SDX: A Software Defined Internet Exchange

Arpit Gupta[†], Laurent Vanbever[⋆], Muhammad Shahbaz[†], Sean P. Donovan[†], Brandon Schlinker[‡]
Nick Feamster[†], Jennifer Rexford[⋆], Scott Shenker[◇], Russ Clark[†], Ethan Katz-Bassett[‡]
[†]Georgia Tech [⋆]Princeton University [◇]UC Berkeley [‡]Univ. of Southern California

Abstract

BGP severely constrains how networks can deliver traffic over the Internet. Today's networks can only forward traffic based on the destination IP prefix, by selecting among routes offered by their immediate neighbors. We believe Software Defined Networking (SDN) could revolutionize wide-area traffic delivery, by offering direct control over packet-processing rules that match on multiple header fields and perform a variety of actions. Internet exchange points (IXPs) are a compelling place to start, given their central role in interconnecting many networks and their growing importance in bringing popular content closer to end users.

To realize a Software Defined IXP (an "SDX"), we must create compelling applications, such as "application-specific peering"—where two networks peer only for (say) streaming video traffic. We also need new programming abstractions that allow participating networks to create and run these applications and a runtime that both behaves correctly when interacting with BGP and ensures that applications do not interfere with each other. Finally, we must ensure that the system scales, both in rule-table size and computational overhead. In this paper, we tackle these challenges and demonstrate the flexibility and scalability of our solutions through controlled and in-the-wild experiments. Our experiments demonstrate that our SDX implementation can implement representative policies for hundreds of participants who advertise full routing tables while achieving sub-second convergence in response to configuration changes and routing updates.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks] *Network Architecture and Design*: Network Communications

General Terms: Algorithms, Design, Experimentation

Keywords: software defined networking (SDN); Internet exchange point (IXP); BGP

1 Introduction

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. Internet routing's problems result from three characteristics of the Border Gateway Protocol (BGP), the Internet's interdomain routing protocol:

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2619239.2626300

- **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 paper, we develop 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 believe 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 the tier-1 ISPs [1, 4]. 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 four motivating applications: application-specific peering, inbound traffic engineering, server load balancing, and traffic redirection through middleboxes (Section 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).
- **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 4).
- **Realistic deployment.** We have built a prototype and created two example applications (Section 5). Experiments demonstrate that our prototype scales (in terms of rule-table size and CPU time) to many participants, policies, and prefixes (Section 6).

We conclude with a discussion of related work (Section 7) and future possibilities (Section 8).

2 Wide-Area Traffic Delivery

We present four 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.

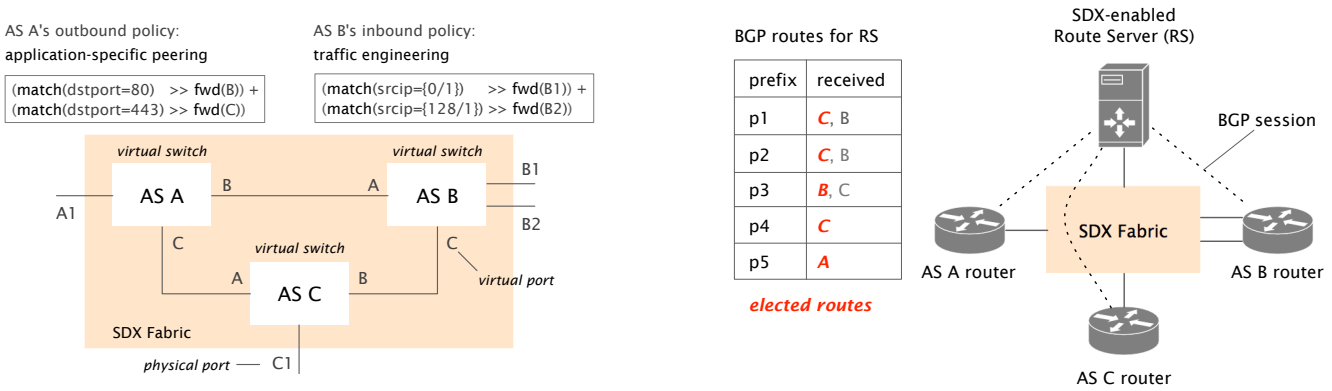
Redirection through middleboxes. Networks increasingly rely on middleboxes to perform a wide range of functions (*e.g.*, firewalls, network address translators, load balancers). Enterprise networks at the edge of the Internet typically place middleboxes at key junctions, such as the boundary between 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 Programming Abstractions

