems** | • Prototype (9K lines of code) with commodity switch (Barefoot Tofino [112]) and stream processor (Apache Spark [111]).<br>• Used by researchers and developers at AT&T; and students enrolled for the Advanced Networking course at Princeton University [20]. | • Prototype (5K lines of code) with commodity switch (Quanta).<br>• Open-sourced with Open Networking Foundation.<br>• Used by researchers and developers at DE-CIX, IX-BR, IIX, NSA; and students enrolled for the SDN Coursera course [22, 21]. |

Table 5.1: Summary of contributions.

## 5.2 Summary of Contributions

Table 5.1 summarizes the contributions of this dissertation in terms of: (1) the *abstractions* that make it easier for network operators to express flexible programs for network monitoring and control, (2) the *algorithms* that make the best use of limited network resources, and (3) *systems* that glue the high-level abstractions to the low-level algorithms.

**Flexible and Scalable Network Monitoring with Sonata.** Ensuring the security and performance of networks requires continually collecting and analyzing data. Sonata makes it easy to do so, by exposing a familiar, unified query interface to operators and building on advances in both stream processing and programmable switches to implement these queries efficiently.

More concretely, the new query interface lets network operators express flexible monitoring tasks as dataflow queries over packet stream. We developed new query-planning algorithms for computing optimal query plans that make the best use of available data-plane resources to minimize the bandwidth (compute) overhead, *i.e.*, the amount of traffic sent by the switch to the stream processor.

We implemented an extensive and modular Sonata prototype that glues the high-level query interface and the low-level query-planning algorithms together. Our prototype, which is publicly available over Github [116], was only $9,000$ lines of code and compiled to state-of-the-art Barefoot Tofino [112] (programmable switch) and Apache Spark [111] (stream processor). Our evaluation shows that Sonata can support a wide range of telemetry tasks while reducing the workload for the stream processor by as much as seven orders of magnitude compared to existing monitoring systems.

**Flexible and Scalable Network Control with SDX (iSDX).** SDX breaks the logjam on long-standing problems in interdomain routing by enabling fine-grained control over packet handling. SDX supports programs that match and act on multiple header fields and allow networks to have remote control over the traffic. It addresses many of the challenges of an SDN-enabled IXP.

More concretely, the virtual-switch abstraction makes it easier for network operators to express their control programs without worrying about other networks at the IXP. It also ensures isolation, guaranteeing that networks cannot see or control aspects of interdomain routing outside of their purview. The compilation algorithms allow the SDX controller to combine policies, resolving conflicts that arise between participants, and ensuring that forwarding is consistent with BGP route advertisements. The attribute-encoding algorithms help reduce the number of TCAM entries for the programmable switches. These algorithms encode the reachability attribute to the packets before they enter the IXP's switching fabric using the fixed-function border routers of IXP's participants.

We implemented an SDX prototype that glues the virtual switch abstraction with the compilation and attributes encoding algorithms. We released our prototype, which was $5,000$ lines of code, as an open-source project with Open Networking Foundation (ONF), and is publicly available over Github [48]. We also released tu-

torials and instructions that have helped catalyze early adoption. More specifically, the Endeavour platform [109], developed by a consortium of researchers and network operators in Europe [29], uses our prototype as the de-facto solution for developing new SDX-based applications. Our evaluation shows that iSDX reduces both forwarding table size and the time to compute these entries by several orders of magnitude. Using BGP routing updates from a route server at one of the world's largest IXPs, we showed that iSDX can support industry-scale operation.

## 5.3 Moving Forward from Lessons Learned

Given the exponential increase in the number of Internet-connected devices and applications, we expect the complexity of network management to keep growing. Network operators have responded to fill the gap between flexibility and scalability by beefing up the available network resources. Unfortunately, given the trends, this approach will get prohibitively expensive over time, and network operators will fall back to less flexible network-management systems, compromising the security and performance of their networks.

In this dissertation, we demonstrated that it is possible to fill the flexibility and scalability gap for network monitoring and control with limited available network resources. We learned how to design modular systems that are capable of pooling resources from a heterogeneous set of network devices, and how deploying these systems at strategic locations (e.g., IXPs) enables incremental deployability. However, we argue that so far we have only scratched the surface. Below, we discuss how we can leverage the lessons learned from this dissertation to move forward and build network-management systems that are not only more flexible and scalable but also intelligent.

