PGA: Using Graphs to Express and Automatically Reconcile Network Policies

Chaithan Prakash[^*] Jeongkeun Lee[†] Yoshio Turner[°*] Joon-Myung Kang[†] Aditya Akella[^]
Sujata Banerjee[†] Charles Clark[‡] Yadi Ma[†] Puneet Sharma[†] Ying Zhang[†]
[^]University of Wisconsin-Madison, [†]HP Labs, [°]Banyan, [‡]HP Networking

ABSTRACT

Software Defined Networking (SDN) and cloud automation enable a large number of diverse parties (network operators, application admins, tenants/end-users) and control programs (SDN Apps, network services) to generate network policies independently and dynamically. Yet existing policy abstractions and frameworks do not support natural expression and automatic composition of high-level policies from diverse sources. We tackle the open problem of automatic, correct and fast composition of multiple independently specified network policies. We first develop a high-level Policy Graph Abstraction (PGA) that allows network policies to be expressed simply and independently, and leverage the graph structure to detect and resolve policy conflicts efficiently. Besides supporting ACL policies, PGA also models and composes service chaining policies, i.e., the sequence of middleboxes to be traversed, by merging multiple service chain requirements into conflict-free composed chains. Our system validation using a large enterprise network policy dataset demonstrates practical composition times even for very large inputs, with only sub-millisecond runtime latencies.

CCS Concepts

•**Networks** → **Programming interfaces; Network management;** *Middle boxes / network appliances; Network domains; Network manageability; Programmable networks;* Data center networks;

Keywords

Policy graphs; Software-Defined Networks

[*]This work was performed while at HP Labs.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787506

1. INTRODUCTION

Computer networks, be they ISPs, enterprise, datacenter, campus or home networks, are governed by high-level policies derived from network-wide requirements. These network policies primarily relate to connectivity, security and performance, and dictate who can have access to what network resources. Further, policies can be static or dynamic (e.g., triggered). Traditionally, network admins translate high level network policies into low level network configuration commands and implement them on network devices, such as switches, routers and specialized network middleboxes (e.g., firewalls, proxies, etc.). The process is largely manual, often internalized by experienced network admins over time. In large organizations, multiple policy sub-domains exist (e.g., server admins, network engineers, DNS admins, different departments) that set their own policies to be applied to the network components they own or manage. Admins and users who share a network have to manually coordinate with each other and check that the growing set of policies do not conflict and match their individually planned high level policies when deployed together.

Given this current status of distributed network policy management, policy changes take a long time to plan and implement (often days to weeks) as careful semi-manual checking with all the relevant policy sub-domains is essential to maintain correctness and consistency. Even so, problems are typically detected only at runtime when users unexpectedly lose connectivity, security holes are exploited, or applications experience performance degradation.

And the situation can get worse as we progress towards more automated network infrastructures, where the number of entities that generate policies independently and dynamically will increase manyfold. Examples include SDN applications in enterprise networks, tenants/users of virtualized cloud infrastructures, and Network Functions Virtualization (NFV) environments, details in §2.1.

In all of these settings, it would be ideal to eagerly and automatically detect and resolve conflicts between individual policies, and compose them into a coherent conflict-free policy set, well before the policies are deployed on the physical infrastructure. Further, having a high level policy abstraction and decoupling the policy specification from the underlying physical infrastructure would significantly reduce the burden

on the admins/users/application developers of the network both in crafting and implementing their policies.

In this paper, we address the open problem of automatic, correct and fast composition of independently generated network policies. Our contributions are:

1. We present a new high-level model called Policy Graph Abstraction (PGA) for expressing network policies. PGA is a simple and intuitive graph-based abstraction for each sub-domain to separately express networking policy on endpoints, independent of underlying network infrastructure. PGA naturally incorporates network middleboxes and enables automatic, eager composition.
2. We develop algorithms to automatically and scalably compose multiple policy graphs. The composition maintains the individually specified invariants from each policy graph. It also systematically determines an appropriate service order when merging service chains.
3. We present the design and implementation of the PGA system. We leverage existing SDN programming languages (e.g., Pyretic) to represent middlebox functionality and analyze service chains. The system can compose over 20K ACL policies from a real policy dataset in under 600s, while incurring sub-millisecond latency for the first packet of a flow when running reactively.

In this paper, we focus on policies related to network endpoints and decouple underlying network state information from the PGA abstraction. PGA treats the underlying network as "one big switch", which in some cases may not reflect all the low level policy requirements: e.g., traffic engineering decisions regarding specific switches/routers/network path [32]. We also do not focus on run-time network state conflicts, which has been recently studied in [33, 39, 13]. Occasionally, there will be a need to consider lower layer network state in making correct overall network-wide policy decisions. We leave this interplay between the PGA abstraction layer and the physical infrastructure state for future work.

2. BACKGROUND

We present three scenarios where the distributed policy specification and composition problem arises, followed by a detailed example that demonstrates technical challenges.

2.1 Target Scenarios

Enterprise networks manage policies using the notion of network compartments, which are defined based on administrative domains (e.g., external access network, geographical site), application/service (e.g., DNS), network protocol (e.g., IPv6) or network technology (e.g., WLAN). There can be hundreds of compartments governed by special policies that are independently specified and managed; the policies generated by multiple compartments may be applied to the same set of network devices.

Cloud infrastructures: tenants want to have their virtual networks with their own policies. These policies at the virtualized network level are created without any knowledge of the underlying physical infrastructure, but must comply with the network operator policies at runtime. A simple change of operator policy may affect and conflict with thousands of existing tenant policies.

NFV service networks aim to virtualize the service functions that are today implemented on specialized hardware middleboxes [10, 11]. A key requirement is to provide service function chaining (SFC) in these environments, with no or minimal knowledge of the network infrastructure and middlebox internals. SFC can be specified as network policies where specific packets/flows have to strictly follow the chain of service functions. Multiple chains devised by different administrative entities may be deployed on the same network infrastructure; the logical chains need to be composed into one coherent service chain.

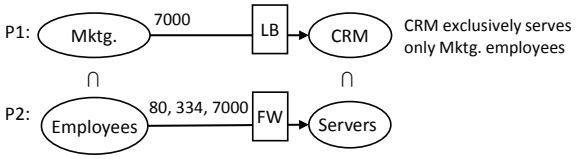
2.2 Challenges in Policy Composition

We use an example to illustrate the challenges that arise in composing policies in the above scenarios. Consider the two policies depicted graphically in Fig.1(a), inspired by a real enterprise scenario. Suppose a company's marketing department wants to deploy a CRM (Customer Relationship Management) application on some of the company's servers. The CRM admin specifies a network policy, called P1, that allows only marketing employees to send traffic to the CRM servers; the traffic must use TCP port 7000 and must pass through a load balancing service (LB).

Independently, the company-wide network admin specifies another policy, called P2, that restricts company employees to access company servers only through TCP ports 80, 334, and 7000; and the traffic must pass through a firewall service (FW). Note that Marketing employees are a subset of all employees, just as CRM servers are a subset of all the company servers as indicated in Fig. 1(a) by the *subset* symbol $\subset$. These independently specified policies need to be combined into a coherent composed policy that respects the intent of both stakeholders.

Correctly composing P1 and P2 is actually not that simple with currently available tools and languages. First, P1 and P2 contain two different types of policies: access control whitelisting (ACL) and network service chaining (FW, LB). Since the src/dst and port range of P1 are completely encompassed by those in P2, one may naively compose the ACL policies by prioritizing P1 over P2, but this incorrectly allows non-Marketing traffic to reach CRM servers. In addition, assuming the intended order of the service chain is FW followed by LB, the intent needs to be factored in for [Mktg$\rightarrow$CRM] traffic.

Network programming frameworks such as Merlin [38] and GBP [6] are not suitable for independent policy specification. GBP does not support composition and so a single combined program needs to be written manually. Merlin supports distributed but not fully independent policy specification. The network admin needs to explicitly delegate a policy to the CRM admin. The CRM admin is only allowed to modify that delegated policy in restricted ways, and in do-



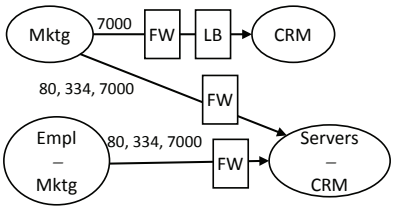
(a) Independently specified policies.

```
if_(match(srcip=Mktg, tcp, dstport=7000, dstip=CRM),
   FW>>LB>>route,
   if_(match(dstip=CRM), drop,
      if_(match(srcip=Empl, tcp, dstport=80|334|7000, dstip=Servers),
         FW>>route, drop)))
```

(b) Pyretic-style composite program.

```
Subject1: CRM-Access
     tcp, dstport =7000 : Permit, FW-LB chain
Subject2: CRM-Block
     * : Deny
Subject3: Server-Access
     tcp, dstport=80|334|7000 : Permit, FW chain
Clauses (Prioritized):
1. Mktg --> CRM : CRM-Access
2. * --> CRM : CRM-Block
3. Employees --> Server : Server-Access
```

(c) composite policy specified in GBP.