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.

3.1 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 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,



(a) *Virtual switch abstraction.*

(b) *Integration with interdomain routes.*

Figure 1: *SDX programming abstractions.*

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 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 [12] 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 2.

Application-specific peering. In Figure 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 [12].

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 [21]. 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.

3.2 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 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 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 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 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 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.

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 4.1). We then describe how we made that process efficient. We consider *data-plane* efficiency (Section 4.2), to minimize the number of rules in the switches, and *control-plane* efficiency (Section 4.3), to minimize the computation time under realistic workloads.

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 1a) or a virtual connection to another participant's virtual switch (*e.g.*, port B on AS A's virtual switch in Figure 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 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 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 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 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 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.

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 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 [13].

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 1, AS A has the same forwarding behavior for p_1 and p_2 (*i.e.*, send Web traffic via AS B, and send

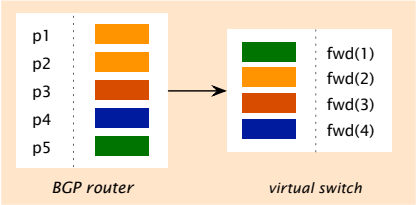


Figure 2: *Multi-stage FIB for each participant, where the first stage corresponds to the participant's border router and the second stage corresponds to the participant's virtual switch at the SDX.*

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 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 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 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 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.

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.

4.3.1 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, 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 [1]. 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.

	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 1: *IXP datasets. We use BGP update traces from RIPE collectors [16] 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 [23].*

4.3.2 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 1 summarizes the data that we used for this analysis.

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 [15]. 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

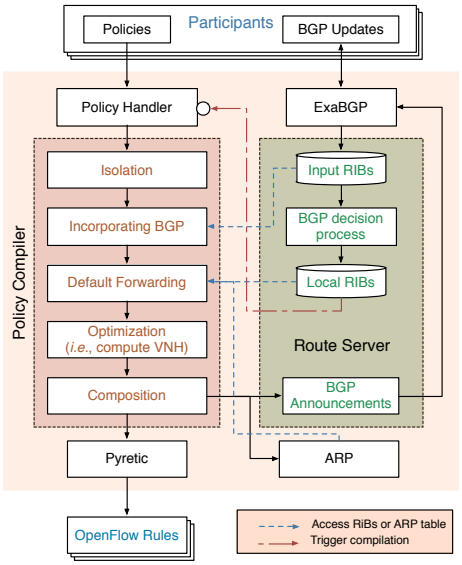


Figure 3: *The SDX controller implementation, which has two pipelines: a policy compiler and a route server.*

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 reoptimization. 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 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.

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).

5.1 Implementation

Figure 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.

