DeSI: A Decentralized Software-Defined Network Architecture for Internet Exchange Points

Habib Mostafaei, Davinder Kumar, Gabriele Lospoto, Marco Chiesa, and Giuseppe Di Battista

***Abstract*—Applications of Software-Defined Networking (SDN) to the Internet Routing hold great promises for supporting the ever-growing performance requirements of Internet applications. The inherent centralization of these SDN approaches on the Internet routing comes with the following concerns: 1) *privacy*, the operators are reluctant to share private routing information, 2) *separation of responsibilities*, the Internet eXchange Point (IXP) running the centralized controller is involved in the routing and forwarding at too many levels, and 3) *scalability*, the growing number of IP prefixes routed on the Internet (i.e., hundreds of thousands) pose extremely high requirements at both the control- and data-planes, e.g., several minutes for policy compilations and a large number of forwarding rules, in SDN. In this paper, we propose DeSI to apply SDN at IXPs by considering the above concerns. We break this centralization by devising an SDN-enabled IXP architecture in which each member connects to an SDN-enabled IXP through its SDN controller and SDN switches, thus tackling privacy, scalability, and separation of concerns issues. To spur adoption, we introduce an expressive, yet simple, language to configure the routing policies of the members. Our evaluation shows that DeSI needs *n* times fewer forwarding table entries for an IXP in which *n* is the number of IXP members. DeSI also gives the possibility of slowly migrating to the SDN-enabled IXPs.**

***Index Terms*—Software-defined networking, IP networks, Internet eXchange Point (IXP).**

I. Introduction

MODERN-DAY Internet applications pose ever-growing performance requirements on the Internet. Such services require heterogeneous support for performance from the underlying network, including high bandwidth [1] and low-latency [2]. Yet, the underlying network protocol used to determine the Internet paths through which domains send Internet traffic, i.e., the *Border Gateway Protocol* (BGP), is alarmingly oblivious to such performance metrics, ultimately hindering performance. Unfortunately, modifying BGP "overnight" has proven to be an elusive goal because of the need to achieve some sort of wide consensus among independent network entities. Researchers and operators have therefore concentrated efforts to improve the status quo at the emerging crossroads of Internet traffic, i.e., *Internet eXchange Points* (IXPs), where hundreds of organizations connect to exchange traffic at a reduced cost.

IXPs have traditionally acted as mere layer-2 interconnects that transit packets among BGP-speaking networks. The *Software-Defined-eXchange* (SDX) [3] is a new IXP architecture that brings the high programmability of Software-Defined Networking (SDN) [4] to the IXP ecosystem. Both IXP operators and IXP members program the IXP fabric through a well-defined interface (e.g., OpenFlow [5]) to implement their routing policies. The potential impact of SDXes is huge: a recent work [6] showed the high benefits of improved Traffic-Engineering, security, traffic monitoring, network management, and more. Yet, the most notable SDX architecture, i.e., iSDX [7], comes with a variety of concerns. First, several iSDX architectures collect all the members' routing policies in a central controller owned by a third entity. Typically, the IXP members do not want to disclose these routing policies to the IXP. These policies dictate how packets should be routed at the IXP and therefore reveal potentially business information that is deemed confidential [8]. Second, SDXes solutions that install the forwarding policies of different IXP members on the same physical device (e.g., iSDX) may exacerbate any dispute regarding the separation of responsibility in case of failures in delivering traffic. In fact, traditional Layer-2 IXPs separates the responsibility of IXPs, i.e., transporting traffic between two statically configured MAC addresses, from selecting the routes through which sending traffic, which is left to the operators and does not involve any computation on a third-party entity (e.g., the SDX controller). Third, solutions that install forwarding state in the IXP fabric tend to scale poorly in the number of members and configured policies due to the current hardware constraints [9].

We argue that an SDX architecture must satisfy the following requirements. First, *privacy*, i.e., the routing policies of the IXP members should not be disclosed to any unintended third party, including the IXP itself. Second, *separation of responsibility*, i.e., identifying who is responsible for what. Third, *forwarding state scalability*, i.e., the IXP fabric should scale to the number of members, minimizing the amount of state stored in the IXP fabric.

Manuscript received November 8, 2020; revised March 18, 2021; accepted May 16, 2021. Date of publication May 21, 2021; date of current version September 16, 2021. This work was partially funded by the German Ministry for Education and Research as BIFOLD–Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025 A and ref. 01IS18037 A). This work has also been partially supported by KTH Digital Futures. Recommended for acceptance by Dr. Di Wu. *(Corresponding author: Habib Mostafaei.)*

Habib Mostafaei is with Technische Universität Berlin, 10587 Berlin, Germany and also with Roma Tre University, 00146 Roma, Lazio, Italy (e-mail: habib@inet.tu-berlin.de).

Davinder Kumar, Gabriele Lospoto, and Giuseppe Di Battista are with Roma Tre University, 00146 Roma, Lazio, Italy (e-mail: dav.kumar2293@gmail.com; lospoto@dia.uniroma3.it; gdb@dia.uniroma3.it).

Marco Chiesa is with the KTH Royal Institute of Technology, 114 28 Stockholm, Sweden (e-mail: mchiesa@kth.se).

Digital Object Identifier 10.1109/TNSE.2021.3082575

In this paper, we present our envisioned architecture for SDXes, called DeSI, that satisfies all of the above requirements. We argue that IXPs should not be involved in the route computation among members. Instead, in DeSI, IXP members connect to the IXP with their SDN-enabled equipment, including an SDN switch to be connected to the SDX fabric as well as an SDN controller to configure the switch and coordinate with other IXP members. The proposed approach to the design of SDX architectures comes with huge benefits in terms of privacy (policies are stored locally), separation of responsibilities (IXPs are not involved in the route computation), and forwarding state scalability (each IXP member only stores its forwarding state). DeSI overcomes the scalability limitation of current SDN-based IXPs by designing a scalable architecture that allows the members to handle their forwarding states.

DeSI has a decentralized architecture in which the tasks of the IXP controller are distributed among the SDN controllers of the members. The controllers of IXP members in DeSI use two complementary mechanisms to install the forwarding state to scale the forwarding state. Through a *proactive* approach, an IXP member installs the whole forwarding state regardless of whether some rules are never matched by actual data packets. This approach has the benefit of quickly handling the incoming traffic but may result in overly large forwarding tables whose size may not be supported by the underlying SDN hardware.[1] Through a *reactive* approach, the DeSI controller installs a forwarding rule only after a packet matching that rule is received by the SDN switch. With this approach, an operator limits the amount of forwarding state needed to support the defined forwarding rules but introduces additional latency and controller load overheads for each packet. DeSI relies on a combination of these two approaches to achieve forwarding state scalability. Finally, we introduce an expressive policy language, inspired by Pyretic [10], that can be used by the IXP members to define their routing policies.

We evaluated our system to assess its practical feasibility. We observed that the most critical resource of DeSI, i.e., the member controller, scales well in the number of policies and BGP announcements being installed.

The rest of the paper is organized as follows. Section II reviews the most relevant contributions for the application of SDN to the inter-domain routing and IXPs. In Section II-B we briefly illustrate the current SDX-based architectures. In Section III we present the architecture of DeSI. Section IV shows our routing policy model. Section V introduces the reactive and proactive approaches, explaining the main differences between them. Section VI describes the architecture of our SDN-controller and Section VII states several applicability considerations. In Section VIII, we show the results collected during the evaluation of our SDN-controller. Discussion come in Section IX. Finally, Section X concludes our paper and discusses the research perspectives opened by our proposal.

[1]SDX routing policies rely on wildcard matching, thus requiring TCAM support from the underlying SDN switch. TCAM space is often limited due to being a power-hungry and expensive resource.

II. Related Work and Background

In this section, we review the state-of-the-art Software Defined Internet eXchange points and SDX-based IXP architectures. We also discuss some of the works on the application of SDN to inter-domain routing, limiting the scope to those that are more related to IXPs.

A. Related Work

An IXP can be seen as a Layer 2 (L2) interconnection network through which IXP member networks connect and exchange routing information using BGP.

SDX [3] is the first attempt to apply SDN to the inter-domain routing inside IXPs. In SDX, standard BGP outbound policies, i.e., policies used to apply on outgoing traffic, are overridden by an SDN controller, improving the flexibility of the BGP protocol. The most relevant SDX contribution consists in replacing the IXP switching fabric with an SDN-capable switch handled by a centralized controller which collects policies from all the members, compiles them, and installs the forwarding state into the SDN capable switch that implements the members' routing policies.

SDX suffers from scalability problems [7] in terms of control plane computation time and the number of generated forwarding rules. Those issues are addressed by an improved version called industrial-SDX (iSDX) [7]. Despite its improvements in scaling the forwarding plane, iSDX still requires exposing the routing policies to the centralized controller and still requires installing too many forwarding rules [9].

Hermans *et al.* [9] report that currently employed switch platforms by the AMS-IX are incapable of supporting iSDX because of the current hardware limitations. Also, the policy compression mechanism of iSDX and the frequency of BGP updates in the memory for large IXPs require further investigation.

Endeavour [11] reduces the number of installed rules on the SDN-enabled switch of an IXP switch fabric (70% less than those of SDX and iSDX). Endeavour is built on top of SDX [3], iSDX [7], and Umbrella [12]. It proposes a new architecture for an IXP switch fabric which is composed of edge and core switches. The rules are installed on edge switches, while the core switches are in charge of forwarding traffic to its designated egress points. This architecture helps to improve the scalability of IXP fabric even if it adds duplication in the forwarding state while installing the inbound and outbound routing policies of the participants. The proposed architecture introduces a mechanism to check (possible) dependencies among the forwarding rules.

Umbrella [13] improves the switching fabric of IXPs by introducing edge and core switches. The edge switches have OpenFlow capabilities in rewriting the layer-2 destination fields whereas the core switches are legacy switches. The privacy and separation of responsibilities of the controller are major concerns because IXPs have been traditionally neutral entities and provide L2 forwarding. Even large IXPs that

TABLE I
COMPARISON OF SDN-BASED SOLUTIONS FOR INTER-DOMAIN ROUTING

Name	Number of controllers	Switch fabric	Handling forwarding rules	Path computation	Privacy and separation concerns	Forwarding state scalability
SDX	1	one switch	SDX controller	controller and BGP	low	low
iSDX	1+one per member	one switch	iSDX controller	controller and BGP	medium	medium
Endeavor	1+one per member	several switches	Endeavor controller	controller and BGP	medium	medium
Umbrella	1	several switches	Umbrella controller	controller and BGP	low	medium
DESI	one per member	IXP dependent (one or more switches)	ISP controller	controller and BGP	high	high

deploy Route Servers do not have access to the private policies of the IXP members.