(d) Composed policy in a graph.

Figure 1: Policy composition example.

ing so must be exposed to and take into account the network admin's intent.

Other frameworks such as Frenetic [20] and Pyretic [34] allow users to compose modular policies/programs into a more complex control program. For example, Pyretic users can cause two policies to be sequentially applied to an incoming packet using the $>>$ operator, and this is effective to chain multiple service functions (such as FW$>>$LB). The sequential operator cannot be used to directly compose P1 and P2: e.g., P1$>>$P2 composition fails to allow [Empl$\rightarrow$ Servers$-$CRM] traffic since the traffic cannot pass P1's ACL. Pyretic's parallel composition, P1$+$P2, will apply each policy to a different copy of the same packet and fail to block non-Marketing traffic and to create the FW$>>$LB chain.

From these failed attempts to combine P1 and P2, we see that a correct composition requires carefully **decomposing** each of P1 and P2 into ACL and service requirements and **recomposing** them into a single program. In particular, Pyretic supports an $if\_(match(), A, B)$ statement for 'if match() do A, else B'. With this, users can write a composite program implementing P1 and P2, as shown in Fig.1(b). However, the user has to carefully consider the flow space relations (P1 $\subset$ P2) and manually compose the FW$>>$LB chain for [Mktg$\rightarrow$CRM], place P1's match() classifier followed by P2, and insert $if\_(match(dstip = CRM), drop, ...)$ in between to implement exclusive access to CRM. A similar manually composed program for GBP is shown in Fig.1(c).

Such manual decomposition and re-composition process is possible when done by a human operator who clearly understands the **joint intent** of P1 and P2. Based on such understanding, the operator can 1) resolve, the ACL conflict between P1 and P2 and 2) decide, the order between FW and LB. P1 and P2 ACL policies do conflict since P1 blocks traffic from non-Mktg employees to CRM while P2 allows the traffic (by allowing its super-set). The joint intent used to resolve this conflict would be *P1's exclusive access policy overrides P2's allow policy*. Similarly, the order of the FW-LB chain is chosen using the operator's internal knowledge of the service functions.

Even if such joint intents are clear to the human operator, it is impractical and error-prone to manually compose thousands of real world policies that have more complex super/sub-set relations and access control requirements: e.g., exclusive access to source/destination, conditional on other attributes such as location and security level.

Thus, automated composition by the system, not by a human, is critical to build a practical and scalable policy framework. The key to enabling automatic composition is to *explicitly capture the internal intents of the individual policy writer in each policy*. Existing policy abstractions [19, 23, 38] do not support this. For instance, they cannot express the intent that an Allow ACL rule MUST allow its specified traffic; thus, the Allow rule can be overridden by a Deny rule from another policy it is combined with. Similarly, existing service chain policy work [38, 31] can only capture the intent that certain service functions should be deployed on the specified path, but they cannot capture the service functions/actions that MUST NOT be applied by other policies. Hence, a human oracle is required to manually combine service function requirements from different policies.

Fig.1(d) shows a correct composition of P1 and P2 in a simple graph. Only Mktg. employees can send to the CRM servers through the FW-LB service chain. The other servers except for CRM, expressed by a primitive set operator *diff* ('$-$'), accept traffic from all Employee devices including Mktg. An entire policy for any endpoint pair is expressed on the edge between two nodes representing the endpoints, with no need to carefully walk through multiple lines of prioritized rules or if_then statements. We show how to automatically create such composed policies in §4 and §5. We first lay out the required properties for a policy framework.

2.3 Requirements for policy framework

Simple and intuitive: Anecdotally, we found that many network admins and cloud tenants design their policies by

drawing diagrams on whiteboards. We believe a policy abstraction must be as simple as drawing diagrams similar to Fig.1(a), yet **expressive** enough to capture their intents for diverse and dynamic SDN, cloud and NFV applications with sophisticated service chain requirements [34, 22, 38].

Independent and Composable: Each policy writer should be able to write policies independently without coordinating with other policy writers; yet ensuring that their intents are being composed and enforced correctly, or receiving notification if there are conflicts or more information is needed.

Eager composition: Composing policies and ensuring that individual policy intents are satisfied prior to deployment, i.e., eagerly, is highly desirable. Eager composition can greatly reduce the number of conflicts/errors that a runtime system has to handle, enable speedy runtime operation, and reduce the chance of system misbehavior compared to lazy composition. However, eager composition without actual endpoints in the system can lead to exponential state explosion because in the worst case, every combination of input policies should be considered. The policy framework design should enable fast composition.

Automated: The policy framework must be highly automated to free network admins from manual and error-prone policy composition. In some cases when the system cannot identify the best policy composition, a human may be required to provide input and pick one composition. However, this is much less burdensome than the existing approach of manual composition.

Well-formed: The composed policy generated from the input policies should be well-formed such that a unique policy can be chosen without ambiguity for any given packet and associated dynamic conditions. This would allow the runtime operation to be deterministic.

Service chain analysis: Handling service chains for correct policy composition is crucial: e.g., a misplaced FW can drop packets that are legitimately allowed by ACL policies. To the extent possible, the policy framework needs to model the behavior of service functions for composition analysis.

Our framework, PGA, provides a simple and intuitive *graphical interface* that is similar to how network admins typically visualize their policies on a whiteboard. In PGA, each user writes policies for arbitrary selection of endpoints based on *logical endpoint properties* that makes sense to them. This gives users the flexibility to write policies independent of each other as well as the underlying physical network infrastructure. PGA achieves automated, eager composition of such policies by capturing the *relationship* between endpoint properties and enabling individual policies to *constrain* each other during composition. Lastly, it provides abstractions to succinctly *model middlebox behavior* that aids in service chain analysis.

3. SYSTEM OVERVIEW

Fig. 2 provides an overview of the PGA system components and their interactions with external components. As mentioned earlier, PGA uses a graph-based abstraction to

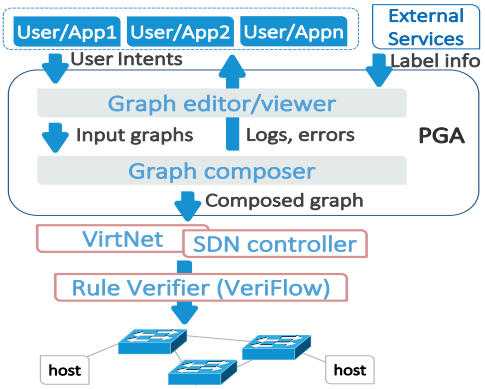


Figure 2: PGA system architecture.

specify network policies. The users/tenants/admins and SDN applications independently generate their policies as graphs and submit them to the Graph Composer through a PGA User Interface (UI). The UI and composer utilize additional information (e.g., tenant hierarchies, tenant/endpoint locations) from external sources to assist in the policy specification and eager composition. The composer automatically composes input graphs into a combined conflict-free graph, resolving or flagging conflicts/errors and reporting them to users, possibly with suggested fixes.

The composed high-level policy can be compiled down to low-level configurations/rules, either proactively or reactively; PGA's eager policy composition is orthogonal to the lower-level compilation methodology. Our prototype supports two different network environments. The first is an SDN environment with OpenFlow enabled network devices, where we use the POX OpenFlow controller to reactively generate OpenFlow rules for pkt-in events based on a composed policy graph. To guard against possible bugs in rule generation, and against the possibility that unrelated SDN modules, e.g., traffic engineering, may generate conflicting rules, we use a rule verification tool (§6) to detect and verify changes in end-to-end communication paths against the ACL policies that PGA has generated for recent pkt-in events.

Our prototype also supports a virtual network abstraction provided by OpenStack Neutron [8]. Our PGA service proactively configures virtual network resources – specifically, Neutron Security Groups and service functions – to implement the polices of a composed graph, and dynamically associates VMs to the best-matching graph nodes at runtime.