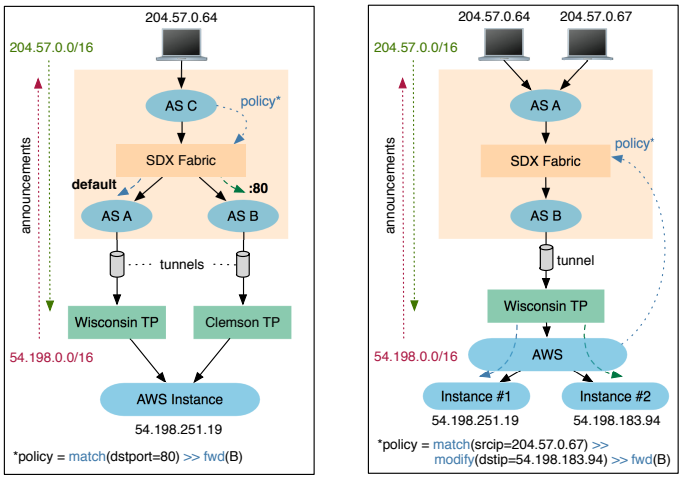
SDX policy compiler. The policy compiler is a Pyretic process that compiles participant policies to forwarding rules. Based on the virtual SDX abstraction from the SDX configuration (*i.e.*, the static configuration of which ASes are connected to each other at layer two), the policy compiler isolates 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 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 [5], an existing route server that is implemented in Python. As in other traditional route servers [2, 14], 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 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 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.

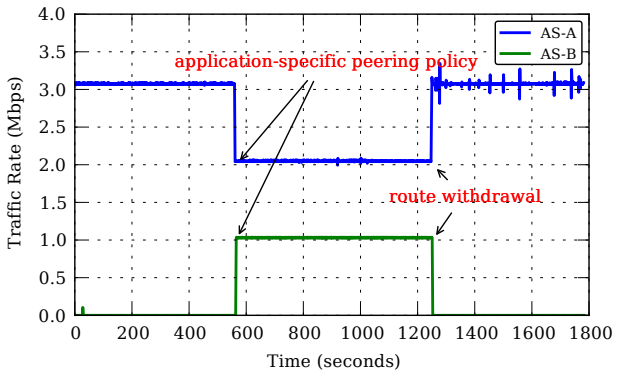
5.2 Deployment

We have developed a prototype of the SDX [18] and a version that can be deployed using virtual containers in Mininet [7]. Figure 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 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 [19]. The client generates three 1 Mbps UDP flows, varying the

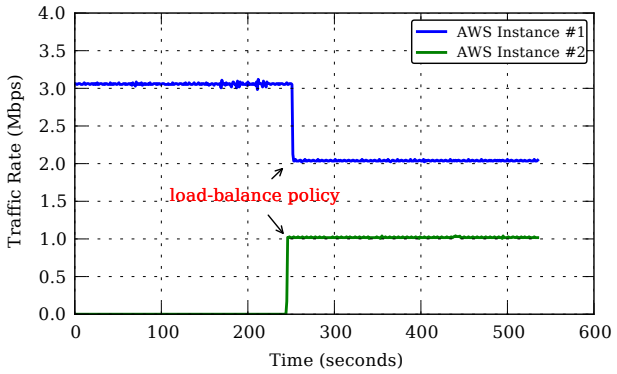


(a) *Application-Specific Peering.* **(b)** *Wide-Area Load Balance.*

Figure 4: *Setup for deployment experiments.*



(a) *Application-specific peering.*



(b) *Wide-area load balance.*

Figure 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.*

source and destination IP addresses and ports as required for the demonstrations below.

Application-specific peering. Figure 4a shows an SDX setup where we test the application-specific peering use-case described in Section 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 **??** 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 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 **??** 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.

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.

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.

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 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.

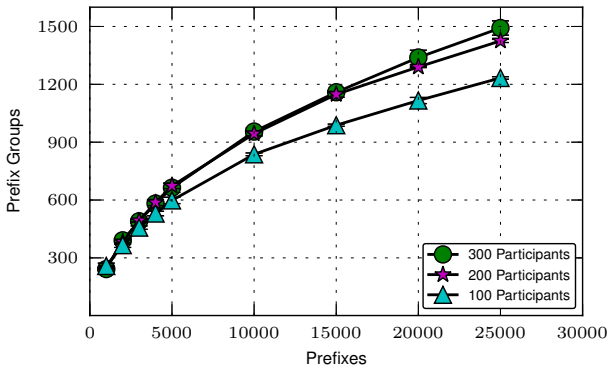


Figure 6: *Number of prefix groups as a function of the number of prefixes, for different numbers of participants.*