### 5.3.1 Developing Intelligent Network-Monitoring System

Intelligent network-monitoring systems should be capable of detecting network events by themselves. Currently, network operators either utilize existing learning models or train new ones to infer various network events. Network-monitoring systems like Sonata determine the execution plans for queries representing these learning models. Currently, the learning algorithms, which are designed to run over general-purpose CPUs, focus only on training learning models that maximize detection accuracy. At the same time, network-monitoring systems' query-planning algorithms solely concentrate on minimizing the execution cost for queries that represent these learning models. This disconnect between the learning and query-planning algorithms often results in very accurate yet prohibitively expensive learning models that are not deployable in production networks.

To design an intelligent network-monitoring system, we need to co-design the learning and query-planning algorithms. This new algorithm determines the learning model as well as the query plan by modifying the optimization problem that most state-of-the-art learning algorithms solve by using the query execution cost as a metric to be minimized, and by adding various resource constraints (e.g., limited memory in the data plane). This new algorithm can replace the query-planning algorithms used by the query-planning module of the existing network-monitoring systems. Here, modularization ensures that the system can evolve with time as we develop more efficient algorithms (computing both the learning models and query plans) over time.

### 5.3.2 Expanding Networking-Monitoring Footprint

Sonata only focuses on detecting network events observable at a single location, e.g., border router for large ISPs, or an IXP switch. Though such centralized deployments provide broader visibility with limited overhead, they miss detecting several network-wide events, such as port scanning, or end-to-end performance for specific web ap-

plications (e.g., Netflix, CNN, etc.) in a region. These insights, are either available at multiple vantage points within a network or the edge of the Internet (e.g., home routers, web browsers, etc.), outside the administrative domain of network operators. To expand the network-monitoring footprint network-monitoring systems should be capable of detecting network-wide events by combining insights extracted from multiple vantage points. For the cases, where monitoring systems need to extract insights from locations outside the network operator's administrative domain, they need to balance fundamental friction between data producers and consumers. Here, the data consumers (*i.e.*, network-monitoring systems) are trying to get as much information as possible, and the data producers (e.g., owners of remote home routers) are trying to preserve their privacy. Thus, the design of such a decentralized network-monitoring system needs to execute queries in a privacy-preserving manner.

To design a privacy-preserving distributed network-monitoring system, we need to extend the query interface to support additional meta fields, such as `path`, to the packet tuple, design a new coordination algorithm that synchronizes information collected at different locations in a privacy-preserving manner, and design new query-planning algorithms that make the best use of limited network-wide storage and compute resources.

### 5.3.3 Closing the Network Monitoring and Control Loop

So far, this dissertation focusses on building flexible and scalable systems for network monitoring (Sonata) and network control (SDX). Though the output of network monitoring drives network control, both these systems still require human operators first to express the queries to infer various network events and then express control programs for reactive actions, *i.e.*, these existing systems require humans in the loop. This approach works for relatively simpler monitoring tasks. However, as the complexity and frequency of network events increases, the overhead of manually reacting to

network events will get impractical. In the future, we envision network-management systems that can take the human operators out of this control loop, determining which queries to execute, when to run them, and how to react to various network events, all by themselves—closing the loop between network monitoring and control.

To close the loop between network monitoring and control, we need to develop new algorithms that can learn how to model the relationship between the observed network events and reactive control actions. To develop such learning algorithms, we require labeled training data. We can collect this data from the information logged by various network devices. For example, we can use logs of network-monitoring queries ran by the network operators to detect a network event and the related control actions to address the problem.

## 5.4 Concluding Remarks

This dissertation developed two flexible and scalable systems for network monitoring and control, respectively. The network-monitoring system, *Sonata*, collects and analyzes raw network data at scale to infer various network events, such as DDoS attacks or link failures, in real time. The network-control system, *SDX*, applies fine-grained reactive control actions without disrupting the routing protocols in today's Internet. The design of these systems focused on: (i) the *abstractions* that allows network operators to express flexible programs for monitoring and control; (ii) the *algorithms* that make the best use of limited compute and storage resources in the network; and (iii) the *systems* that form the glue between the high-level abstractions and the low-level algorithms.

As the complexity of network management keeps increasing with the explosion in the number of Internet-connected devices and applications, we expect that network management needs to be more sophisticated and less reliant on human operators. We

believe that one should be able to apply the lessons learned from this dissertation to develop network management systems that are more intelligent, can extract insights from multiple vantage points, and can close the loop between monitoring and control without any human intervention. We argue that by building the two systems in a modular fashion, they facilitate embedding intelligence over time, and in that sense, they can serve as building blocks for next-generation network-management systems.