Both the SDN and virtual network systems can be extended to support NFV; e.g., PGA's composed graph can be used to generate Network Service Headers (NSH) and Service Classifiers [4] for service function chaining.

For simplicity, our discussion around prototype system will assume the SDN-OpenFlow environment while the graph model and composition algorithm below can be applied to diverse environments.

4. GRAPH MODEL

In PGA, network policies are described using a *graph structure* that represents: 1) allowed communication between

network endpoints, and 2) any required service function chain traversal for each communication. PGA is designed so that concise models that contain only a small number of elements are able to express policy for a much larger number of endpoints. PGA achieves this model scalability by describing policies at the granularity of *groups* of endpoints (similar to [6]) that share common properties expressed in terms of primitives called *labels*. Labels can be assigned and changed at runtime as endpoint properties (states) change, enabling a static PGA graph model to capture all the policies that can be dynamically assigned to an endpoint. PGA is a whitelisting model; communication must be explicitly allowed by a PGA model, else it is implicitly denied.

We use the example graph models shown in Fig. 3 to illustrate various features of the PGA model and composition. Model (a) is given by an admin for all Departments of a company and specifies that traffic is allowed from the IT department to the Engineering department using specified protocol port numbers. Model (b) is given by a web application admin. This model allows traffic from any Department to access the Web application using port 80 (`HTTP` protocol) and indicates that the traffic will be load balanced. The model also specifies that traffic is allowed from the Web to DB tiers, and from DB to itself. Model (c) is specified by an SDN application for DNS-based security protection. This model requires that DNS traffic from network endpoints with "Normal" security status must be inspected by a Deep Packet Inspection service when it performs DNS lookups, while network endpoints that are deemed "Quarantined" can only send their traffic (of any type) to a Security Remediation server. Model (d) is actually two models given by the Data Center admin. The first model specifies that traffic coming into the Data Center from the Campus must pass through a Firewall service and a Byte Counter service, while all east-west traffic within the data center also traverses a Byte Counter service. The second model allows monitoring traffic (port 9099) between endpoints within the Data Center.

PGA automatically combines multiple independently specified policy graphs into a coherent composed policy. This requires the novel ability to merge service functions chains in addition to access control policies. The PGA model has three primitives that support composition.

1. The *packet processing behavior* of each network service function in a service function chain is explicitly specified using a variant of the Pyretic network programming language [34] that we developed. Our composition engine (§5) analyzes these descriptions in order to automatically assemble composed service function chains that correctly combine policies from multiple policy graphs.

2. A *label mapping* input is introduced to enable identification of endpoint groups that can have overlapping endpoint membership. This avoids unnecessarily composing endpoint groups that are mutually exclusive, thus greatly reducing computation time and memory requirements. Label mapping is necessary to detect overlapping membership not only between endpoint groups of different graphs but also within a single graph.

3. *Composition constraints* are introduced to give policy writers the flexibility to express invariants that can never be violated under composition. Each policy writer can independently express their intended invariants, and the PGA system will automatically compose these individual policies while respecting the invariants. This avoids imposing rigid universally applied conflict resolution policies (such as static priority), and minimizes the need for human intervention during composition.

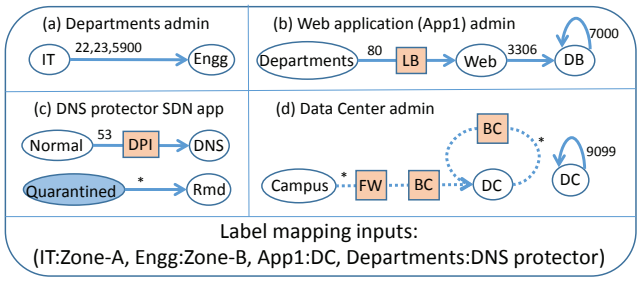


Figure 3: Sample input graphs and label mapping.

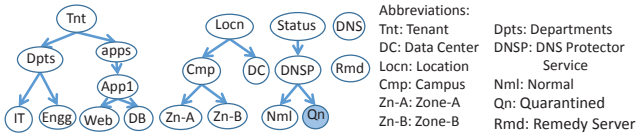


Figure 4: Sample input label namespace hierarchy.

We next describe the basic PGA graph constructs, followed by the primitives that support composition.

4.1 Graph Constructs

Vertices and Labels: Each vertex in a PGA graph model represents an *endpoint group* (EPG) which comprises a set of *endpoints* (EPs). An EP is the smallest unit of abstraction for which a policy is applied, e.g., a server, VM, client device, network, subnet, or end-user. An EPG comprises all EPs that satisfy a *membership predicate* specified for the EPG. In Fig. 3, each membership predicate is given as a *label*, e.g., Web, DC, etc. In general, a membership predicate can be a boolean expression over all labels.

Fig. 4 shows an example set of labels arranged in a hierarchy. The labels at the leaves, e.g., IT, Engg, Web, etc. are the truly basic elements, while each non-leaf label is a composite label that is simply a convenient shorthand for the logical disjunction (boolean OR) of all of its descendant leaf labels, e.g., Dpts is equivalent to IT OR Engg. Though not shown in the figure, composite labels can be defined that do not fit into a hierarchy, and can in general represent shorthand notation for arbitrary expressions over leaf labels. The hierarchy serves another important purpose for PGA composition. As we describe in §5, the composition process translates input graphs into a normalized form in which all EPGs have disjoint membership. PGA therefore needs to know which labels are mutually exclusive, i.e., cannot ever be assigned

simultaneously to an EP. For example, in the DNS protector app of model (c), an EP cannot be both Normal and Quarantined. The hierarchy provides this information. Specifically, in any single tree of the hierarchy, any set of labels that do not have an ancestor relationship are mutually exclusive, e.g., {Zn-A, Zn-B}, {Cmp, DC}, and {Dpts, App1}.

The PGA system can restrict the scope of each policy writer to a particular relevant subset of the label space; for example, the admin of the Campus Network might only be allowed to use labels Cmp, Zn-A, and Zn-B for defining EPGs in its policy graphs. Each leaf label represents a collection of boolean variables, one per EP, i.e., leaf label x represents the set of boolean variables $\{e.x|e \in E\}$, where E is the set of all EPs. Assigning a leaf label to an EP sets the value of the corresponding boolean variable to True, otherwise it is False. Labels can be split into three categories: 'tenant' labels identifying end-users or their applications, e.g., IT or Web, 'network location' labels identifying regions of the network topology, e.g., Zn-A or DC, and 'status' labels indicating dynamically changing properties, e.g., Qn for currently quarantined EPs. To illustrate how labels are used, say that server S is an EP that is located in the Data Center and hosts the database of Web Application of Fig. 3. Then, server S would be assigned labels DC and DB, setting its boolean variables $S.DC$ and $S.DB$ to True.

EPs can be assigned labels dynamically at runtime, causing them to move from one EPG to another. For example, a server that was assigned the label Nml (normal) could subsequently be relabeled Qn (quarantined) when a network monitor detects the server issuing a DNS query for a known malicious Internet domain. Thus, a *static* PGA graph model actually describes a set of network policies that are applied *dynamically* to each EP according to the EP's status changes over time (that can be programmed as a finite state machine [28]). Moreover, analysis and composition of policy graphs into a fully composed network policy graph is a procedure that only needs to be invoked when policy graphs are added, modified, or removed. Analysis is not needed when EPs change EPG membership. Instead, the runtime system only needs to perform the lightweight operation of looking up and applying the correct rules for each EP depending on its current EPG membership. As EPs change membership across EPGs, the set of addresses encompassed within an EPG also changes. In general, an EPG can be associated with a variable representing virtual addresses indicating 'some EP' within the EPG – e.g., virtual IPs used for server load-balancing – and policies can be written using the EPG variable as well.

Edges and Service Chains: As stated earlier, PGA is a whitelisting model; by default, no communication is allowed between any EPs. A directed edge between EPGs is required to specify allowed communication in a PGA policy. An edge consists of a classifier, which matches packet header fields to represent the security whitelisting rule. Classifiers may include virtual addresses described above. It also optionally has a service chain consisting of a sequence of one or more *network function boxes*. A network function box may correspond to an SDN controller function or to a network middlebox or a set of middleboxes. A directed edge from endpoint group E to endpoint group E' with Boolean predicate (classifier) B and path expression (service chain) P indicates that a correct implementation will forward traffic from all hosts in E satisfying B along paths that are a prefix of the concatenation of the network function boxes in P and some host in E'. For example, model (c) in Fig. 3 specifies that traffic sent on port 53 is allowed from Normal (Nml) EPs to DNS server EPs and the packets must pass through a DPI network service. The implementation should also take care of forwarding to the right destination EP which might be decided at the source EP or at a network function box (e.g. load balancer). In the degenerate case where E and E' are singletons, B is the trivial predicate that always returns true, and P is the empty path expression, an edge reduces to a specification of point-to-point forwarding.