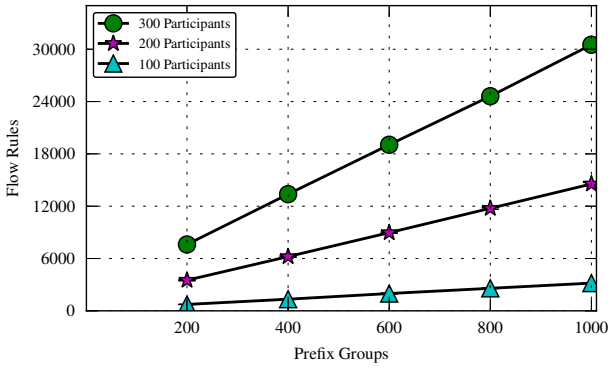


Figure 7: *The number of forwarding rules as a function of the number of prefix groups for different number of participants.*

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 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 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 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 6). We run the experiment as described above, selecting participant ASes according to common

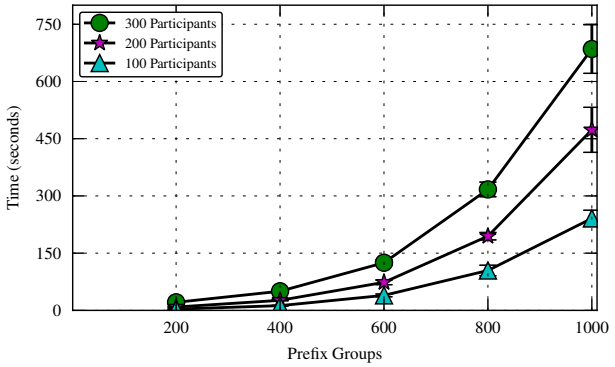


Figure 8: *Compilation time as a function of the number of prefix groups, for different numbers of participants.*

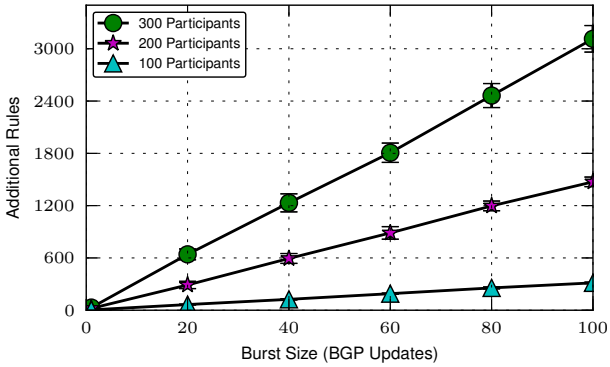


Figure 9: *Number of additional forwarding rules.*

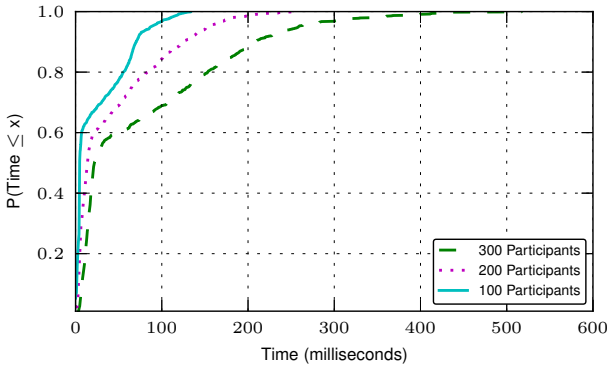


Figure 10: *Time to process a single BGP update for various participants.*

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.

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

As discussed in Section 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 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 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 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 10, re-computing the tables takes less than 100 milliseconds most of the time.