In contrast to SDX, iSDX, Umbrella, and Endeavor, our architecture does not replace the IXP switching fabric with an SDN-based one. Indeed, we introduce SDN only on the ISP side, without forcing other members to be equipped with an SDN controller. Finally, we preserve backward compatibility with providers that are not interested in using SDN on their side. Table I shows the features offered by each solution.

B. SDX Based IXP Architectures

This subsection describes the typical architecture of an SDN-based IXP (we mainly concentrate on the latest version of SDX, which is called iSDX [7]) discussing the main components and explaining their functionalities. Since DESI builds upon similar forwarding ideas of iSDX, though a completely different architecture, we now introduce details about the iSDX forwarding mechanisms.

1) Components: The main components of the architecture are the following.

1) *An SDN-enabled switch.* To program the switching fabric of an IXP, there is the need for at least one SDN-enabled switch. It is the collector of the policies of all the ISPs that participate in the IXP.
2) *A BGP route server.* Currently, the most important IXPs offer a *route server*. An ISP can substitute its bilateral peerings with just one peering with the route server. The route server computes the best routes to reach the target prefixes and redistributes such routes to the ISPs. The iSDX route server is implemented with ExaBGP [14].
3) *An IXP controller.* The controller cooperates with the route server to integrate the BGP policies with custom outbound and inbound policies.
4) *The members' SDN-controllers.* Each participant in the IXP can have its SDN-controller that shares part of the computations performed by the IXP controller. This improves the scalability of the architecture.
5) *The Members' border routers.* Each member runs (at least) one border router to exchange BGP messages with the router server. The route server can check the BGP reachability information of each member by checking the BGP update messages that come from these devices.

2) SDX and iSDX Architectures: SDX was the first attempt to bring SDN to inter-domain routing, but it does not scale for the following reasons: 1) it generates many SDN rules to handle traffic and currently available TCAM size for SDN-enabled switches is not able to maintain them [9] and 2) the computation time to generate low-level forwarding table entries from high-level forwarding policies, which may change the forwarding behavior of BGP [7], is high.

There are two key design improvements in iSDX to the original SDX proposal. First, the control plane computations of iSDX are partitioned among the participants' controllers to ensure that the routing policies of a participant remain independent from the others. Second, the BGP and the SDN policies are kept separate. This avoids the recomputations that can be triggered when new updates are received. Once the control plane computation is carried out, a forwarding equivalence class (FEC) for each member is created to allow the forwarding of the traffic. For this goal, the reachability information is encoded inside a tag that is stored inside the destination MAC address field of the packets' header. To do so, the multiple match-action tables feature of an OpenFlow-enabled switch is leveraged. iSDX uses one table for inbound and one for outbound policies of each participant. A virtual MAC address is used to encode the reachability information [7].

3) Limits of the Existing SDX Architectures: Although the proposed architectures are the result of deep and sophisticated research work, up to now very few IXPs have adopted SDX or iSDX technologies (just one IXP is based on SDX [11]) due to the hardware limitations [9]. However, in our opinion, the following main concerns [15] should be considered.

The first issue is the privacy of routing policies. Current architectures do not offer a guarantee on the privacy of the policies of the participants. Both the IXP SDN-controller and the Route Server have shared equipment. Anybody that is allowed to enter such machines can access information that can unveil (totally or partially) the policies of the members [16].

A second issue is that the proposed architectures do not allow to separate and identify who is responsible for what in case of failure in traffic delivery. Namely, in a traditional IXP, the center of the architecture is a basic layer-2 switch, with limited intelligence and limited capabilities (in large IXPs this is substituted by more complex layer-2 switch fabric; however, its overall behavior is the one of a simple switch). This allows, in case of problems, to separate the responsibility of

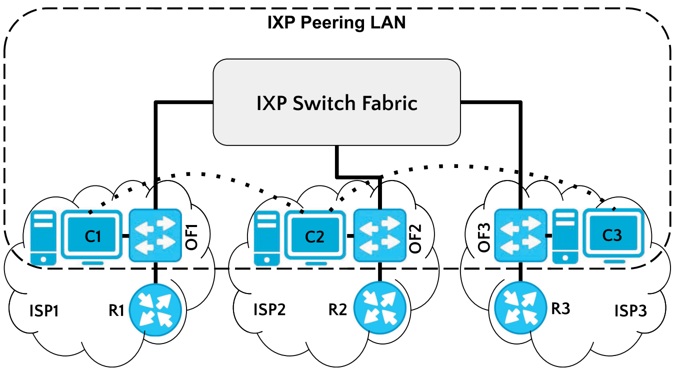


Fig. 1. Our architecture in which SDN is moved in the provider's side to avoid policies sharing and to easily identify responsibilities.

members and the responsibility of the IXP. In SDX and iSDX, the central SDN-enabled switch and the IXP controller are sophisticated machines where the policies of all participants are mixed into a unique container. This avoids having a clear boundary between the ISPs and the IXP. For example, when an Ethernet frame crosses the iSDX fabric, the validity of the policy is checked using the encoded reachability information on the Ethernet header. The iSDX ignores the frame if the encoded reachability information is unreachable.

A third issue is scalability. Integrating into a unique switch the policies of a large-size IXP can be unfeasible due to the current hardware limitations [9].

III. A New SDN Architecture for Internet Exchange Points

In this section, we describe our architecture for an SDN-based Internet eXchange Point. We point out the main differences between SDX and DeSI to show how we overcome the limitations imposed by the SDX architecture. DeSI distributes the operations of the SDX controller to those of members using a decentralized architecture.

Our architecture is depicted in Fig. 1. Each provider joins the IXP with its SDN-enabled switch and its own SDN-controller. In the figure, there are three providers, whose names are `ISP1`, `ISP2`, and `ISP3`. They are connected to the IXP peering LAN by means of SDN-enabled switches, which are called `OF1`, `OF2`, and `OF3`. Each SDN-enabled switch is handled by a specific SDN-controller, namely `C1`, `C2`, and `C3`. Such an architecture does not impose any limitations on the possibility for a provider to be interconnected to multiple IXPs. We provide more details about this scenario in Section VII.

In Fig. 1, routers `R1`, `R2`, and `R3` are IP-speaking devices directly connected, on a specific port, to the SDN-enabled devices and they represent the whole network of each provider. Note that the IXP switch fabric is not SDN-based. We decided to move SDN capabilities inside the network of each provider. This choice has two advantages: first, policies are not stored anymore in a centralized place; second, each provider independently acts on its SDN-enabled device, still having the flexibility offered by SDN, but avoiding the possibility

TABLE II
Table of Notations

$\mathcal{I}$	$\triangleq$	a set of ISPs
$\mathcal{N}$	$\triangleq$	a set of BGP neighbors
$\mathcal{B}$	$\triangleq$	a set of BGP peerings
P	$\triangleq$	a set of policies
p	$\triangleq$	a policy
π	$\triangleq$	a prefix
$\mathcal{P}_i^j$	$\triangleq$	a set of IP prefixes announced by the AS j to AS i
$Pr(p)$	$\triangleq$	the priority of policy p
$M(p)$	$\triangleq$	a set of atoms in the match pattern of policy p
$\mathcal{N}(p)$	$\triangleq$	a set of BGP neighbors of policy p

of compromising the policies of any other IXP member connected to the IXP.

Still referring to Fig. 1, while the bold lines are physical connections, the dotted ones represent BGP peerings. This is another change we introduce. We assume that the peerings are established between SDN-controllers to allow each provider to be as flexible as possible. The most valuable benefit is that it aligns with the current practices of establishing bilateral BGP peerings at IXPs [6], which preserve privacy and enhance visibility into BGP multiple routes.

Another consideration regarding the choice of publicly exposing the controller on the peering LAN. We argue that such a situation is not dangerous for IXP members, for two reasons. First, the peering LAN is typically assumed to be trusted; second, only the BGP speaker component of the SDN-controller is publicly exposed on that LAN.

IV. A Routing Policy Model

In this section, we describe our routing policy model, which is based on a language allowing each provider to forward traffic along multiple paths. Also, we discuss the semantics of our language, highlighting its main properties. Then, we describe the covering problem.

A. Policy Language

Our language does not replace the BGP configuration of members for outbound and inbound policies, i.e., the outbound and inbound policies are BGP filters that determine a set of prefixes that a member desires to receive or send traffic to them. Rather, it can be used in conjunction with the BGP policy specification language to extend the standard BGP capabilities. So, the backward compatibility with standard BGP speakers is preserved. The policy language provides a high-level abstraction for the members to write their routing policies without worrying about the low-level implementation of them in OpenFlow. Also, we do not exploit Pyretic [10], since it is based on the POX controller and it imposes constraints on the controller to use, while our proposal is more general.

We now introduce the model that represents the building block on which we build our language. Table II illustrates the notations used in our policy model.

We model the IXP as the set of the ISPs connected to the IXP itself. Let $\mathcal{B} \subseteq \mathcal{I} \times \mathcal{I}$ be the set of all the BGP peerings established at the IXP. Given any two providers $i_1, i_2 \in \mathcal{I}$, we say that i_1 and i_2 establish a BGP peering if and only if $(i_1, i_2) \in \mathcal{B}$. All the BGP neighbors of an ISP are modeled as a set: $\forall i, j \in \mathcal{I}$, if $(i, j) \in \mathcal{B}$, then $j \in \mathcal{N}_i \subset \mathcal{I}$ and $i \in \mathcal{N}_j \subset \mathcal{I}$, namely j belongs to the set of all the BGP neighbors of i and i belongs to the set of all the BGP neighbors of j, respectively. Finally, given two ISPs i, j such that $j \in \mathcal{N}_i$.

Policies: Policies are the main building block of our language. A policy determines how to route traffic through the Internet. Each provider specifies a set of policies. Given a provider $i \in \mathcal{I}$, P_i is the policy set of i. We define a *policy* $p \in P_i$ as a pair $p = \langle match \rightarrow neighbors \rangle$.