A model can have multiple directed edges from an EPG, as long as the edge classifiers cover non-overlapping flow spaces. Edge whitelist rules are stateful, such that the reverse traffic on established connections (e.g. TCP) is also allowed. Two types of edges can be specified. A *whitelist edge* is depicted as a solid line and describes an allowed communication, e.g., from IT to Engg in Fig. 3. A *conditional edge* is depicted as a dotted line and specifies a service chain requirement which is instantiated if and only if the edge's match condition overlaps the classifier of a whitelist edge in a *different* policy graph. A conditional edge, by itself, does not allow communication. Fig. 3(d) illustrates the utility of conditional edges. The Data Center admin's intent is not to allow all communication from Campus to DC but to have any communication allowed by any other graph pass through the specified service chain. If no policy graph has a whitelist edge from Campus to DC, then the service chain requirement is not needed in the fully composed policy graph.

4.2 Primitives Supporting Composition

Network Function Box Behavior: To enable automatic composition of service function chains, PGA models need to specify the packet processing behavior of each network function box. For this, PGA uses the open source Pyretic SDN language [34], and extends it to enable programming based on EPGs. This allows a user to specify the full behavior of a network function box as a Pyretic program; we term this *white boxing*. Whiteboxing is not a hard requirement. In cases where middleboxes are used for which the internal logic is not known, we support *gray boxing*. Herein, the corresponding network function box is described by a Pyretic program that captures only the high-level bounding behavior of the middlebox. For example, a commercial L7-aware load balancer can be gray boxed as `match(dstip = Web.virtIP) » modify(dstip = Web.RIPs)`, where `Web.RIPs` is a set of real IP addresses of destination web server EPs and `Web.virtIP` is the exposed virtual address of the web service. While the precise mapping of an input packet to a destination IP address is not known, the model accurately bounds the output header packet space of the middlebox.

We note that whiteboxing and grayboxing may not cover every possible NFV scenario. The current PGA framework does not consider network functions that duplicate and forward packets along different paths; this also implies that a network function box is allowed to modify the destination address to only hosts within the destination EPG of the edge that the box belongs to. Extending PGA to handle those cases is our future work.

Label Mapping: Policy composition combines the EPGs of one graph with EPGs of a second graph, potentially causing exponential growth in the size of the composed graph as the number of graphs to be composed increases. Usually, however, the vast majority of EPG combinations are mutually exclusive, i.e., it is impossible for any EP to belong to both of the original EPGs that are being combined. For example, if all IT dept. EPs are always located in the datacenter and never in the campus network, then a combined EPG that corresponds to EPs that are both in the IT dept. and in the campus network is guaranteed to be the empty set and thus does not need to be generated in the composed policy graph. The *label mapping* input to PGA enables composition to avoid generating such impossible EPG combinations by capturing the relationship between labels across different label trees (e.g. tenant and location label trees). The label mapping is a symmetric relation $F \subseteq L \times L$, where L is the set of all leaf labels. If labels $(x, y) \in F$, then a single EP can be assigned both labels x and y. For example, the label mapping shown in Fig. 3 indicates that an IT EP can also be a Zone-A EP, i.e., the IT dept. is located in the network Zone-A. For simplicity, the syntax used to specify the label mapping can give only one side of the symmetric relation. Also, it can use non-leaf labels as a shorthand for all of the leaf label descendants. The detailed use of label mapping to limit composition complexity is covered in §5.

Composition Constraints: A policy graph can flexibly specify constraints on the policy changes that are allowed when the policy graph is to be composed with any other policy graph. Constraints can be specified for any ordered pair of EPGs in a policy graph. The constraints can limit the addition of new classifiers that would allow additional traffic from source EPG to destination EPG. For example, a policy that strictly allows only port 80 traffic from a source EPG to a destination EPG can use constraints to prohibit additional ports from ever being allowed even if they are allowed for the same EPG pair by whitelists in other policy graphs. Classifier constraints can also limit the removal of allowed traffic as a result of composition.

Regarding service chain composition, the constraints can limit the behavior of function boxes that are added to service chains when the policy graph is composed with another graph. Specifically, the constraints can place limits on the packet header field modifications and packet drop operations that additional function boxes can perform on packets. Since the behavior of function boxes is modeled using our extended Pyretic, composition analysis can check whether adding a specific function box to a given service chain would violate the constraints given by the policy graphs that are being composed together.

	Classifier		Function Box	
Match	Add	Remove	Drop	Modify
port 80	Y	N	N	DSCP=16,18,20
port 88	N			
*	Y			

Table 1: Example composition constraints.

Table 1 shows example composition constraints from a source EPG to a destination EPG. The table indicates that port 80 traffic cannot be disallowed through composition with other graphs, and function boxes that are added cannot drop packets but are allowed to modify the DSCP packet field to a set of specific values. The table also shows that port 88 cannot be allowed through composition, but otherwise all other traffic can be allowed, with no restriction on function boxes. Composition constraints could more generally be specified using a constraint language such as Prolog.

A commonly needed special case for composition constraints occurs when a policy graph specifies that no additional traffic should be allowed to or from a particular EPG. Composition constraints can express that, i.e., in a policy graph, all EPG ordered pairs that contain the EPG have the constraint that composition is prohibited from changing the range of allowed traffic or the service chain. We can more conveniently and concisely represent this set of constraints by marking the EPG as "exclusive". For example, the Qn EPG in Fig. 3 is an exclusive EPG, preventing other policies from thwarting the intention of the policy writer to redirect all traffic from quarantined hosts to a remediation server.

5. GRAPH COMPOSITION

A key goal of PGA is to enable policy writers to specify their policies independently and delegate the composition process to the system. The system should compute the union of all policies from the input graphs, subject to composition constraints, to generate a composed graph. The composed graph should be well-formed (§2), i.e., comprise purely mutually exclusive EPGs to allow the PGA runtime to determine the unique EPG for each EP, and then apply the associated network policies to the EP (or determine that the EP is not in any EPG and so no communication for it is allowed).

Composition in PGA is different from the union (parallel) and sequential composition operators used in NetKAT [12] and Pyretic [34]. Their union (parallel) operator applies each of the constituent policies to a different copy of the input packet and then computes the union of their output packets while the sequential operator applies the output of one policy as input to the other. In PGA, we compute the union of the policies themselves based on set theoretic Venn diagram analysis. A PGA policy in a simplified form is a combination of *match* + *action*: EPG labels and edge classifiers form the match space while edge types and service chains constitute the action part. Conceptually, PGA composition takes the union of the match spaces of the input graphs: inheriting actions from the input graphs for the non-overlapping match spaces, and combining actions for the overlapping (intersecting) match spaces subject to composition constraints. Note

that EPGs can have overlapping EP membership specified as arbitrary Boolean expressions over the label space.

We accomplish this composition in two steps. We first *normalize input graphs* (§5.1) by transforming their EPGs into an equivalent set of disjoint EPGs to easily identify the overlapping space and generate well-formed policies in the composed graph. We then *compute the union of the normalized graphs* (§5.2) by creating directed edges equivalent to the union of the original policies, except where doing so would violate the invariants given by composition constraints. The overlapping policies, identified by a common classifier for the same src-dst EPG pair in the normalized graphs, can require two service chains to be merged in the union. We use composition constraints to select the network function boxes to include in the combined service chain. To select an appropriate service order, PGA detects dependencies between function boxes based on modeling of their packet processing behaviors, and use the dependencies to determine valid orderings. Additionally, we detect possible remaining conflicts that should be flagged to policy writers.

5.1 Normalization of Input Graphs

In normalization, we compute a set of globally disjoint EPGs which represent the equivalence classes of endpoints to which the same set of policies should be applied. We then transform each input graph into an equivalent normalized form where policies are expressed only with respect to the newly computed EPGs.

The first step is to translate input graph EPGs into EPGs with globally disjoint membership. Second, composition constraints from the input graph must be replicated and merged to the normalized graph. Third, edge policies from the input graph must be replicated and merged to the normalized graph, subject to the composition constraints added in the previous step.