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 [22], which shares our broad goal of using SDN to enable innovation at IXPs. Cardigan runs a route server based on RouteFlow [17] 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 [8, 9]. 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 [3]. 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 [10, 11], 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 [6, 12, 20]. 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 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.

8 Conclusion

SDX can break the logjam on long-standing problems in interdomain routing by enabling entirely new policies with fine-grained control over packet handling. The SDX supports policies that match and act on multiple header fields, and allow ASes to have remote control over the traffic. The SDX addresses many of the challenges of an SDN-enabled IXP. The virtual switch abstraction ensures that ASes cannot see or control aspects of interdomain routing outside of their purview. Policy compilation allows the SDX controller to combine policies, resolving conflicts that arise between participants. The SDX policy compilation algorithm ensures that forwarding is consistent with BGP route advertisements; various optimizations ensure that SDX policies can be efficiently compiled to flow rules; and that these rules can be updated quickly when policies or BGP routes change. We have run experiments with the SDX in both controlled settings and in the wide area, and we have released a preliminary version of the SDX controller [18]. In ongoing work, we are working with a large regional IXP to deploy OpenFlow switches and our SDX controller for an initial deployment of interdomain routing applications beyond BGP.

As demand grows for more flexible data-plane functionality, we believe that BGP should also evolve to support richer patterns (beyond destination prefix) and actions (beyond selecting a single next-hop). We also envision that participant ASes might eventually write policies not only to control how traffic flows between ASes, but also to control how traffic flows through middleboxes (and other cloud-hosted services) along the path between source and destination, thereby enabling "service chaining" through middleboxes.

Acknowledgments

We thank our shepherd Walter Willinger, Hyojoon Kim, João Luís Sobrinho, Jennifer Gossels, Darrell Newcomb, Michael Schapira, and the anonymous reviewers for comments. This research was supported by NSF awards CNS-1040705, CNS-1040838, CNS-1162112, CNS-1261462, and CNS-1261357. We thank Internet2 for their support and recognition with an Internet Innovation Award, the organizations that host Transit Portal nodes, and the GENI Project Office for supporting our ongoing deployment efforts.

References

[1] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a large European IXP. In *Proc. ACM SIGCOMM*, 2012.

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

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. USENIX NSDI*, 2005.

[4] Euro-IX Public Resources. `https://www.euro-ix.net/resources`.

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

[6] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, A. Story, and D. Walker. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, 2013.

[7] B. Heller, N. Handigol, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container based emulation. In *Proc. ACM CoNEXT*, December 2012.

[8] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. ACM SIGCOMM*, 2013.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. ACM SIGCOMM*, 2013.

[10] V. Kotronis, X. Dimitropoulos, and B. Ager. Outsourcing the routing control logic: Better Internet routing based on SDN principles. In *Proc. HotNets Workshop*, pages 55–60, 2012.

[11] K. Lakshminarayanan, I. Stoica, and S. Shenker. Routing as a service. Technical Report UCB/CSD-04-1327, UC Berkeley, 2004.

[12] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *Proc. USENIX NSDI*, 2013.

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

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

[15] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *Proc. Internet Measurment Workshop*, pages 197–202. ACM, 2002.

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

[17] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Proc. HotSDN Workshop*, pages 13–18. ACM, 2012.

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

[19] V. Valancius, N. Feamster, J. Rexford, and A. Nakao. Wide-area route control for distributed services. In *Proc. USENIX Annual Technical Conference*, 2010.

[20] A. Voellmy, H. Kim, and N. Feamster. Procera: A language for high-level reactive network control. In *Proc. HotSDN Workshop*, 2012.

[21] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Proc. HotICE Workshop*, March 2011.

[22] S. 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.

[23] B. Zhang, V. Kambhampati, M. Lad, D. Massey, and L. Zhang. Identifying BGP routing table transfer. In *Proc. SIGCOMM MineNet Workshop*, August 2005.