Match part of a policy. The $match$ is a (possibly empty) *expression*. The operators of the expression are the logical operators AND ($\wedge$) and OR ($\vee$). In our language, expressions including the AND operator are evaluated before those including the OR operator. Thus, an expression consists of a set of logical operators and atomic elements. The *atomic elements* of the expression are relational conditions in the form $atom = value$. Each $atom$ is an element of the quadruple $\langle srcip, dstip, srcport, dstport \rangle$, where: 1) $srcip$ is a source IPv4 or IPv6 address; 2) $dstip$ is a destination IPv4 or IPv6 address; 3) $srcport$ is a source TCP or UDP port; and 4) $dstport$ is a destination TCP or UDP port. None of the elements is mandatory: a policy p without any match condition, i.e., M(p), means *all the traffic*. Note that $M(p)$ can be extended by including any matchable field defined in the OpenFlow specification [5].

Neighbors of a policy. The $neighbors$ part of the policy $p \in P_i$ is a list of neighbors $\mathcal{N}_p \subseteq \mathcal{N}_i$, that are candidates to receive the packets that match the policy. If a policy contains more than a neighbor, then the priority to send the matched packets is based on the precedence. The neighbor with higher precedence in $\mathcal{N}_p$ has the highest priority to receive the packets. We assume that a single type of action exists within each policy whose semantic is: *(potentially) forward to a neighbor*. This assumption is not restrictive since our language does not replace the BGP policy specification.

Example. We now state an example of our policy language. Referring to Fig. 1, suppose that `ISP1`has two BGP peerings, one with `ISP2` and another one with `ISP3` (this is made possible by the interconnection with the standard IXP fabric switch), and suppose that it wants to send a portion of its outgoing traffic to `ISP2` and another portion to `ISP3`, according to some field of the packet header. By using a standard BGP policy specification language, this is not feasible, since BGP computes a single best path and all the traffic is forwarded along that path.

Given $ISP1, ISP2, ISP3 \in \mathcal{I}$ and $ISP2, ISP3 \in \mathcal{N}_{ISP1}$, consider the following policies $p_1, p_2 \in P_{ISP1}$:

$p_1 = \langle dstip = 20.1.2.0/24 \wedge dstport = 80 \rightarrow (ISP2, ISP3) \rangle$

$p_2 = \langle dstport = 21 \vee dstport = 22 \rightarrow (ISP3) \rangle$

Also, consider a prefix $\pi = 20.1.2.0/24$ such that $\pi \in \mathcal{P}^{ISP2}_{ISP1}$ and $\pi \in \mathcal{P}^{ISP3}_{ISP1}$. The semantic of p_1 is: *send all the traffic whose destination IP address falls in the subnet 20.1.2.0/24 and whose destination port has value 80 to* `ISP2`*. Else, (either* `ISP2does` *not announce that prefix or it is not reachable for temporary connectivity problems), send that traffic to neighbor* `ISP3`. The semantic of policy p_2 is: *send all the traffic whose destination port has value either 21 or 22 to neighbor* `ISP3`. Observe that p_1 and p_2 use a subset of the available atoms. If not specified in the policy, an atom is considered as wildcard. Also, the BGP routing must support the traffic forwarding through the neighbors specified in the $actions$ part of the matched policy.

Note that policy p_1 allows the traffic to be forwarded to `ISP2` even if `ISP2` is not the best choice for BGP. To send that traffic to `ISP2` it is enough that `ISP2` announces prefix 20.1.2.0/24. We augment the semantics of a policy by implicitly stating that if none of the neighbors specified in the action announces the prefix mentioned in the match, then the traffic is forwarded according to the BGP computation, even if the neighbor to which the traffic is being forwarded is not mentioned in the $actions$ part of the policy.

Policy priority. Our language allows for a double level of priority level. Indeed, the first level is expressed inside the policy when multiple neighbors are defined in the $actions$ list, as reported in policy p_1. In that case, *forwarding traffic to neighbor* `ISP2` has higher priority than *forwarding traffic to neighbor* `ISP3`. The second priority level is among policies. In the example, policy p_1 has higher priority than policy p_2. This means that policy p_1 must be checked always before policy p_2 and the latter one can be considered by the SDN controller if and only if traffic cannot be forwarded according to policy p_1. Hence, the policy priority levels are defined by the order of the policies themselves. Such an ordering might lead to a problem that we call *Covering Problem.*

B. The Covering Problem

As we just said, the order of the policies defines their priority. A naive installation of the rules may lead to certain situations in which some rules will never get matched. This SDN update problem has been first observed in [17]. This circumstance might happen due to human error, and also, it can happen either in proactive or reactive approaches. Before the formal definition of this problem, we show it with an example in Fig. 2. Referring to Fig. 1, suppose that `ISP1` wants to forward traffic with $dstport = 80$ to neighbor `ISP2` and traffic with $dstport = 80$ and the source IP address falling in the subnet $2.0.0.0/8$ to the neighbor `ISP3`. The `ISP1`'s network administrator might write the following two policies:

$p_1 = \langle dstport = 80 \rightarrow (ISP2) \rangle$

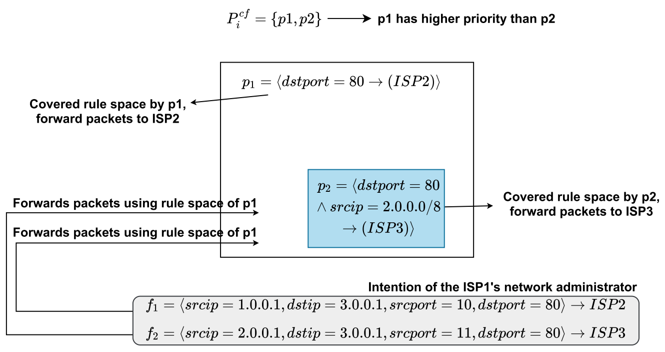
$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \rightarrow (ISP3) \rangle$

Also, suppose that both `ISP2` and `ISP3` send BGP announcements for the same IP prefix π. Now, suppose that two flows must be forwarded according to those policies. In particular, the flows have the following (portion of the) header:

$f_1 = \langle srcip = 1.0.0.1, dstip = 3.0.0.1, srcport = 10, dstport = 80 \rangle$

$f_2 = \langle srcip = 2.0.0.1, dstip = 3.0.0.1, srcport = 11, dstport = 80 \rangle$

and the IP address 3.0.0.1 belongs to the announced IP prefix π, so that it can be reached through `ISP2` or through `ISP3`.



$P_i^{cf} = \{p1, p2\}$
p1 has higher priority than p2
$p_1 = \langle dstport = 80 \to (ISP2) \rangle$
Covered rule space by p1, forward packets to ISP2
$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \to (ISP3) \rangle$
Covered rule space by p2, forward packets to ISP3
Forwards packets using rule space of p1
Forwards packets using rule space of p1
Intention of the ISP1's network administrator
$f_1 = \langle srcip = 1.0.0.1, dstip = 3.0.0.1, srcport = 10, dstport = 80 \rangle \to ISP2$
$f_2 = \langle srcip = 2.0.0.1, dstip = 3.0.0.1, srcport = 11, dstport = 80 \rangle \to ISP3$

Fig. 2. An example of the covering problem with two policies. The controller installs the OpenFlow rules of $p1$ first and all packets belonging to $f1$ and $f2$ are forwarded by $p1$ due to covering problem.

In the intention of the ISP1's network administrator, flow f_1 must be forwarded according to policy p_1, whereas the flow f_2 must be forwarded according to policy p_2. Suppose that the first packets that arrive belong to f_1. Then, the SDN controller selects policy p_1, installing into the open-flow switch the corresponding OpenFlow rule. Upon receiving the flow f_2, the SDN-enabled switch already has the rule to use. Hence, that flow is forwarded according to the OpenFlow rule installed after the selection of policy p_1, resulting in a policy misusage. We call such a problem *Covering Problem*, since policy p_1*covers* policy p_2, preventing its selection.

In contrast to [18], we do not aim at finding dependencies for performance purposes. Indeed, we aim at guaranteeing that the forwarding is performed according to what an ISP wants to achieve.

Before formally showing the *Covering Problem*, we define the *priority* of a policy: given $i \in \mathcal{I}$ and $p \in P_i$, $pr(p)$ is the priority value of p.

Definition. Given $i_1, i_2, i_3 \in \mathcal{I}$ and $p_1, p_2 \in P_{i_1}$ such that $pr(p_1) > pr(p_2)$, we say that p_1 **covers** p_2 if the following two conditions are satisfied: 1) $\mathcal{P}_{i_1}^{i_2} \bigcap \mathcal{P}_{i_1}^{i_3} \neq \emptyset$ and 2) $M(p_2) \subset M(p_1)$.

Roughly speaking, the *Covering problem* states that, given any two policies of the same provider, the policy with the higher priority value must be the policy with the largest match condition set, if one of the two policies has the match condition set that fully includes the other. Note that the *Covering problem* does not occur if the match condition sets of any two policies partially overlap.

To overcome the covering problem, it is enough to give higher priority to the covered policy (p_2 in the example). This results in writing the policies (p_1 and p_2) in the reverse order. Hence, given $i \in \mathcal{I}$, we say that the set $P_i^{cf} = \{p_2, p_1\}$ is the set of cover-free policies, where:

$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \to (ISP3) \rangle$ and
$p_1 = \langle dstport = 80 \to (ISP2) \rangle$

Note that: 1) the policies set P_i^{cf} is not affected by the covering problem, and 2) such a new policies order makes $pr(p_2) > pr(p_1)$. If the order of the policies induces a covering problem, DESI must arise a notification, without undertaking any specific action (e.g., by executing any re-ordering algorithm for the policies). This means that this problem does not depend on the BGP announcements, since it is only a static check of the policies.

V. From Policies to Forwarding Rules

After a computational process inside the SDN controller, a policy is translated into one or more suitable forwarding *rules* to be installed inside each SDN-enabled switch (e.g., OpenFlow rules). Such rules allow the device to forward the traffic to the proper neighbors.

First, we clarify the difference between policies and forwarding rules. A policy represents a high-level way to declare how traffic must be routed in the network while a forwarding rule is the translation of that policy resulting in suitable data structures on the SDN-enabled switches. In our case, each policy is translated into one or more OpenFlow rules. More details are given in Section VIII.

We present two approaches namely, *Reactive Approach* and *Proactive Approach*. The first one performs the translation from policies to the forwarding rules when the traffic reaches the switch, whereas the latter one computes such a translation before any packets reach the device. We explain the benefits and drawbacks of each approach.

Memory cost. The proactive approach reduces the packet waiting time at the switch. However, it comes with extra memory cost to store the flow table entries. While the reactive approach overcomes this problem by installing the necessary forwarding rules in the forwarding tables.