The abstractions of label hierarchy and label mapping facilitate the translation of input graph EPGs into globally disjoint EPGs in the normalized graph. As described in §4, label hierarchy captures the sets of mutually exclusive labels. Each EPG in the input graph is first split into locally disjoint EPGs by algebraically rewriting the EPG's membership predicate expression into an equivalent positive disjunctive normal form. Each term in the resulting expression describes a locally disjoint EPG. More specifically, we replace each composite label in the expression with its leaf label equivalents. Then, we expand the expression and remove any negated labels by replacing them with the disjunction of all the other sibling leaf labels in the hierarchy (since they are mutually exclusive). Finally, we convert the expression, which is now using only leaf labels in positive form, into disjunctive normal form, i.e., ORing a collection of terms where each term is the AND of leaf labels. Any conjunctive term that has any mutually exclusive labels must be the empty set, and so we delete these empty terms from the expression. Each remaining conjunctive term defines an EPG that is disjoint within the same input graph.

To obtain globally disjoint EPGs, we may need to further divide each locally disjoint EPG. We take each conjunctive term and check the label mapping to identify all other potentially related labels, i.e., labels that are transitively related to the labels in the term. For example, if the term is $label_1$ AND $label_2$, we check the label mapping starting from each of these labels and add the result to the conjunction. Suppose this adds $label_3$ AND $label_4$ AND $label_5$. If any of the resulting labels are mutually exclusive, e.g., Nml and Qn, we split the terms accordingly. For example, if $label_4$ and $label_5$ are mutually exclusive, we form two terms: 1) $label_1$ AND $label_2$ AND $label_3$ AND $label_4$, and 2) $label_1$ AND $label_2$ AND $label_3$ AND $label_5$. We continue splitting the terms until no term has mutually exclusive labels. This final set of terms corresponds to globally disjoint EPGs.

Once the normalized EPGs are generated, composition constraints from the input graph need to be replicated to the normalized graph. In particular, suppose that EPG S in the original input graph is translated to normalized EPGs S_1, S_2, ..., S_m, and input graph EPG D is normalized to EPGs D_1, D_2, ..., D_n. If the input graph has constraints for source EPG S and destination EPG D, then the constraints must be replicated in the normalized graph for ordered EPG pairs $(S_i, D_j), \forall i = 1..m, \forall j = 1..n$.

Composition constraints may need to be merged in addition to being replicated. If the original graph has constraints for EPGs (G, H), and these EPGs overlap with EPGs (S, D), then any constraint specified for (G, H) must be merged with the constraint for (S, D) in order to construct the constraints for the normalized EPGs that constitute the overlap. Merging composition constraints entails adopting the union of restrictive invariants of two overlapping constraints. However, conflicts may be detected if an invariant from one composition constraint, e.g., never allow port 80 traffic, is opposed by an invariant from another composition constraint, e.g., never deny port 80 traffic. Such conflicts are error conditions flagged to operators.

Finally, normalization replicates and merges edges from the original graph's EPG pairs to the normalized graph's EPG pairs to express equivalent policies. If EPG S has an edge to EPG D in the original graph, then the original edge is replicated to the normalized graph EPGs $(S_i, D_j), \forall i = 1..m, \forall j = 1..n$. Any edges for (G, H) need to be merged with those of (S, D) on the normalized EPG pairs that constitute the overlap. This merging is governed by the (merged) composition constraints.

5.2 Graph Union

In graph union, for each globally disjoint EPG that may be in multiple normalized graphs with different policies, we compute the union of those policies to obtain the final policy that should be applied to its endpoints. This involves merging whitelists and service chains from multiple edges without violating each other's constraints.

In normalized form, EPGs in two different graphs are either disjoint or equal. Partial overlap is not possible. This property enables multiple normalized graphs to be composed together using a simple union operation. The composed graph has the union of all the EPGs of the individual graphs. The union operation also copies and merges composition con-

straints and directed edges from the individual graphs to the composed graph. Merging composition constraints is performed identically as described in §5.1.

As edges from the individual graphs are added to the composed graph, they are checked against the composition constraints for the source and destination EPGs. This check determines whether an edge's classifier satisfies the constraints or needs to be narrowed to be in compliance with them. If a new edge passes this test with a non-null surviving classifier, then it may be added or merged with existing edges from source to destination EPGs.

The first step in merging the new edge is to find the intersection of its classifier flow space with the existing classifiers. For the non-intersecting classifier space of the new edge, the new edge and its function boxes can be added directly, subject to function box composition constraints. For any intersecting classifier space, merging is needed. We break down the intersecting space into a matching set of subspaces in the existing policies and for the new edge. This allows us to merge, for each subspace, one existing edge with one new edge.

For each pair of edges that need to be merged, if either edge has a service chain for the intersecting space, then service function composition is required that combines function boxes from the service chains of both edges. An important challenge is to determine a proper ordering of the service functions in the composed chain. PGA determines the order by analyzing function box actions on different flow spaces. The analysis identifies input-output dependencies between different function boxes and constructs an ordering that is consistent with the dependencies. In addition, the analysis identifies potential function box conflicts in which two boxes perform apparently incompatible operations on a common packet space. For example, if one function box drops all packets in a packet space for which the other function box has a byte counting action, then there is likely a conflict. Finally, a set of heuristics is used to further improve ordering after satisfying detected dependencies and flagging possible conflicts.

The key to enabling PGA to compose and merge service function chains automatically is that the packet processing behavior of function boxes is explicitly exposed to the PGA system for analysis. PGA models network function *white boxes* and *gray boxes* behavior expressed explicitly in an extended version of Pyretic. The Pyretic compiler can take these descriptions and convert each network function box's behavior into a set of prioritized *match-action* rules. PGA analyzes these rules to characterize the *In Packet Space* and an *Out Packet Space* for each rule of every service function. The In Packet Space of a rule is defined as the flow space that would match and thus be processed by the rule, while the Out Packet Space is the outcome of processing the In Packet Space by the rule.

When merging two service function chains, PGA analyzes every pair of function boxes composed of one function box from each chain to identify a full dependency graph, and possible conflicts between different function boxes. For each function box pair, analysis considers every pair of *match-action* rules across the two service functions to find all dependencies and possible conflicts.

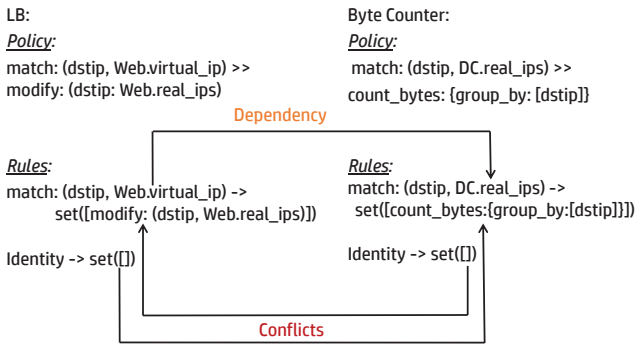


Figure 5: Analysis of service functions to determine order.

A dependency is identified from one rule to another if the first rule actively creates (through modify action) an Out Packet Space that overlaps with the second rule's In Packet Space. In Fig. 5 we show the Pyretic policies for two function boxes, Load Balancer (LB) and Byte Counter. Pyretic's compiler has converted these policies into prioritized match-action rules as shown below each Pyretic policy. In this example, as shown by the Dependency arrow, the In Packet Space of the Byte Counter is dependent on the Out Packet Space of LB, because the action of LB modifies the destination IP address to a real_ip address that is checked by the match portion of the Byte Counter's rule.

Possible conflicts between function boxes may be uncovered by the analysis and avoided through ordering or reported to users as possibly unresolvable conflicts. In Fig. 5, one possible conflict is between the default drop action in the Load Balancer and the byte counting in the Byte Counter, because for the same In Packet Space (destination IP address is not Web.virtual_ip), the two boxes have apparently incompatible actions (drop versus count). Another possible conflict is between the modify destination IP address action of the Load Balancer and the default packet drop action of Byte Counter. For the same In Packet Space (destination IP is Web.virtual_ip) the two actions are modify destination IP address versus drop, which are seemingly incompatible.

The final service order is determined using topological sort over the dependency graph. When there are more than one possible order, we choose an order that satisfies constraints on the input edges, and finally resort to heuristics, such as: a security service function, identified by its policy that actively drops packets based on some criteria, should be placed ahead of other service functions. This is consistent with how networks are typically administered.

For reference, the final composed graph obtained for our example from Fig. 3 is shown in Fig. 6. Note that a single function box from an input graph may be replicated across multiple edges in the composed graph (e.g. DPI). These are logically the same function box.