# Bibliography


[1] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow MACs: Scalable Label-switching for Commodity Ethernet. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2014.

[2] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. Anatomy of a Large European IXP. In *ACM SIGCOMM*, 2012.

[3] AMS Internet Exchange. `https://www.ams-ix.net/ams-ix-route-servers/`, 2013.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, and Michalis Kallitsis. Understanding the Mirai botnet. In *USENIX Security Symposium*, 2017.

[5] Apache Cassandra. `http://cassandra.apache.org/`.

[6] Apache Thrift API. `https://thrift.apache.org/`.

[7] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, 2016.

[8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, and Ali Ghodsi. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD/PODS International Conference on Management of Data*, 2015.

[9] AtlanticWave-SDX. `https://itnews.fiu.edu/wp-content/uploads/sites/8/2015/04/AtlanticWaveSDX-Press-Release_FinalDraft.pdf`.

[10] Josh Bailey, Dean Pemberton, Andy Linton, Cristel Pelsser, and Randy Bush. Enforcing rpki-based routing policy on the data plane at an internet exchange. ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking, 2014.

[11] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *Network and Distributed System Security Symposium*, 2011.

[12] BIRD. `http://bird.network.cz/`.

[13] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium*, 2012.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[16] Victor Boteanu, Hanieh Bagheri, and Martin Pels. Minimizing ARP traffic in the AMS-IX switching platform using OpenFlow. 2013.

[17] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *USENIX NSDI*, 2005.

[18] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *IEEE INFOCOM*, 2015.

[19] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2012.

[20] COS 561: Advanced Computer Networks, Princeton University, Fall 2017. `http://www.cs.princeton.edu/courses/archive/fall17/cos561/`.

[21] Coursera SDN Course. `class.coursera.org/sdn-002/`.

[22] Coursera SDN Course, 2015. `https://www.coursera.org/course/sdn1`.

[23] Chuck Cranor, Theodore Johnson, Oliver Spatschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD/PODS International Conference on Management of Data*, 2003.

[24] The CAIDA UCSD Anonymized Internet Traces 2016-09. `http://www.caida.org/data/passive/passive_2016_dataset.xml`.

[25] Bruce S. Davie and Yakov Rekhter. *MPLS: technology and applications*. San Francisco, 2000.

[26] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an Elastic Distributed SDN Controller. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2013.

[27] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *ACM Internet Measurement Conference*, 2015.

[28] Endance packet capture cards. `https://www.endace.com/endace-dag-high-speed-packet-capture-cards.html`.

[29] ENDEAVOUR: Towards a flexible software-defined network ecosystem. `https://www.h2020-endeavour.eu/`.

[30] Cristian Estan, Stefan Savage, and George Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *ACM SIGCOMM*, 2003.

[31] ExaBGP. `https://github.com/Exa-Networks/exabgp`.

[32] Jinliang Fan, Jun Xu, Mostafa H. Ammar, and Sue B. Moon. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. *Computer Networks*, 2004.

[33] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. The Locator/ID Separation Protocol (LISP). Internet Requests for Comments, January 2013. `http://www.rfc-editor.org/rfc/rfc6830.txt`.

[34] Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, Ron Hutchins, Dave Levin, and Josh Bailey. SDX: A Software Defined Internet Exchange. 2013.

[35] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker. Languages for software-defined networks. *IEEE Communications Magazine*, 2013.

[36] Thomer M Gil and Massimiliano Poletto. MULTOPS: A data-structure for bandwidth attack detection. In *USENIX Security Symposium*, 2001.

[37] Ramesh Govindan, Cengiz Alaettinoglu, Kannan Varadhan, and Deborah Estrin. Route servers for inter-domain routing. In *Computer Networks and ISDN Systems*, 1998.

[38] Arpit Gupta, Rudiger Birkner, Marco Canini, Nick Feamster, Chris MacStoker, and Walter Willinger. Network Monitoring as a Streaming Analytics Problem. In *ACM SIGCOMM Workshop on Hot Topics in Networking*, 2016.

[39] Arpit Gupta, Rob Harrison, Rüdiger Birkner, Ankita Pawar, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Network Telemetry. *Arxiv*, 2017.

[40] Arpit Gupta, Robert MacDavid, Rudiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *USENIX NSDI*, 2016.

[41] Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Hyojoon Kim, Russ Clark, Nick Feamster, Jennifer Rexford, and Scott Shenker. SDX: A Software Defined Internet Exchange. *Technical Report*, 2013.