There are no limitations in adopting one of the two approaches under different conditions. We describe the details of those approaches in the following.

A. The Reactive Approach

In the reactive approach, several conditions must be taken into account. First of all, there could be *dependencies* among policies. If such a situation happens, the SDN-controller must be able to detect it and acts properly. To support that task, we define the *Dependency Graph* which is a graph modeling specific relations among policies.

The Dependency Graph

This subsection describes the dependencies among the policies and an approach to detect it.

The policies set P^{cf}, shown in Section IV, is not affected by the covering problem. However, this policy set might be affected by another issue. Suppose that a traffic flow matching the policy p_2 (e.g., $dport = 80$) reaches the SDN-enabled switch and there is no suitable rule in the forwarding table of the switch to match that flow. According to the reactive approach, a packet of that flow is sent to the controller asking for the suitable policy to apply. After selecting the policy p_2 and translating it into a forwarding rule, the SDN-enabled switch can route the packets.

Problem. Suppose there is a rule on the SDN-enabled switch to handle the traffic matching the policy p_2 (e.g., $dport = 80$). Now, a new traffic flow arrives at the same SDN-enabled switch

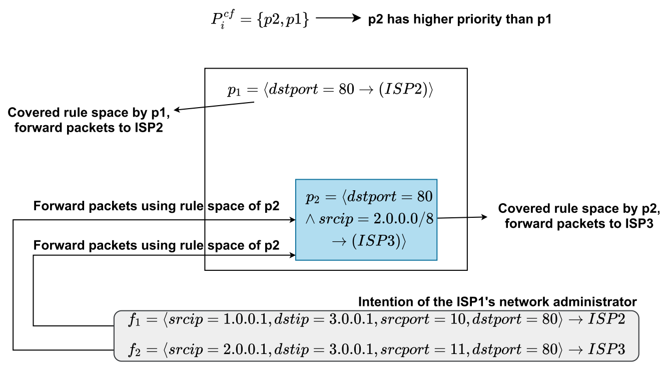

$P_i^{cf} = \{p2, p1\}$
p2 has higher priority than p1
$p_1 = \langle dstport = 80 \rightarrow (ISP2)\rangle$
Covered rule space by p1, forward packets to ISP2
$p_2 = \langle dstport = 80 \wedge srcip = 2.0.0.0/8 \rightarrow (ISP3)\rangle$
Covered rule space by p2, forward packets to ISP3
Forward packets using rule space of p2
Forward packets using rule space of p2
Intention of the ISP1's network administrator
$f_1 = \langle srcip = 1.0.0.1, dstip = 3.0.0.1, srcport = 10, dstport = 80\rangle \rightarrow ISP2$
$f_2 = \langle srcip = 2.0.0.1, dstip = 3.0.0.1, srcport = 11, dstport = 80\rangle \rightarrow ISP3$

Fig. 3. An example of rule dependency. The controller installs the OpenFlow rules of $p2$ first and all packets belonging to $f1$ and $f2$ are forwarded by $p2$ due to rule dependency.

and it has still in the header of the packets the value $dstport = 80$, but the source IP address now falls in the subnet $2.0.0.0/8$. This traffic flow must be forwarded according to the policy p_1, but it is not, since that traffic flow matches the previously installed forwarding rule (e.g., the rule obtained from the policy p_2). Fig. 3 illustrates such a situation. Through this very simple example, it is evident that two policies, or more, might depend on each other. In particular, this is true when a lower priority policy is matched before a higher one.

Solution. To find the dependencies among the policies, we introduce the concept of *Dependency Graph*.

Definition. The dependency graph is a directed graph $G = (V, E)$ modeling dependency relationships among the policies, in which: 1) V is the set of vertices. Each vertex represents a policy. Hence, we say that $V = P$, and 2) E is the set of the edges. Each edge is a pair $\langle v_1, v_2\rangle$ where $v_1, v_2 \in V$. Since the graph is oriented, the pair $\langle v_1, v_2\rangle$ represents an edge from v_1 to v_2. Given a set of policy P, the dependency graph for that set of policy is a graph $G = (P, E)$ where P is the set of the vertices, each of which models a policy, and $\forall p_i, p_j \in P$ where $i \neq j$, $\langle p_i, p_j\rangle \in E$ if and only if the two following conditions are satisfied: 1) $Pr(p_i) < Pr(p_j)$ and 2) $M(p_i) \cap M(p_j) \neq \emptyset$.

We now state an example to show how a dependency graph is built for a given set of policy P^{cf}. The graph $G = (P^{cf}, E)$ is the dependency graph for the set P^{cf}, where $P^{cf} = \{p_1, p_2\}$, and $E = \{\langle p_2, p_1\rangle\}$. In fact, $Pr(p_2) < Pr(p_1)$ and $M(p_2) \cap M(p_1) = \langle dstport = 80\rangle$. The conditions for the set of policy P^{cf} are satisfied which result is having a dependency among $\{p_1, p_2\}$.

We highlight that the covering problem and the dependency graph address two different problems, but complementary. In particular, the covering problem is the problem of a higher policy which prevents the selection of a lower one, whereas the dependency graph is a data-structure aiming at avoiding to forward the traffic according to a lower priority policy in place of a higher one if present. Alg. 1 builds the dependency graph.

After building the dependency graph, the SDN-controller is now able to produce the suitable set of forwarding rules that allow the traffic to be forwarded without any mistakes. This step is called *Expansion Process* and we explain it in the following.

Algorithm 1. Creating dependency graph among the policies

```
1:  **Input** The set of policies ($P$) for a controller
2:  **Output** The set of dependent policies.
3:  **procedure** CreateGraph
4:    state all policies as $M(p)$
5:    **for** each $p \in P$ **do**
6:      create a vertex for each policy $p$
7:    **end for**
8:    $M(p_i) \leftarrow match$ fields of vertex i
9:    $M(p_j) \leftarrow match$ fields of vertex j
10:   **for** i=1 to $|P|$ **do**
11:     pick a policy $p$
12:     pick corresponding vertex for $p$
13:     **for** j=1 to $j < i$ **do**
14:       pick the policy for vertex j
15:       **if** $M(p_j) \subset M(p_i)$ **then**
16:         add an edge from vertex i to vertex j
17:       **else if** $(M(p_j) \not\supseteq M(p_i)) \wedge (M(p_j) \cap M(p_i) \neq \emptyset)$ **then**
18:         add an edge from vertex i to vertex j
19:       **end if**
20:     **end for**
21:   **end for**
22: **end procedure**
```

The Expansion Process

In this section, we describe how the policies are translated into forwarding rules. We explained in Section IV that our routing policy model uses two operators: AND and OR. Since the match part of an OpenFlow flow entry can be seen as a sequence of match conditions evaluated by using the AND operator (e.g., a packet matching **all** the fields specified in the match condition), we start our explanation by considering a policy whose match part consists of a set of matching conditions using the AND operator. After that, we show how policies including the OR operator are translated into an equivalent set of policies that only use the AND operator.

As the first step of the expansion process, we build a tree for each policy in which the parent node in the tree indicates the used operator and the leaves of the tree show the match fields. Fig 4 depicts an example of such a representation. Then, we run the Depth-First Search (DFS) algorithm on the tree to create the forwarding rule to install on the device.

According to Section IV, we consider *wildcard* $(*)$ for all the other match conditions which do not explicitly appear in the policy itself, and we assume them in AND with all the other match conditions.

Relying on this representation, we are now able to represent a policy containing the OR operator into a set of policies only containing the AND operator. Consider the policy:

$$p = \langle dstport = 21 \vee dstport = 22 \rightarrow (n)\rangle$$

According to the representation, we build the tree shown in Fig. 5. We run the DFS algorithm on this tree. Each time a node containing an OR operator is visited, a new sub-policy is created. By doing so, the policy p leads to two policies p' and p'' that only contain the AND operator. In particular, those two policies are:

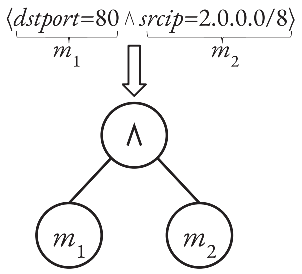


Fig. 4. Tree representation of a policy only containing the AND operator.

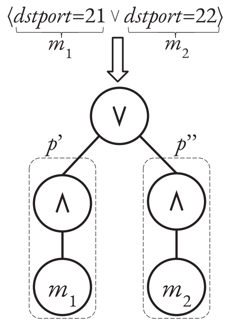


Fig. 5. Tree representation of a policy containing the OR operator.

$p' = \langle dstport = 21 \rightarrow (n)\rangle$
$p'' = \langle dstport = 22 \rightarrow (n)\rangle$

To better clarify the presence of wildcards and operators, p' and p'' can be seen in the following way:

$p' = \langle srcip = * \wedge dstip = * \wedge srcport = * \wedge dstport = 21 \rightarrow (n)\rangle$

$p'' = \langle srcip = * \wedge dstip = * \wedge srcport = * \wedge dstport = 22 \rightarrow (n)\rangle$

As the second step of the expansion process, we actually *expand* the policy. This includes checking the policy with BGP routing information. To do that, we need to interact with the BGP Routing Information Base (RIB). Indeed, once a packet arrives at the switch, a policy is selected. Then, we check whether there is an entry in the BGP RIB allowing that packet to be forwarded according to the $action$ part of the policy. If so, the policy is expanded. This means that, if no destination IP address is specified in the matching part of the policy, that IP address is added, after a lookup in the RIB. We describe it in more detail.

Suppose that a packet matching policy p arrives at the SDN-enabled switch, whose destination IP address is π. Since no destination IP address is specified in p ($dip = *$), that value must be specified, to avoid a possible mismatch with other forwarding rules. So, a lookup in the BGP RIP is carried out, checking whether $\pi \in \mathcal{P}(n)$. If it is the case, the policy p is *expanded* by setting $dip = \pi$. Hence p becomes:

$p = \langle (dstport = 21 \vee dstport = 22) \wedge dstip = \pi \rightarrow (n)\rangle$.

The expansion process represents the last step resulting in the creation of a set of forwarding rules. Algorithm. 2 shows the pseudo-code of the expansion process.