6. PROTOTYPE

Our prototype implementation contains ≈2.5K SLOC in Python, including the following Pyretic extensions.

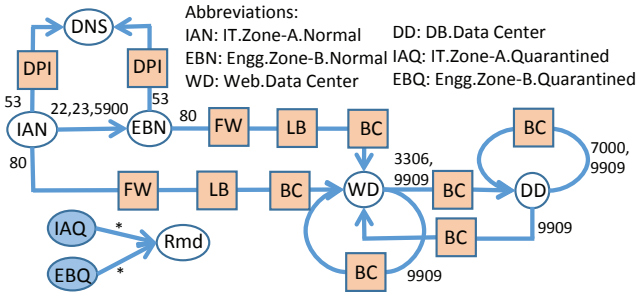


Figure 6: Composed graph for the running example.

6.1 Abstractions

To support policy graph specification, we extend Pyretic with three primitives: EPGs, Function boxes and Whitelists.

EPG. The user may create EPGs and specify membership requirement for endpoints to join the EPG through labels. EPGs also contain member variables indicating the set of L4 ports and endpoint IP addresses, which will be dynamically updated at runtime, and virtual IP addresses (used for load-balancing) of that EPG. The user may reference these variables while writing policies without having to assign values to them.

Function Box. The function boxes in our prototype are expressed in our extended Pyretic programming language. The policies are described in terms of the names of EPG variables, like the set of endpoint IPs and virtual IP, without having to assign values to the variables a priori. Both static and dynamic policies are supported as long as the dynamic policies contain an expression of their bounding behavior as described in §4. This expression is required for analyzing dependencies and conflicts between function boxes to determine their intermixed order when merging edges. The extended Pyretic language provides the necessary constructs to express this behavior.

WhiteList. The user may express Whitelists as arbitrary ranges of values for different flow header fields, e.g. `WhiteList(dstport=[80,>8000],proto=[6])`. Set operations – Union, Intersection and Difference – are supported to express more complex Whitelists.

6.2 System operation

As shown in Fig. 2, a *composer* module takes all the graphs and any auxiliary inputs and generates a composed graph by implementing the algorithm of §5. In our SDN prototype of the runtime system, the composed graph is stored as an in-memory hash table keyed with the source and destination EPGs for fast lookup of policies at runtime.

OpenFlow rule generation: In principle, the eagerly composed PGA policy could be enforced by proactively installing an equivalent set of OpenFlow rules into the network switches, determined using state of the art rule compilers [24, 36]. For simplicity, however, our runtime system implementation takes a fully reactive approach that evaluates the first packet of a new flow (≈ OpenFlow *pkt_in* event) against the composed policy graph and then installs rules to enforce the ACL and service chain policies for the new flow. For this, we modify the Pyretic runtime to query the in-memory graph upon a *pkt_in* event to lookup the ACL and service chain policies, which are then compiled down to OpenFlow v1.0 rules through the Pyretic compiler and POX controller.

The underlying network topology is run on mininet [2]. The prototype system installs flow rules on the edge switches (similar to ref. [29]). The switch/port that each endpoint (EP) is attached to is pulled out from mininet but can be given by external sources such as 802.1X. The path from source EP to destination EP is given by the Pyretic *mac_learner*; more sophisticated waypoint routing algorithms [38, 31, 26] can be used to support middleboxes and include non-edge switches for policy enforcement.

Runtime verification: The presence of service chains in a composed PGA graph complicates runtime policy verification. We leave it to future work; but verify only whether the ACL policies in the composed graph are correctly realized on the network, by using VeriFlow [27] in our prototype, against potential violations due to: 1) the policy-to-rule compiler having a bug in generating flow rules, or 2) there being another control module in charge of other types of policies (e.g., traffic engineering, switch firmware upgrade) that changes flow rules without involving PGA.

VeriFlow runs as a proxy intercepting OpenFlow messages between the controller and switches to detect three event types: reachability setup, blackhole (incomplete routing path) and forwarding loop. We modify VeriFlow to report detected events and affected flows to the PGA runtime, which caches prior pkt_in events and the ACL decisions made for them. PGA compares the reported VeriFlow event with the cached ACL decisions to verify the following: reachability should be set up for allowed flows and no path should be set up for ACL-denied flows. If the allowed path is not set up properly (blackhole, loop, or lack of reachability setup), it is a violation of PGA policy. Similarly, we raise an alarm if a reachability setup event is reported for a pair of EPs for which the composed PGA graph does not allow communication. Note that these VeriFlow verifications may not hold if the service functions in the path drop/add/change packets.

7. PGA IN ACTION

We demonstrate the power of PGA in conflict detection & resolution through two case studies on real world examples.

7.1 Conflict between SDN apps

We consider the following example inspired by a real world anecdote of installing two SDN apps on the same network: one is the QoS app in Fig. 7 and the other is the DNS filtering/protector app (Fig. 3). One way to handle the OpenFlow rules generated from the two apps is to compose them into one prioritized rule table to detect potential conflicts. An exclusive set of non-contiguous priority ranges was manually assigned to each app; e.g., the first and third highest ranges were given to the DNS protector while the second and the fourth highest ranges to the QoS app in one table snapshot. Since there is no global priority coordination across the apps, it is possible for the QoS app's rule that marks QoS bits

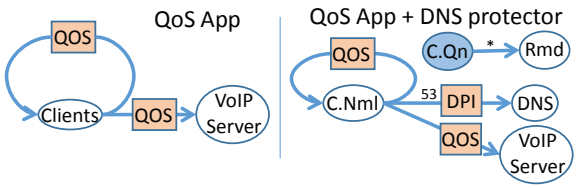


Figure 7: QoS app vs. DNS protector.

of VoIP packets and forwards them to output ports to be assigned a higher priority than one of the DNS protector rules that blocks quarantined IPs, potentially failing to block due to the prioritized QoS rule. This potential conflict between the two SDN apps was not initially detected because the two rules were written for non-overlapping IP addresses in the given snapshot of the rule table, although the two apps can be applied to the same client devices in other settings.

Another potential conflict was detected between the QoS rule and another DNS protector rule that redirects packets from DNS servers to different output ports. However, the detected conflict turned out to be a false positive as DNS servers were not supposed to be VoIP clients, and so the two rules actually don't overlap. These examples demonstrate the limitations of specifying and combining SDN apps at OpenFlow rule level, coupled with prioritization [24] and IP address assignment. PGA's label-based policy specification and graph composition enables eager conflict detection/resolution without requiring prioritized rules and IP addresses.

On the right of Fig. 7 is the composed PGA policy from the two apps. The EPG for quarantined clients (C.Qn) is set to 'exclusive'; this disallows any communication specified by the QoS app graph. And it is clear that DNS servers do not overlap with the client devices as there is no mapping between the two labels (omitted in the figure).

7.2 Large Enterprise ACL

Enterprise dataset: We obtain ACL policies from the policy management system in a large enterprise network. These policies are defined for each compartment, which are EPs that share the same set of ACL rules. The ACLs permit or deny the communications between the EPs within each compartment. Policies are written by network admins or application owners, manually reviewed for correctness and consistency, and then deployed in over 30+ global sites.

By analyzing the policies for the entire 136 compartments, we identified over 20K ACL policies that are written for 4916 EPGs. The sizes of EPGs vary, ranging from one IP address to over 600 non-contiguous subnets (100M IP addresses). We construct label mapping based on the super-sub set relationship between EPGs. Note that one EPG has multiple parents since the EPG may contain non-contiguous address blocks. We aggregate multiple policies for the same EPG pair by listing multiple protocol fields and port ranges for one edge, creating a total of 11,786 edges.

Redundancy and conflict are detected using PGA in this dataset. A rule is redundant if there exists another rule in the same compartment that completely covers the former rule's space with the same action. We have identified that 7% of the aggregated policies are redundant. PGA also detects two types of conflicts: *unmatched outgoing rules*, i.e., an outgoing rule in source compartment without the incoming rule to permit the communication in the destination compartment; and *unmatched incoming rules*, i.e., the incoming rule without the matching outgoing rule. The former is more damaging since it will result in blackholing. The latter means the incoming rules are not used since no traffic will be sent out. In total, we detect that 4.7% of outgoing policies and 4.5% of incoming policies are unmatched.

8. SYSTEM EVALUATION

For our experiments we used three data sets

D1: the synthetic running example from §4 and §5.

D2: the large enterprise dataset as described in §7.2.

D3: D2 with randomly added function boxes.