[42] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Patrick Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A Software Defined Internet Exchange. In *SIGCOMM*, 2014.

[43] Gurobi Solver. `http://www.gurobi.com/`.

[44] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2012.

[45] Brandon Heller, Nikhil Handigol, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments using Container Based Emulation. In *CoNEXT*, 2012.

[46] Mukesh Hira and L. J. Wobker. Improving Network Monitoring and Management with Programmable Data Planes. Blog posting, `http://p4.org/p4/inband-network-telemetry/`, September 2015.

[47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, 2013.

[48] iSDX Gihub Repo. `https://github.com/sdn-ixp/iSDX`.

[49] iSDX HW Test Instructions. `https://github.com/sdn-ixp/iSDX/tree/master/examples/test-ms/ofdpa`.

[50] Martin Izzard. The Programmable Switch Chip Consigns Legacy Fixed-Function Chips to the History Books. `https://goo.gl/JKWnQc`, September 2016.

[51] Sushant Jain, Alok Kumar, Subhashree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, J. Zolla, Urs Hölzle, S. Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, 2013.

[52] E. Jasinska, N. Hilliard, R. Raszuk, and N. Bakker. Internet Exchange Route Server. *IETF*, 2013.

[53] Mobin Javed and Vern Paxson. Detecting stealthy, distributed SSH bruteforcing. In *ACM SIGSAC Conference on Computer & Communications Security*, pages 85–96, 2013.

[54] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *USENIX NSDI*, 2015.

[55] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2011.

[56] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, pages 211–225. IEEE, 2004.

[57] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *ACM Symposium on SDN Research*, 2016.

[58] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *ACM Symposium on SDN Research*, 2015.

[59] Teemu Koponen, 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. 2010.

[60] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager. Outsourcing the routing control logic: Better Internet routing based on SDN principles. In *ACM SIGCOMM Workshop on Hot Topics in Networking*, 2012.

[61] Vasileios Kotronis, Xenofontas Dimitropoulos, Rowan Klöti, Bernhard Ager, Panagiotis Georgopoulos, and Stefan Schmid. Control Exchange Points: Providing QoS-enabled End-to-End Services via SDN-based Inter-domain Routing Orchestration. 2014.

[62] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[63] Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. Routing as a Service. Technical Report UCB/CSD-04-1327, UC Berkeley, 2004.

[64] LightReading. Pica8 Powers French TOUIX SDN-Driven Internet Exchange, June 2015. `http://ubm.io/1Vc0SLE`.

[65] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[66] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny Aggregation Service for Ad-hoc Sensor Networks. In *USENIX OSDI*, 2002.

[67] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transaction on Database System*, 2005.

[68] Joe Mambretti. Software-defined network exchanges (SDXs) and software-defined infrastructure (SDI), 2014. Workshop on Prototyping and Deploying Experimental Software Defined Exchanges.

[69] mongoDB. `https://www.mongodb.org/`.

[70] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *USENIX NSDI*, 2013.

[71] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Dream: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, 2015.

[72] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *CoNEXT*, 2015.

[73] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[74] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 1990.

[75] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. Compiling path queries. In *USENIX NSDI*, 2016.

[76] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

[77] Noviflow. `http://noviflow.com/`.

[78] NoviSwitch 1132. `http://noviflow.com/wp-content/uploads/2014/09/NoviSwitch-1132-Datasheet.pdf`.

[79] Openflow 1.3 specifications. `http://bit.ly/1eyrkxY`.

[80] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: Analysing the rise of IoT compromises. In *USENIX Workshop on Offensive Technology*, 2015.

[81] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. Track Join: Distributed joins with minimal network traffic. In *ACM SIGMOD/PODS International Conference on Management of Data*, 2014.

[82] Quagga. `http://www.nongnu.org/quagga/`.

[83] QuantaMesh BMS T3048-LY2. `http://www.qct.io/Product/Networking/Bare-Metal-Switch/QuantaMesh-BMS-T3048-LY2-p55c77c75c159`.

[84] DNS Amplification Attacks Alert, March 2013. `https://www.us-cert.gov/ncas/alerts/TA13-088A`.

[85] An update on the Memcached/Redis benchmark. `http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html`.

[86] Apache Flink. `http://flink.apache.org/`.

[87] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`.

[88] Deepfield Defender. `http://deepfield.com/products/deepfield-defender/`.