Time complexity of Algorithms. Algorithm 1 creates a vertex for each policy p in $O(p)$. Then, it has to check the dependency among all the policies which needs $O(p^2)$. Therefore, the overall time complexity of this Algorithm 1 is $O(p^2)$. Algorithms 2 uses the binary search tree (BST) to create the graph which needs $O(V)$ for insertion and search. This algorithm uses DFS tree traversal, which runs in $O(|V| + |E|)$. However, we need $O(\mathcal{R})$ to check the π in BGP RIB. Considering p policies, Algorithms 2 requires $O(p|V| + p|E|)$ to complete the forwarding rule generation.

Algorithm 2. Policies expansion

```
1:  Input The set of policies ($P$) for a controller, $\mathcal{P}(n)$ IP prefixes
    announced by member $n$
2:  Output The set of forwarding rules $\mathcal{R}$
3:  procedure ExpandPolicy
4:    $\mathcal{R} = \emptyset$
5:    for each $p \in P$ do
6:      create a tree from $M(p)$
7:      if root of tree is $\wedge$ then
8:        create a rule $r$ for the tree
9:        $\mathcal{R} = \mathcal{R} \bigcup r$
10:     else if root of tree is $\vee$ then
11:       for each child of root do
12:         create a rule $r$ for each child
13:         $\mathcal{R} = \mathcal{R} \bigcup r$
14:       end for
15:     end if
16:   end for
17:   for each $r \in \mathcal{R}$ do
18:     lookup $\pi$ in BGP RIB
19:     if $\pi \in \mathcal{P}(n)$ then
20:       set $r$.dstip=$\pi$
21:     end if
22:   end for
23: end procedure
```

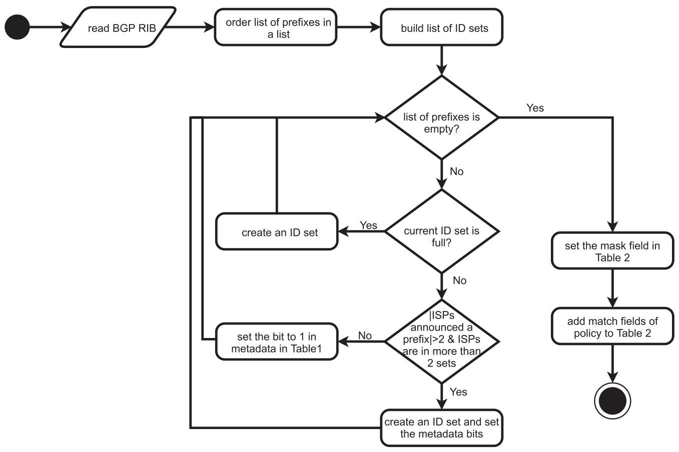
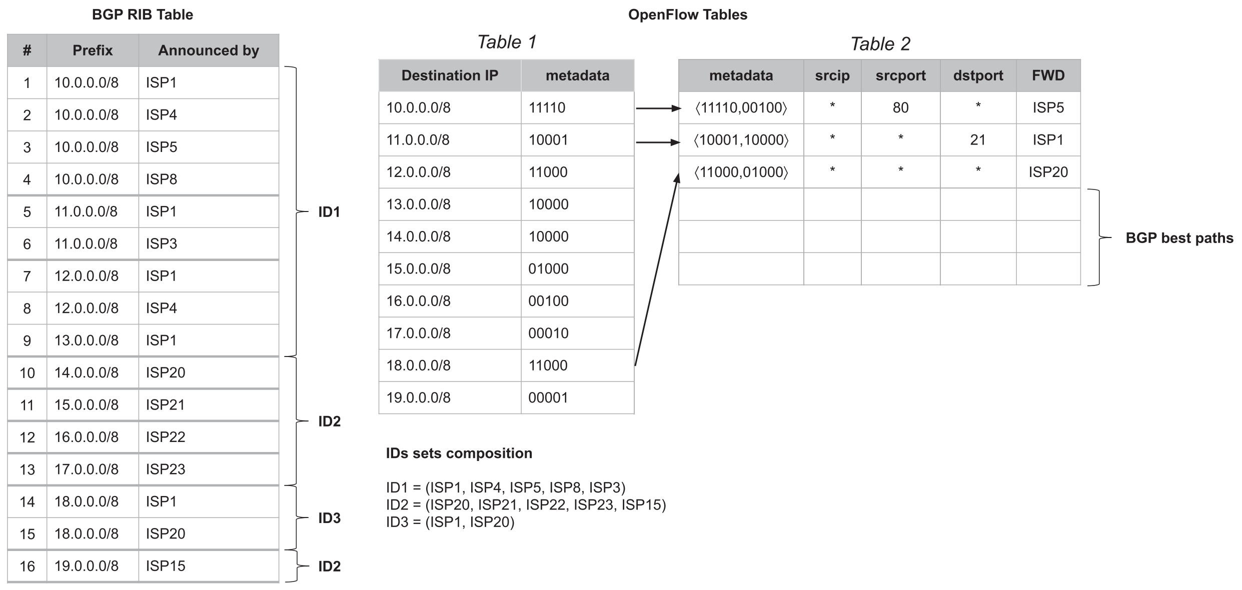

read BGP RIB
order list of prefixes in a list
build list of ID sets
list of prefixes is empty?
Yes
No
current ID set is full?
Yes
create an ID set
No
|ISPs announced a prefix|>2 & ISPs are in more than 2 sets
No
set the bit to 1 in metadata in Table1
Yes
create an ID set and set the metadata bits
set the mask field in Table 2
add match fields of policy to Table 2

Fig. 6. The flowchart of the proactive approach.

B. The Proactive Approach

In the proactive approach, all the policies are translated into forwarding rules before the traffic flows reach the SDN-enabled switch, namely during the startup phase of the SDN-controller. Fig. 6 shows the flowchart of this approach.



BGP RIB Table

#	Prefix	Announced by
1	10.0.0.0/8	ISP1
2	10.0.0.0/8	ISP4
3	10.0.0.0/8	ISP5
4	10.0.0.0/8	ISP8
5	11.0.0.0/8	ISP1
6	11.0.0.0/8	ISP3
7	12.0.0.0/8	ISP1
8	12.0.0.0/8	ISP4
9	13.0.0.0/8	ISP1
10	14.0.0.0/8	ISP20
11	15.0.0.0/8	ISP21
12	16.0.0.0/8	ISP22
13	17.0.0.0/8	ISP23
14	18.0.0.0/8	ISP1
15	18.0.0.0/8	ISP20
16	19.0.0.0/8	ISP15

Table 1

Destination IP	metadata
10.0.0.0/8	11110
11.0.0.0/8	10001
12.0.0.0/8	11000
13.0.0.0/8	10000
14.0.0.0/8	10000
15.0.0.0/8	01000
16.0.0.0/8	00100
17.0.0.0/8	00010
18.0.0.0/8	11000
19.0.0.0/8	00001

Table 2

metadata	srcip	srcport	dstport	FWD
⟨11110,00100⟩	*	80	*	ISP5
⟨10001,10000⟩	*	*	21	ISP1
⟨11000,01000⟩	*	*	*	ISP20

IDs sets composition

ID1 = (ISP1, ISP4, ISP5, ISP8, ISP3)
ID2 = (ISP20, ISP21, ISP22, ISP23, ISP15)
ID3 = (ISP1, ISP20)

Fig. 7. Example of how to use metadata to encode network reachability information.

As reported in Section II-B, iSDX relies on the destination MAC address to encode all the reachability information, namely the set of neighbors in which a flow can be sent. We rely on the same encoding iSDX implements, but we exploit the OpenFlow metadata registry [19]. It has two advantages: first, we can encode more information, since the metadata consists of 64 bits and future architecture may provide even larger metadata fields while MAC addresses will be limited to 48 bits. Second, since we do not change the destination MAC address, we do not need to inject additional ARP traffic in the network reducing the overhead of the whole IXP's peering LAN.

Now, we show how we exploit the OpenFlow metadata to encode network reachability information. We assume that:

1) the BGP RIB table is ordered based on the announced IP prefixes;
2) the metadata is a pair $\langle ID, mask\rangle$ and they have the same length. This assumption is supported by the OpenFlow specification [19].

We use two OpenFlow tables to forward traffic according to the policies defined at the SDN-controller.

Fig. 7 shows an example of how we encode the network reachability information in the OpenFlow metadata. To do that, we rely on several sets. Each element of these sets is an ID, namely a value that uniquely identifies a provider (e.g., the Autonomous System number). For simplicity, we assume that the metadata is 5 bits length so that each IDs set exactly contains 5 elements as well which will be used as the mask.

The proactive approach consists of three steps: 1) building the set of IDs that are used to populate the metadata registry. 2) filling the OpenFlow Table 1, and 3) filling the OpenFlow Table 2. First, we build a set of IDs sets from the BGP RIB table in Fig. 7 to fill the content of the destination IP field of OpenFlow Table 1. Then, DeSI fills the metadata field of Table 1 by setting the IDs. Finally, the values of the metadata column of OpenFlow Table 1 and the corresponding masks field are filled in the metadata column of OpenFlow Table 2. Here, DeSI uses the other match fields of the rule to forward the traffic to the corresponding ISP. We now explain the encoding steps in more detail.

First, our controller starts to scan the BGP RIB table. It finds that provider ISP1 announces the prefix $10.0.0.0/8$. So, the first set, called ID1, is created and the first element of that set is ISP1. Still scanning the BGP RIB table, ISP4 announcing $10.0.0.0/8$ is the second entry found. Since we have space in the set ID1 (4 bits are still available), ISP4 is included in that set. Such a process continues until the set ID1 is full. This condition happens when reaching the BGP RIP entry number 9. Once entry number 10 is being scanned, a new ISP is found, namely ISP20. Since the set ID1 is full a new set called ID2 is created and the bit in position one corresponds to ISP20. This is the first condition that triggers the creation of a new ID set. The second condition that triggers the creation of a new set is the following. If a prefix is announced by two or more providers that are already inside two or more IDs sets, then a new ID set is needed. Such a condition applies when scanning lines 14 and 15 of the BGP RIP table. Indeed, the prefix $18.0.0.0/8$ is announced by providers ISP1 and ISP20. Since ISP1 is in set ID1 and ISP20 is in set ID2 (because of prefixes $10.0.0.0/8$ and $14.0.0.0/8$, respectively), a new set, called ID3, is needed. The first step is accomplished.

The second step, namely filling the OpenFlow Table 1, is carried out. The first OpenFlow table contains an entry for each prefix in the BGP RIB table. Each entry is associated with a metadata value that is built in the following way: there is a bit set to 1 for each provider that announces the prefix. The choice of which bit is set to 1 depends on the position of the ID in the set. As an example, consider the first entry of the OpenFlow Table 1. Prefix $10.0.0.0/8$ is announced by ISP1, ISP4, ISP5, ISP8. Since those providers belong to the set ID1, the metadata value for the considered prefix is 11 110.