Our primary results show that due to eager composition, the runtime overhead of PGA is minimal even for very large composed graphs. PGA is practical and scalable, being able to analyze and compose thousands of policies producing nearly a million edges in under 600s when considering only ACLs and 800s in most cases for policy graphs with both ACLs and service chains. As described earlier, the composed graphs are correct by design. Additionally, for D1, we verify the correctness manually and for D2 and D3, we use Veriflow for indirect validation with reachability analysis.

The PGA prototype system is implemented in Python and currently runs as a single threaded program. For our evaluations, PGA is deployed on a server with 2x8 Intel Xeon 2.6 GHz cores with 132 GB memory running Linux kernel 3.13.0. We emulate switches and hosts with mininet 2.1.0 and openvswitch 2.0.2 on the server in order to create a topology and generate packets between hosts in different EPGs. Since the focus here is not to optimally assign flow rules to switches, the results described use simple topologies with one or two switches providing connectivity to the end hosts. We randomly generate packets between hosts to analyze reachability. A POX controller running on the same server uses PGA to look-up the policy to be applied upon receiving a *pkt_in* event. The input graphs are pre-composed by PGA and the composed graph is stored in memory. The composed graph from D1 has 8 EPGs and 11 edges. For D2, the composed graph has nearly 4K unique EPGs and over 76× the number of edges as the input graphs; in this case, high edge multiplication occurred in graph normalization because some input graph EPGs were defined by top-level labels in the label hierarchy.

Runtime Overhead. Fig. 8 shows the additional runtime latency incurred at the controller for the first packet of a flow by relying on PGA for determining the policy from the composed graphs of D1 (Small) D2 (Large). The input samples (pairs of EPGs) were randomly picked for D1. For D2, we picked more samples that had a large number of edges (10-30) between the EPG pair. This is because the prototype evaluates a packet linearly against each edge between the EPG pair until it finds the matching flow space and so

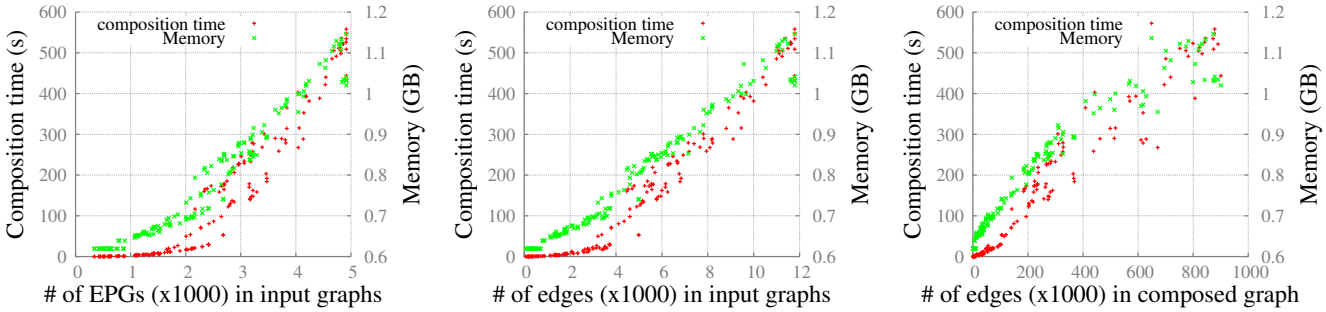


Figure 10: Scalability of PGA with D2.

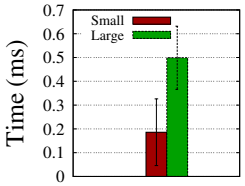


Figure 8: PGA induced additional runtime latency at the controller for D1 (Small), D2 (Large).

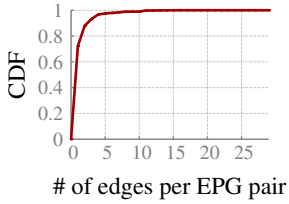


Figure 9: CDF of #edges per EPG pair in composed graph of D2.

incurs additional overhead for every edge; by selecting samples with the highest number of edges, we evaluate the worst case performance of PGA for this dataset.

Overall, the latencies are small for both cases. As expected, D2 results in a higher latency than D1 due to the higher number of edges.[1] Fig. 9 shows the CDF for number of edges per EPG pair in the composed graph of D2. 95% of EPG pairs have less than 4 edges between them and 99% below 11. Even the worst case performance numbers (obtained with >25 edges between EPG pair) have a sub-ms average latency overhead which is still small and practical for real time operation. We thus find that due to eager composition, the runtime overhead of policy lookup is negligible even for very large graphs. Additionally, the lookup time is independent of the network size, number of endpoints or the presence of service chains.

Scalability of PGA. Although PGA composes input graphs eagerly, it still needs to be able to handle very large inputs in a reasonable amount of time and with practical consumption of resources. We exercised PGA by varying the input from D2. We randomly selected different sets of compartments, composed their input graphs and measured the composition time as well as memory consumption. Fig. 10 shows the measures plotted against number of EPGs in input graphs, edges in input graphs, and edges in the composed graph. We omit the measures against number of EPGs in composed graph since it is similar to that for number of EPGs in input graphs. The largest input (entire D2) which produced nearly 1M edges in the composed graph required less than 1.2 GB of memory while completing in under 10 minutes.

If input graphs contain service chains, then the cost of composition is increased because pairwise analysis of function box interactions is required to determine correct function box orderings in the composed graph. Since D2 only contains ACL policies, we simulated a policy dataset with function boxes (D3) by first creating a pool of 16 synthetic function boxes, each having a bounding behavior that compiles to two match-action rules. We then randomly picked input edges from D2 to which we added a randomly chosen function box from the pool; up to 3500 functions boxes were added to D2, larger than 1900 middleboxes observed from very large enterprises (>100K hosts) [37], to yield D3.

Fig. 11 shows the composition time and memory consumption for D3. As expected, composition with service chain analysis is more expensive than composing only ACL policies (Fig. 10). Nevertheless, for most cases the composition time is <800 seconds and memory consumption is <20 GB. These are practical numbers for an engine eagerly composing such large number of policies. Note that the composed graph from D3 can have millions of function boxes, replicated from the 3500 functions boxes randomly placed in the inputs graphs; we cached and reused the results of intermediate chain analysis to prevent redundant computations and to save memory.

We also individually measured the time for each of the two steps of our composition algorithm. The normalization step is quite inexpensive and accounts for only around 1% of the total composition time. Although the graph union operation is expensive, we note that it is done in a progressive manner. We first compute unions for pairs of smaller graphs to get larger graphs and then repeat the procedure on the larger graphs; this algorithm is well suited for a multi-threaded implementation (future work).

9. RELATED WORK

Abstractions. Providing powerful abstractions for programming network policies has received considerable attention recently [20, 34, 41, 35, 12, 21, 19, 38, 6, 23, 14, 28]. However, most of these frameworks ([20, 34, 41, 35, 12, 21, 19, 38]) tie policy expression to low-level specifics such as device IP/MAC addresses or current locations of packets within the network. This keeps their complexity as well as the learning curve for users from lowering, especially when they want to express pure end-to-end communication polices without concerning the topology or other details of the network. It also makes them non-intuitive to express dynamic policies that change based on external events like EP status.

[1]This may be reduced by using packet classification algorithms that run in logarithmic time [16].

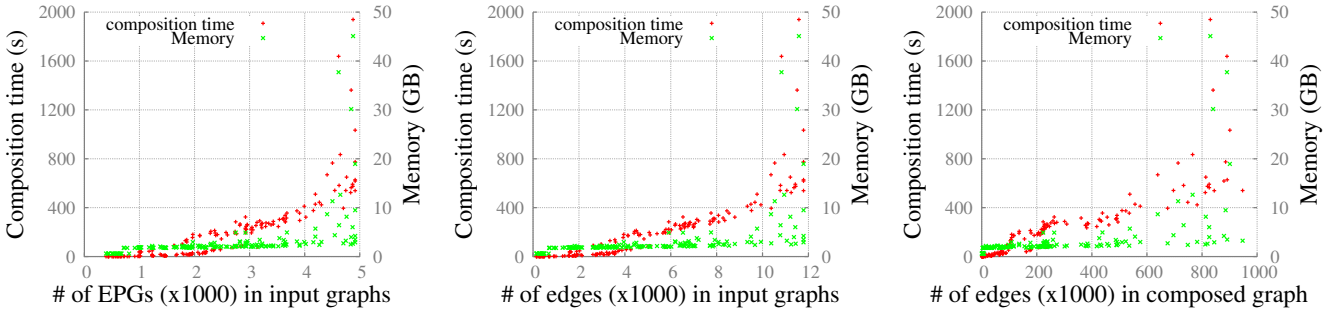


Figure 11: Scalability of PGA with D3 (D2 + function boxes).