[89] Jitin Thomas, Petr Lapukhov Tracking Packets' Paths and Latency via INT. `https://schd.ws/hosted_files/2016p4workshop/1d/FB%20BFN%2C%20INT-PLT_Talk_May_2016-v1.1-1.pdf`.

[90] OpenSOC. `http://opensoc.github.io/`.

[91] OpenSOC Scalability. `https://goo.gl/CX2jWr`.

[92] The CAIDA Anonymized Internet Traces 2016 Dataset. `https://www.caida.org/data/passive/passive_2016_dataset.xml`.

[93] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. BGP routing stability of popular destinations. In *Internet Measurment Workshop*, pages 197–202. ACM, 2002.

[94] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at Peerings: On the Role of IXP Route Servers. In *Internet Measurement Conference*, 2014.

[95] RIPE. RIS Raw Data, 2015. `https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data`.

[96] RIPE Routing Information Service (RIS). http://www.ripe.net/ris.

[97] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2012.

[98] Ryu SDN Framework. http://osrg.github.io/ryu/.

[99] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's Law for Traffic Offloading. *ACM SIGCOMM Computer Communication Review*, 2012.

[100] SDX Controller. https://github.com/sdn-ixp/sdx-platform.

[101] Big Monitoring Fabric. http://www.slideshare.net/bigswitchnetworks/big-monitoring-fabric-58389045.

[102] Slowloris HTTP DoS. https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris, June 2009.

[103] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for netkat. In *ICFP*, 2015.

[104] SQLite. https://www.sqlite.org/.

[105] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Symposium on Principles of Database Systems*, 2005.

[106] J. Stringer, D. Pemberton, Qiang Fu, C. Lorier, R. Nelson, J. Bailey, C.N.A. Correa, and C. Esteve Rothenberg. Cardigan: SDN distributed routing fabric going live at an Internet exchange. In *Symposium on Computers and Communication*, 2014.

[107] Adrian S-W Tam, Kang Xi, and H. Jonathan Chao. Use of devolved controllers in data center networks. *arXiv preprint arXiv:1103.5586*, 2011.

[108] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with PathDump. In *USENIX OSDI*, 2016.

[109] The ENDEAVOUR platform. https://www.h2020-endeavour.eu/.

[110] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Workshop on Internet Network Management*, 2010.

[111] Apache Spark. http://spark.apache.org/.

[112] Barefoot's Tofino. https://www.barefootnetworks.com/technology/.

[113] Kentik. https://www.kentik.com.

[114] P4 software switch. https://github.com/p4lang/behavioral-model.

[115] Scapy: Python-based interactive packet manipulation program. https://github.com/secdev/scapy/.

[116] Sonata Github. https://github.com/Sonata-Princeton/SONATA-DEV.

[117] Sonata Queries. https://github.com/sonata-queries/sonata-queries.

[118] Velocidata. http://velocidata.com/.

[119] Vytautas Valancius, Nick Feamster, Jennifer Rexford, and Akihiro Nakao. Wide-area route control for distributed services. 2010.

[120] Bapi Vinnakota. P4 with the Netronome Server Networking Platform. https://goo.gl/PKQtC7, May 2016.

[121] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *ACM SIGCOMM Workshop on Hot Topics in Software-defined Networking*, 2012.

[122] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *HotICE Workshop*, March 2011.

[123] Scott Whyte. Project CARDIGAN An SDN Controlled Exchange Fabric. https://www.nanog.org/meetings/nanog57/presentations/Wednesday/wed.lightning3.whyte.sdn.controlled.exchange.fabric.pdf, 2012.

[124] Workshop on Prototyping and Deploying Experimental Software Defined Exchanges. https://www.nitrd.gov/nitrdgroups/images/4/4d/SDX_Workshop_Proceedings.pdf.

[125] Q. Wu, J. Strassner, A. Farrel, and L. Zhang. Network telemetry and big data analysis. *Network Working Group Internet-Draft*, 2016 (Expired).

[126] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*, 2013.

[127] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable Flow-based Networking with DIFANE. *ACM SIGCOMM*, 2010.

[128] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. Progme: Towards programmable network measurement. In *ACM SIGCOMM*, 2007.

[129] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative Network Monitoring with NetQRE. In *ACM SIGCOMM*, 2017.

[130] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

[131] Beichuan Zhang, Vamsi Kambhampati, Mohit Lad, Daniel Massey, and Lixia Zhang. Identifying BGP routing table transfer. In *SIGCOMM MineNet Workshop*, August 2005.

[132] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, and Ben Y. Zhao. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.