Finally, the third step, namely filling the OpenFlow Table 2, is accomplished. This step involves the policies defined at the

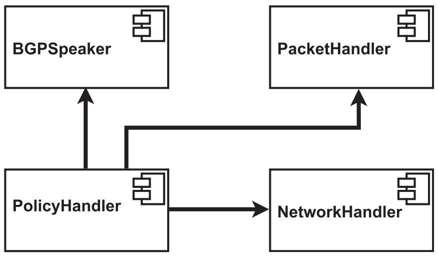


Fig. 8. High level view of the internal architecture of our SDN-controller.

controller. Indeed, there is an entry for each policy defined. Consider the policy:

$p_1 = \langle dstport = 80 \rightarrow (ISP5)\rangle$

and a traffic flow:

$f_1 = \langle srcip = 1.0.0.1, dstip = 10.0.0.1, srcport = 10, dstport = 80\rangle$

then an entry is built in the following way: since the destination falls in the subnet $10.0.0.0/8$ and this subnet is announced by providers ISP1, ISP4, ISP5, and ISP8, the metadata 11 110 must be used. Since the policy p_1 must be taken into account and the metadata value does not give any information about which neighbor the traffic must be forwarded to, a mask is needed. The policy p_1 has ISP5 in the action atom, the mask 00 100 is used to forward the traffic. This process results in the creation of the first entry for the OpenFlow Table 2 in Fig. 7. After the last forwarding rule is installed because of a policy, this table contains the rules for forwarding the traffic simply according to the BGP best paths. We recall that this is the forwarding rule we apply whether the incoming traffic does not match any policies or whether the BGP routing does not allow the traffic to cross the neighbors specified in the action part of the policies. Note that a complete reset of forwarding tables happens when the policy change affects all the other policies of a member. Otherwise, the routing entries of the changed policies are re-computed and the affected OpenFlow forwarding rules are updated in the forwarding tables.

Link failure. We rely on the Fast-Failover Group in the OpenFlow specification to react to the link failures [20]. A corresponding table in the switch can be configured to monitor the status of ports and interfaces to undertake an action independently of the controller.

VI. THE ARCHITECTURE OF OUR SDN-CONTROLLER

In this section, we illustrate the internal architecture of our controller. We show the main components of our system and how they cooperate to allow the traffic to be forwarded according to the policies defined by the user.

The internal architecture of our SDN-controller is shown in Fig. 8. It consists of four main components, called: 1) BGPSpeaker, 2) PacketHandler, 3) Network Handler, and 4) PolicyHandler. We now discuss each component in detail.

The BGPSpeaker component implements a BGP speaker being able to establish BGP peering with other speakers, to receive and to announce the BGP packets, and to maintain a full BGP RIB. This component is crucial for two main reasons: it guarantees backward compatibility with the standard IP-speaking routers running the BGP protocols and it allows the policies to be expanded, according to the process widely described in Section V.

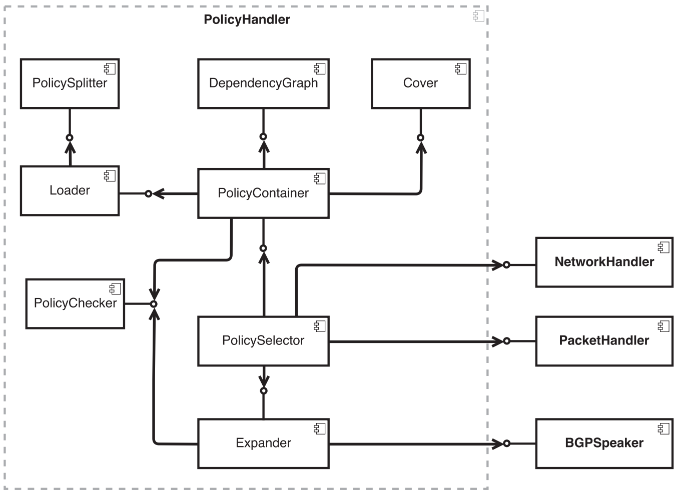


Fig. 9. A detailed view of the internal architecture of the PolicyHandler.

The PacketHandler component offers basic functionalities for parsing and creating standard packets used by the controller to accomplish its tasks. For example, our SDN-controller relies on this component to handle the ARP traffic exchanged on the peering LAN of the IXP.

The NetworkHandler component allows the SDN-controller to interact with each SDN-enabled switch. This component implements most of the functionalities described in Section VII.

Finally, PolicyHandler is the core component of the DeSI controller which implements all the algorithms described in this paper. We explain it as follows.

The internal representation of the PolicyHandler is depicted in Fig. 9. It consists of a set of sub-components, each of which performs a specific task. The Loader component simply loads the policies (e.g., from a file) and builds the suitable data-structures representing them. The policyHandler performs a check on the input policies using the PolicyChecker. It cooperates with the PolicySplitter component to build policies only containing the AND operator, according to Section V. After the policies have been loaded, they are stored in the Policy-Container component, which exploits the DependencyGraph component to build the graph of policies dependencies. Also, by interacting with the Cover component, the PolicyContainer can raise a warning in case of a problem of covering among policies is happening. Now, the policies are available to be selected and then translated into forwarding rules according to our approaches.

The PolicySelector component selects a policy from the set of policies. In the case of the *Reactive* approach, this component is triggered once a packet reaches the network devices; if the *Proactive* approach is running, then it is triggered in advance (e.g., during the start-up phase of the controller). The interaction with the network devices explains why it exploits the NetworkHandler component, whereas the interaction with the PacketHandler is justified by the need of interacting with the traffic. Finally, it also interacts with the Expander to carry out the expansion process described in Section V. To perform such a step, the Expander needs to exploit the BGPSpeaker component, which provides a simple way to access the information contained in the BGP RIB. We recall that such an interface can be made available since that component implements a fully standard BGP speaker capable of establishing BGP peering. Note that after calling the Expander the controller checks the rule using PolicyChecker.

VII. Applicability Considerations

In this section, we discuss several aspects related to the applicability of DeSI. We focus on considerations about specific scenarios and backward compatibility with standard (or legacy) solutions (e.g., interconnection with IP-speaking nodes running the BGP protocol).

Commonly, a provider is interconnected to many IXPs. Such a choice is typically motivated by either resilience or performance reasons. In the first case, a provider typically implements the primary-backup strategies over the peering, whereas, in the second scenario, load-balancing policies are applied. In every case, there might be the need of having multiple controllers. On one hand, more controllers represent a valid fault-tolerance strategy. On the other hand, performance increases when each device is handled by its controller, especially in the case of processing the full routing table.

Many techniques can be adopted to design solutions using multiple controllers. We now discuss several of them. The first solution is built according to the master-slave architecture, consisting of a pair of controllers. A controller of that pair (called *master*) is managing the SDN-enabled devices and the second one (called *slave*) starts to act when the other fails. In case robustness is very crucial, more than two controllers can be used, resulting in a cluster. In this case, we assume that both master and slave controllers handle all the SDN-enabled devices. Sometimes, such an architecture is natively supported by OpenFlow devices. Indeed, it is possible to set two (or even more) controllers during the device configuration: one acting as master and the others acting as the slave. In this scenario, the OpenFlow device typically sends the same information to all the controllers, allowing them to have the internal knowledge of the network perfectly aligned. Relying on keep-alive messages, the switch can verify if the master controller is running or not. In case of failure, the device immediately changes the controller, giving to the slave the role of master, being ready to change once again as soon as the master becomes again reachable.

If such an operational way is not supported by the device, multiple controllers can still be used. Nevertheless, the synchronization among the controllers is demanded to the controllers themselves, that now have to exchange information about the SDN-enabled devices autonomously, without any kind of support from the device. Such synchronization is carried out according to standard strategies used in distributed systems (e.g., cold or warm approaches) so that the slave can replace the master without any lack of information. Surely, other solutions can be implemented, provided that the internal state of the controllers is aligned when the master fails and the slave replaces it.

Another scenario involving multiple controllers is the following. A provider might choose to have a single controller for each IXP it is connected to. The main difference with the previous scenario is that in this case, each controller handles a single device, or in general, a subset of the whole devices the provider has in different IXPs, whereas in the master-slave approach each controller handles all the devices. Even in this approach, controllers must synchronize their internal states. As a solution, iBGP peering among the controllers can be set as in standard architectures. Also, route reflectors strategies can be applied for increasing the scalability.

A. Backward Compatibility

Another consideration in terms of applicability is referred to as the backward compatibility. Indeed, our architecture is fully compliant with standard (or legacy) ones. There are no limitations in establishing BGP peering with other providers that use IP-speaking nodes. Our solution does not force other providers in the IXP to have an SDN-controller. Our SDN-controller can establish BGP peering either with other SDN-controllers or standard IP-speaking routers without requiring any specific configuration on both sides.

B. Route Server

As the final consideration, we discuss route servers. Commonly, IXPs offer the possibility to each participant to establish BGP peering with one or more router servers. A route server is a collector of BGP announcements, allowing providers to have multiple logical interconnections by setting up a single BGP peering instead of multiple ones. Even in this case, our SDN-controller can establish peering with the route server, with no technological limitations. There could be just a limitation in terms of the possibility to choose among different paths. Indeed, a route server typically computes a single best path and then only that path is announced along with each BGP peering. Such an operational way reduces the number of available alternatives that each provider has. Apart from that, there are no restrictions.

Even if other considerations might be done, we argue that what we discussed in this section is a significant sample addressing the most important aspects related to the adoption of our proposal in production environments.