Instead, policy intents can be suitably captured using logical labels similar to PGA. Logical labels have been advocated or supported in both networking [6, 23, 14] and non-networking [40, 9, 1] contexts. For example, SELinux [9] assigns arbitrary labels per file in extended attributes and allows users to define access control policies based on the label values. Among network policy frameworks, GBP [6] defines an application-centric policy model that has a notion of EPGs whose membership is determined by logical labels, similar to PGA, although it does not model the relationships between labels or provide a graph model. Flow-based Management Language (FML) [23] is a DATALOG-based query language that specifies access control and forwarding policies on logical entities (e.g., user names queried from external authentication services) and conditions (wireless vs. wired). FML has evolved to Nlog [29] for network virtualization and to OpenStack Congress policy language [7] that expresses policies across multiple cloud services of compute, storage, networking, etc. Despite the expressiveness, it is a generic language, non-trivial to map to graph models which are better suited for networking policies. Further, these abstractions do not model middlebox behavior required for service chain analysis and they do not support automated, eager composition.

Composition. Many frameworks support manual composition of network policies, e.g., [20, 34, 12, 21, 15, 19, 38]. Merlin [38] assumes a hierarchy of users like network admins and tenants and allows tenants to only restrictively refine policies that are explicitly delegated to them by the admin, manually composing both their intents in the process. It is thus not suitable for composing modular policies that can arbitrarily constrain each other where conflict resolution is required. Some frameworks [20, 34, 24, 12, 21] allow network operators to write modular programs and manually compose them into one complex program. These are without conflict resolution support as well. Further, their composition is too coarse-grained and cannot automatically decompose and re-compose complex user policies that often mix different types of intents – ACLs, service chains etc. – in each policy. Others have introduced composition operators to resolve conflicts between independently specified access control policies [15, 19] and bandwidth requirements [19]. They map the set of all input policies onto the leaf nodes of a single hierarchical tree and assign a composition operator to each intermediate node of the tree. Such assignments either require a human oracle [15] or are rigidly pre-determined for each conflict type, making it hard to automatically handle arbitrary conflicts between diverse policy writers.

In GBP [6], users are required to manually write a composite policy connecting multiple EPG pairs. Its users write conditions, prioritized rules etc. such that a unique and correct policy will be chosen for any traffic between endpoints that may have varying statuses (security level, location etc).

In contrast, PGA not only supports automated, eager composition of modular policies, it is also to our knowledge, the first to explore the model of individual policies independently constraining each other for composition.

Extensions. Specifying dynamic temporal behavior of the network as a finite state machine has been studied in [28, 42]. Kinetic [28] can coexist with PGA; different SDN control apps programmed in Kinetic can generate access control and service chain policies in PGA graph model, which the PGA framework can then compose while handling conflicts. I.e., Kinetic controls dynamic label assignments of endpoints while PGA captures and composes the network policy associated with each label (EPG).

Refs. [25, 17, 18] have explored ways to effectively model middleboxes. In comparison, PGA's modeling is more abstract but has been sufficient for the composition scenarios we targeted. Our ongoing work is exploring the utility of the richer modeling capability provided by some of these frameworks for service chain analysis and runtime verification of service chain policies.

Corybantic, Athens and Statesman [33, 13, 39] propose solutions to resolve conflicts on underlying network resources or states between different SDN control modules. NEMO [3] and ONOS [5] provide APIs to model a virtual network topology (of switches and routers) and to specify requirements on the topology links and paths. These systems deal with problems that are orthogonal to PGA which handles end-to-end policies that are agnostic to network specifics. PGA can be extended to incorporate some of these solutions. TAG [30] provides a graph abstraction capturing only bandwidth requirements across application components; extending PGA to incorporate TAG is our future work.

CoVisor [24] composes OpenFlow rule tables, separately compiled from individual SDN controllers, and efficiently updates the composed rule table for a change in an input table. However, composing high-level SDN policies in prioritized OpenFlow rules is inherently inefficient; e.g., adding/re-

moving an SDN app/controller would require re-computation of the entire composed table in [24]. In PGA, incremental update of a composed graph per input graph join/leave/change events is an easy extension. In addition, PGA can use CoVisor to proactively compile OpenFlow rules for the composed graph and to incrementally update the rule table.

10. CONCLUSION

PGA provides an intuitive graph abstraction to express and compose policies. Users (or a policy authoring tool) can simply walk through a composed graph to verify connectivity and service chain requirements. PGA expresses policies and resolves conflicts while minimizing operator interventions. To our knowledge, PGA is the first to model the behavior of closed middleboxes and ensure their correct behavior in a service chain. Automatically combining multiple service chains is another unique feature of PGA. As future work, we plan to enhance PGA in a number of ways: e.g., when a large number of endpoints change their labels at the same time, the PGA runtime should be able to update the network in a scalable, responsive and consistent way. Supporting HW/VM middleboxes, verifying their runtime behaviors and chaining them in more flexible ways (e.g., asymmetric forward/reverse) is additional future work.

11. ACKNOWLEDGEMENTS

We greatly appreciate Nate Foster (our shepherd) and the anonymous reviewers for their insightful feedback. This work is supported in part by National Science Foundation (grants CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

12. REFERENCES

[1] Docker. https://github.com/docker/docker/issues/11187.
[2] Mininet. http://mininet.org/.
[3] NEMO (NEtwork MOdeling) Language. http://www.hickoryhill-consulting.com/nemo/.
[4] Network Service Header. https://tools.ietf.org/html/draft-quinn-sfc-nsh-07.
[5] Open Network Operating System (ONOS) Intent Framework. https://wiki.onosproject.org/display/ONOS/The+Intent+Framework.
[6] OpenDaylight Group Policy. https://wiki.opendaylight.org/view/Group_Policy:Main.
[7] Openstack Congress. https://wiki.openstack.org/wiki/Congress.
[8] Openstack Networking – Neutron. https://wiki.openstack.org/wiki/Neutron.
[9] SELinux. http://selinuxproject.org/page/Main_Page.
[10] Service Function Chaining Architecture. https://tools.ietf.org/html/draft-merged-sfc-architecture-02.
[11] Service Function Chaining General Use Cases. https://tools.ietf.org/html/draft-liu-sfc-use-cases-08.
[12] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.
[13] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul. Democratic Resolution of Resource Conflicts Between SDN Control Programs. In *CoNEXT*, 2014.
[14] M. Banikazemi et al. Meridian: an SDN platform for cloud network services. *Communications Magazine, IEEE*, 51(2):120–127, February 2013.
[15] P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. A Modular Approach to Composing Access Control Policies. In *CCS*, 2000.
[16] H. Edelsbrunner et al. Optimal Point Location in a Monotone Subdivision. *SIAM J. Comput.*, 15(2):317–340, May 1986.
[17] S. K. Fayaz and V. Sekar. Testing Stateful and Dynamic Data Planes with FlowTest. In *HotSDN*, 2014.
[18] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *NSDI*, 2014.
[19] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*, 2013.
[20] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.
[21] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A Coalgebraic Decision Procedure for NetKAT. In *POPL*, 2015.
[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.
[23] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *WREN*, 2009.
[24] X. Jin, J. Gossels, and D. Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *NSDI*, 2015.
[25] D. Joseph and I. Stoica. Modeling Middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 22(5):20–25, Sept. 2008.
[26] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT*, 2013.
[27] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.
[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. In *NSDI*, 2015.
[29] T. Koponen et al. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.
[30] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.
[31] L. Li et al. PACE: Policy-Aware Application Cloud Embedding. In *INFOCOM*, 2013.
[32] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic Engineering with Forward Fault Correction. In *SIGCOMM*, 2014.
[33] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner. Corybantic: Towards the Modular Composition of SDN Control Programs. In *HotNets*, 2013.
[34] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.
[35] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-defined Networks. In *NSDI*, 2014.
[36] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From Policies to Pipelines. In *ICFP*, 2014.
[37] J. Sherry et al. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. *SIGCOMM CCR*, 42(4):13–24, Aug. 2012.
[38] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *CoNEXT*, 2014.
[39] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-state Management Service. In *SIGCOMM*, 2014.
[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.
[41] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.
[42] D. M. Volpano, X. Sun, and G. G. Xie. Towards Systematic Detection and Resolution of Network Control Conflicts. In *HotSDN*, 2014.