VIII. Evaluation

To validate the practical applicability of our approach, we implemented a prototype version of DeSI based on the Ryu framework [21], since it provides an implementation of a standard BGP speaker. We used Kathará [22] as a framework to create a network and to run all the SDN components of the testbed. We focus our measurements on scalability aspects, taking into account the impact of our proposal in terms of resource consumption and the time needed to perform its

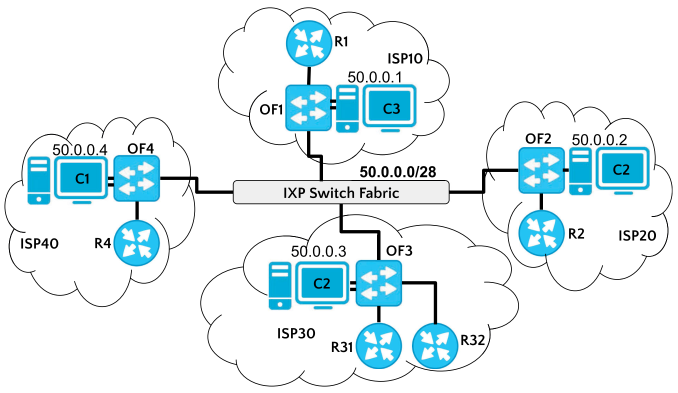


Fig. 10. The topology used for the functionality tests. It simulates a simple IXP consisting of four members, each announcing just one prefix.

tasks.Our simulations consisted of two parts. First, we built a small IXP in which our implementation run, to test the functionality of our controller. Second, we focused on resource consumption on the machine hosting the controller and the time spent by DeSI to carry out its activities. The tests were carried out varying both BGP announcements and the number of policies for both the *Reactive* and the *Proactive* approaches. The number of BGP announcements varies up to 800 k BGP announcements and the number of policies is in the range of [500,2000] resulting in having 1 million policies for an IXP with 500 members.

A. The Testbed

We run our experiments in an Ubuntu virtual machine equipped with 16 CPU cores at 1.9 GHz and 64 GB of RAM. We use Python psutil library to measure the CPU and RAM utilization.

Fig. 10 shows the topology used to run our functionality experiments. The network contains a simple IXP consisting of four members (AS10, AS20, AS30, and AS40), each equipped with an SDN-enabled switch (OF1, OF2, OF3, and OF4) and with an SDN-controller (C1, C2, C3, and C4). Inside each provider's network, we place a standard IP-speaking node (R1, R2, R31, R32, and R4) representing, without loss of generality, the whole network of the provider itself. In the case of member AS30, we use two nodes (R31 and R32) to reproduce the case in which a provider connects multiple border routers in the IXP. This is typically done for robustness or performance purposes, like primary/backup or load balancing strategies, respectively. The IXP switch fabric is a legacy layer 2 switch.

In the testbed, we assume that each controller is directly connected to its corresponding SDN-switch. This assumption is not restrictive. Indeed, since we do not introduce any constraints on the provider's backbone, we only need IP connectivity between SDN-controller and SDN-switch. Note that there are two connections between those components. Such connections represent logical links: one link is used for the OpenFlow messages, whereas the second one is used for the BGP messages. We highlight that this interconnection is logical and not physical. Indeed, every technology that guarantees traffic isolation can be used (e.g., VLAN).

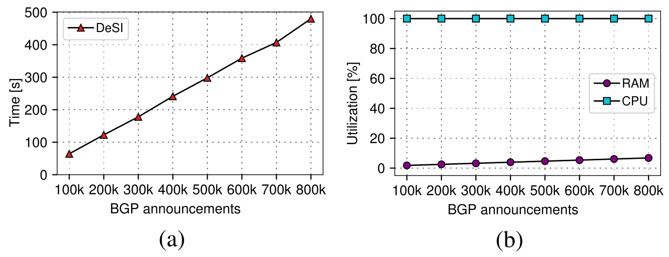


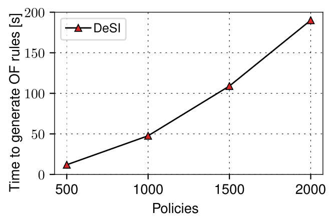
Fig. 11. The performance of DeSI with up to 800 k BGP announcements. (a) Time to handle BGP announcements. (b) Resource utilization.

Each SDN-controller establishes a BGP peering with all the other controllers. Within those peerings, each provider announces a single prefix. Also, each SDN-controller gets its policies from a file. No restrictions are applied, namely, each controller does not filter anything, resulting in a full-mesh of peerings. Therefore, the controllers of the members act independently. For this experiment, we run both reactive and proactive approaches. We considered the following conditions: 1) We checked that each controller was able to successfully perform ARP requests over the peering LAN. This check is needed to allow the BGP messages to reach the right controller. 2) We checked that each BGP announcement was able to reach any other provider. We also checked that the announcements were successfully stored in the BGP RIB of each controller. 3) We checked that the traffic generated by each provider towards each known destination in the IXP was correctly forwarded according to the policies of each member. The above functionality experiments were successfully carried out for each approach. Namely, we observed that our implementation works as expected.

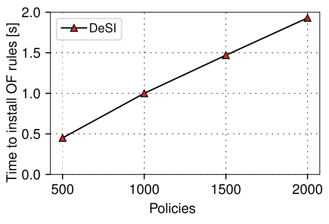
For those experiments, we focus on a pair of SDN-controllers having a peering between them. For privacy concerns, DeSI stores the routing policies of each member within its physical devices and there is no single controller DeSI handling traffic of all IXP members. Therefore, we focus on the resource consumption of DeSI.

B. DeSI Performance

DeSI Stress test. We measure the performance of DeSI with up to 800 k BGP announcements for a large-scale evaluation purpose (see Fig. 11). Fig. 11(a) shows that the required time for DeSI to perform BGP announcements linearly increases by increasing the number of BGP announcements. DeSI needs $\approx$ 8 minutes to announce 800 k prefixes. Fig. 11(b) depicts the percentage of RAM and CPU utilization for announcing the same number of prefixes. The RAM utilization slightly increases by increasing the number of BGP announcements and DeSI uses $\approx$ 7 percent of RAM to announce 800 k prefixes. Note that we fully utilize the CPU of the controller in this experiment and each controller uses just one CPU core for performing its operations. In comparison, today's RSes report convergence times ranging between 3 and 10 minutes [23],

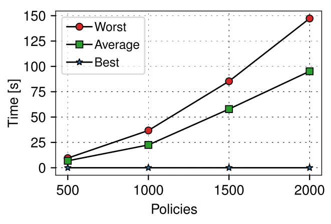


(a)

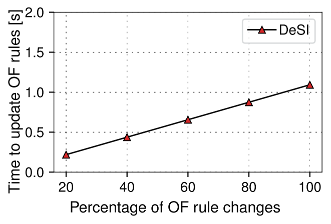


(b)

Fig. 12. The performance of DESI. a) Time to translate the policies into the OpenFlow rules. b) Latency to install the OpenFlow rules generated by the set of policies.



(a)



(b)

Fig. 13. The performance of DESI. a) Time to check the policies for the covering problem in the best, average, and worst-case scenarios. b) Time to update the OF rules impacted by policy changes.

[24]. It is worth noting that our code is not optimized for production and is just a prototype.

Time analysis. We measure the required time for DESI to perform the following tasks: 1) translate policies into OpenFlow rules; 2) install the OpenFlow rules; 3) redo the above tasks when a set of policies changes. We vary the number of policies in the range of [500,2000] for each participant while considering an IXP with 500 participants results in having 1 million policies. Herein, we report the time analysis results of the proactive approach. Fig. 12(a) and Fig. 12(b) show the required time by DESI to translate the participants' policies into OpenFlow rules and to install them into the OpenFlow switch, respectively. The required time for DESI to generate OF rules from the participant's policies is at most 190 seconds for 2000 policies (see Fig. 12(a)). However, the required time to install the generated forwarding rules for the same policies is less than 2 seconds. Note that, in the case of translating policies into OpenFlow rules, we are not considering the time induced by the network latency.

We now check the performance of DESI for scenarios when a provider modifies or changes its policies and measure the required time to recheck the policies for the covering problem for different scenarios. We assume that in the best case, policies of the members stay unchanged while half of them change in the average case. We also assume that all policies change in the worst case resulting in a complete reset of routing policies. Fig. 13(a) shows that by increasing the number of policy changes, DESI requires more time to recheck for the covering problem in the average and worst cases. Finally, we report the time to update the OpenFlow rules resulted from policy changes. Fig. 13(b) illustrates that by increasing the percentages of policy changes, DESI needs more time to update the OpenFlow forwarding rules after checking the covering problem. Indeed, it requires $\approx$ 1 s to update all the OpenFlow rules if all the policies change.

C. Comparison With iSDX

We now compare DESI with iSDX [7] in terms of different design and performance parameters. DESI keeps the IXP fabric as-is and does not install any forwarding rule at IXP fabric. The forwarding rule computation in DESI is done using participant controller whereas in iSDX the participant controller does the computations and sends the forwarding rules to the iSDX controller to install in IXP fabric. In the latter case, the overall computation time comes from the summation of the computing time of the participant controller and iSDX controller plus the communication delay among them. Assuming the same data-trace of iSDX in [25], we compare the two systems as follows.

Forwarding table entries. DESI encodes the reachability information of neighbors similar to those of iSDX. Therefore, the total number of forwarding rules is the same in both systems. DESI installs forwarding rules on the network device of members, while iSDX does this in IXP fabric. Fig. 14(a) depicts the number of forwarding table entries for each device in IXP for DESI and iSDX. DESI requires on average 500x less table space than iSDX for an IXP with 500 members. For example, iSDX installs 65 250 forwarding rules for 500 participants on the IXP fabric while DESI does not install any rule on the IXP fabric but it installs $\approx$ 131 on each member device for the same data-trace.

Update latency. The iSDX controller is in charge of handling all BGP updates whereas in DESI the corresponding controller of the member takes care of them. Fig. 14(b) shows the update latency of iSDX and DESI in response to BGP updates. Similar to forwarding table entries, DESI drastically has lower latency in response to the BGP updates because the corresponding participant controller performs the OpenFlow rule update in the flow table entry independently.

Memory cost. iSDX uses four forwarding tables to encode the reachability information using the VMAC address, but DESI needs two tables for the same purpose using the metadata registry of OpenFlow. Fig. 14(c) presents that DESI requires drastically less memory space to encode the reachability information because it uses two tables with less number of match fields. Additionally, DESI keeps the IXP fabric unchanged.

The ARP overhead. iSDX uses the ARP message to update the virtual MAC address of the participant if the forwarding behavior of that participant changes. The VMAC address rewriting adds an extra overhead of ARP messages to the IXP fabric. The DESI does not introduce extra ARP overhead in such scenarios because the IXP fabric remains unchanged.

IX. DISCUSSION

Migration. DESI allows the IXPs and their members to gradually migrate to a fully SDX-based architecture using the

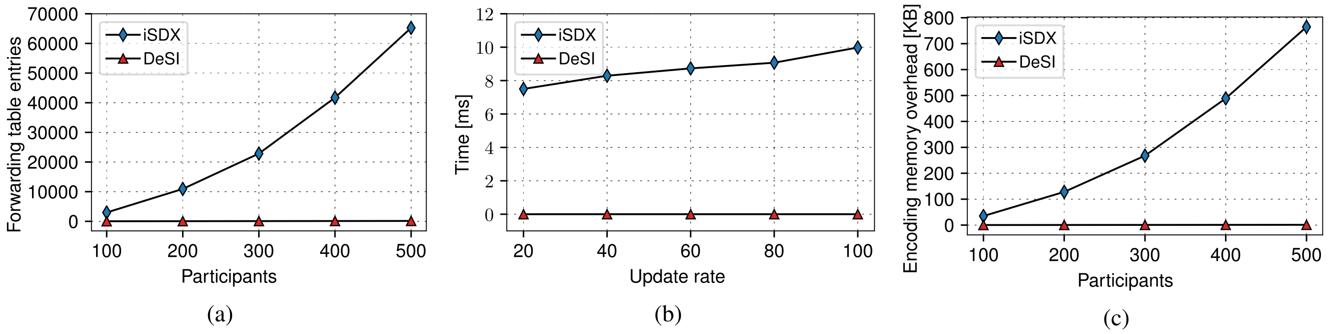


Fig. 14. The performance DeSI vs. iSDX. (a) The number of forwarding table entries on each OF device. (b) Latency to response for BGP updates. (c) Memory used to encode the reachability information of neighbors on each OF device.

decentralized architecture. We did a test in this regard by establishing a BGP peering among the legacy BGP router with an SDN controller of a member and confirm that the two BGP speakers can exchange the BGP packets. Therefore, IXP and its members can leverage the advantages offered by DeSI for the migration.

Memory Space. DeSI leverages the metadata registry offered by the OpenFlow to encode the reachability information of the members. This metadata field uses 64 bits of memory, while iSDX exploits VMAC address with the size of 48 bits for encoding the reachability information. We can use VMAC in DeSI to even decrease the memory usage of the devices, but this choice forces us to rewrite the MAC address of the packets. As our architecture drastically reduces memory usage by distributing the forwarding rules among the members' devices, thus the tradeoff of memory space between the metadata registry and VMAC field is negligible.

Routing correctness. Currently, DeSI relies on the network administrator for the correctness of routing policies. However, individual controllers of the members can install the forwarding rules in such a way that the traffic is deflected away from the default BGP best path for a specific prefix. This situation can lead to a persistent traffic loop [26]. We are studying strategies and countermeasures, mainly based on the extensions of the Gao-Rexford conditions [17] to avoid such a possible issue. We leave this part as future work.

X. Conclusion

In this paper, we present DeSI that is a decentralized SDN-based architecture for IXPs. DeSI guarantees the privacy of routing policies of ISPs while opening to the possibility of overriding the standard traffic forwarding of BGP using the members' SDN-controller. Also, DeSI leaves the IXP's switch fabric as is. DeSI scales for the IXPs with a large number of networks due to its design which requires no changes from the IXP fabric side. It also introduces an expressive policy language for the members to configure their routing policies. Furthermore, it gives the possibility of slowly migrating to the SDN-enabled IXPs. We are interested in exploring the possibility of extending DeSI with novel data-plane programmability paradigms, such as P4 [27].

References

[1] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazieres, "Replication, history, and grafting in the Ori file system," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 151–166.

[2] Y. Shuai, M. Gorius, and T. Herfet, "Low-latency dynamic adaptive video streaming," in *Proc. Broadband Multimedia Syst. Broadcast.*, 2014, pp. 1–6.

[3] A. Gupta *et al.*, "SDX: A software defined internet exchange," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 551–562, 2015.

[4] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE Proc. IRE* , vol. 103, no. 1, pp. 14–76, Jan. 2015.

[5] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[6] M. Chiesa *et al.*, "Inter-domain networking innovation on steroids: Empowering IXPs with SDN capabilities," *IEEE Commun. Mag.*, vol. 54, no. 10, pp. 102–108, Oct. 2016.

[7] A. Gupta *et al.*, "An industrial-scale software defined internet exchange point," in *Proc. 13th USENIX Conf. Networked Syst. Des. Implementation, Ser.* Berkeley, CA, USA: USENIX Assoc., 2016, pp. 1–14.

[8] M. Chiesa, D. Demmler, M. Canini, M. Schapira, and T. Schneider, "SIXPACK: Securing internet exchange points against curious onlookers," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol.*, 2017, pp. 120–133.

[9] S. Hermans, A. Vijn, J. Schutrup, and J. Claassen, "On the feasibility of converting AMS-IX to an industrial-scale software defined internet exchange point," pp. 1–9, 2016. [Online]. Available: https://scripties.uba.uva.nl/download?fid=642434

[10] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN programming with pyretic," in *Proc. USENIX*, 2013, pp. 40–47.

[11] G. Antichi *et al.*, "Endeavour: A scalable SDN architecture for real-world IXPs," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2553–2562, Nov. 2017.

[12] M. Bruyère, "An outright open source approach for simple and pragmatic internet exchange," Ph.D. dissertation, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2016.

[13] M. Bruyere *et al.*, "Rethinking IXPs' architecture in the age of SDN," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2667–2674, Dec. 2018.

[14] "EXABGP," Mar. 2020. [Online]. Available: https://github.com/Exa-Networks/exabgp

[15] A. Dethise, M. Chiesa, and M. Canini, "Prelude: Ensuring inter-domain loop-freedom in SDN-enabled networks," in *Proc. 2nd Asia-Pacific Workshop Netw., Ser.*, 2018, pp. 50–56.

[16] M. Chiesa, R. di Lallo, G. Lospoto, H. Mostafaei, M. Rimondini, and G. Di Battista, "PrIXP: Preserving the privacy of routing policies at internet exchange points," in *Proc. IFIP/IEEE Symp. Integr. Netw. Serv. Manage.*, 2017, pp. 435–441.

[17] L. Gao and J. Rexford, "Stable internet routing without global coordination," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, pp. 681–692, Dec. 2001.

[18] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. Symp. SDN Res., Ser.*, 2016, pp. 1–12.

[19] Open networking foundation, “Openflow switch specification,” 2018. [Online]. Available: https://www.opennetworking.org/software-defined-standards/specifications/

[20] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, “Enabling fast failure recovery in openflow networks,” in *Proc. 8th Int. Workshop Des. Reliable Commun. Netw.*, Krakow, Poland, 2011, pp. 164–171.

[21] “RYU: Component-based software defined networking framework,” Jan. 2020. [Online]. Available: https://osrg.github.io/ryu/

[22] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, “Kathará: A container-based framework for implementing network function virtualization and software defined networks,” in *Proc. NOMS IEEE/IFIP Netw. Operations Manage. Symp.*, Apr. 2018, pp. 1–9.

[23] B. Rudolph, “An IXP route server test framework,” Apr 2016. [Online]. Available: https://www.de-cix.net/Files/c2571b938eebcf5d6cb11e-feff40fce8403ff07e/Benedikt-Rudolph—Route-Server-2.0-in-detail.pdf

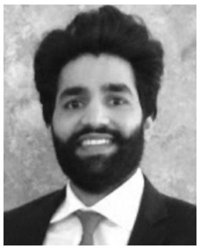
[24] “Follow up : AMS-IX route-server performance test euro-ix 20th,” Apr 2012. [Online]. Available: https://ripe64.ripe.net/presentations/49-Follow_Up_AMS-IX_route-server_test_Euro-IX_20th_RIPE64.pdf

[25] P. Richter, G. Smaragdakis, A. Feldmann, N. Chatzis, J. Boettger, and W. Willinger, “Peering at peerings: On the role of IXP route servers,” in *Proc. Conf. Internet Meas. Conf., Ser.*, 2014, pp. 31–44. [Online]. Available: https://doi.org/10.1145/2663716.2663757

[26] R. Birkner, A. Gupta, N. Feamster, and L. Vanbever, “SDX-based flexibility or internet correctness?: Pick two!” in *Proc. Symp. SDN Res., Ser.*, 2017, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/3050220.3050221

[27] P. Bosshart *et al.*, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

Habib Mostafaei received the Ph.D. degree in computer science and engineering from Roma Tre University, Rome, Italy, in 2019. He is currently a Postdoctoral Researcher with Technische Universität Berlin, Berlin, Germany, where he is involved in the BIFOLD-BBDC project. Before the Ph.D. education, he was a full-time Faculty Member with Computer Engineering Department, Azad University from 2009 to 2015. His main current research interests include networked systems, network measurements, and distributed systems.



Davinder Kumar received the master’s degree in 2018, defending a thesis on the applicability of SDN-based solutions in the Internet exchange Points. He is currently a Software Engineer of EGS-CC project in Darmstadt, Germany. His research focused on SDN-based policies specification in inter-domain routing.

Gabriele Lospoto received the Ph.D. degree in 2016 from Roma Tre University, Rome, Italy, with the thesis Improving flexibility, provisioning, and manageability in intra-domain networks. After two years as a Postdoc Research Fellow with Computer Networks Group, Roma Tre University, he moved in the industry and currently works in an Internet Service Provider. His research focuses on how to exploit software-defined networking in order to simplify service provisioning in production networks, especially in specific contexts such as federated networks. His research topics also include the study of formal methodologies to compare services implemented with different technologies.

Marco Chiesa received the Ph.D. degree in computer engineering from Roma Tre University, Rome, Italy, in 2014. He is currently an Assistant Professor with the KTH Royal Institute of Technology, Stockholm, Sweden. His research interests include Internet architectures and protocols, including aspects of network design, optimization, security, and privacy. He was the recipient of the IEEE William R. Bennett Prize in 2020, the IEEE ICNP Best Paper Award in 2013, and the IETF Applied Network Research Prize in 2012. He has been a Distinguished TPC Member at IEEE Infocom in 2019 and 2020 and he is currently co-chairing ACM Conext 2021.

Giuseppe Di Battista received the Ph.D. degree in computer science from the Sapienza University of Rome, Rome, Italy. He is currently a Faculty Member with the Department of Engineering, Roma Tre University, Rome, Italy. He is also a Professor of computer science. He has authored or coauthored more than 200 papers in his research filed and has given several invited lectures worldwide. His current research interests include computer networks, graph drawing, and information visualization. His research has been funded by the Italian National Research Council, by the EU, and by several industrial sponsors. He is national and/or local Coordinator of many Projects of Relevant Italian National Interest of the MIUR.