Network Policy Whiteboarding and Composition

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

ABSTRACT

We present Policy Graph Abstraction (PGA) that graphically expresses network policies and service chain requirements, just as simple as drawing whiteboard diagrams. Different users independently draw policy graphs that can constrain each other. PGA graph clearly captures user intents and invariants and thus facilitates automatic composition of overlapping policies into a coherent policy.

CCS Concepts

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

1. INTRODUCTION

Managing network traffic and application workloads based on high-level policy is an important current topic in SDN, cloud and NFV (Network Function Virtualization). Recent policy abstractions and languages proposed in [5, 3, 1, 2] provide high expressiveness for describing policies, but they have steep learning curves and are thus confined mostly to expert users. Moreover, existing work do not yet clearly support modular composition of multiple policies from different stakeholders; SDN and cloud automation enabled a large number of diverse parties (operators, application admins, tenants/end-users) and control programs (SDN-apps, network services) to generate network policies independently and dynamically. Simply running those policies in series/parallel [3], or in a prioritized order often fails to satisfy the intent of stakeholders because 1) each policy can consist of multiple dimensions (security, QoS, middlebox), and 2) their intents can constrain each other in each dimension. And some intents are not clear enough in existing languages to systemically compose them; they need to be manually decomposed and re-composed by a human oracle who has accurate knowledge of the hidden- or joint-intents of different stakeholders.

We proposed Policy Graph Abstraction (PGA) at this year SIGCOMM [4] with two major contributions: 1) intuitive high-level graph model enabling diverse stakeholders to clearly express their portable intents without needing IP/subnet assignment, and 2) eager, automatic composition of independently specified policy graphs, creating one coherent composed policy. In particular, eagerly detecting and composing overlapping policies (on common traffic) prior to system deployment is critical to prevent unexpected runtime behavior caused by incomplete resolution of conflicts between the overlapping policies. Our full paper [4] formally describes and evaluates the model and composition algorithm; this demo prototype shows the full cycle of policy graph specification, composition and deployment in OpenStack cloud platform through our GUI. We will also show how the graph model and its constructs facilitate the eager & automatic composition.

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

SIGCOMM '15 August 17-21, 2015, London, United Kingdom

ACM ISBN 978-1-4503-3542-3/15/08.

DOI: http://dx.doi.org/10.1145/2785956.2790039

2. POLICY GRAPH

A PGA policy graph takes three forms: input graph, composed graph, and deployed graph. Each policy stakeholder creates one or more *input graphs*, which our graph composer composes together from multiple stakeholders and generates a single system-wide *composed graph* (Fig. 1) with log/error messages. Some input graphs' policies can be rejected or changed during the composition process, e.g., due to global or other policies' constraints. The owners of rejected/changed input graphs can accept the changes or revise their input graphs and resubmit. A composed graph can be approved by an authority (e.g., operators) and becomes ready to deploy. Once deployed down to the underlying network, it is stored as a *deployed graph* and used for runtime policy enforcement.

Graph Model: a PGA user expresses his/her input policy as a directed multi-graph where a vertex represents a group of network endpoints (EPG, endpoint group) sharing common properties expressed in terms of *labels*. A directed edge connecting a pair of EPGs expresses various intents on the communication from the source EPG to the destination EPG. Each edge has a match classifier on L3/L4 header fields, refining the traffic that the edge cares about from the source to the destination. A user can draw multiple edges of different classifiers between the same EPG pair.

EPG Labeling: high-level, portable policy specification must be decoupled from low-level specifics (e.g., IP/MAC addresses). For that, a PGA user can 1) define an EPG (graph vertex) as a boolean expression of logical labels, each representing diverse endpoint attributes such as tenant ownership, network location or security/QoS status, and 2) specify policies for the EPG. In a running system, the policies are applied to a set of endpoints that have labels satisfying the boolean expression specified for the EPG. In a cloud example, an EPG of [Infected&Tenant1&¬AZ2] represents malware-infected VMs of Tenant1 placed in anywhere else than Availability Zone 2.

Decoupling policy specification from address assignment poses a challenge in eagerly identifying the policies that overlap and thus need to be composed together. PGA captures the overlapping or

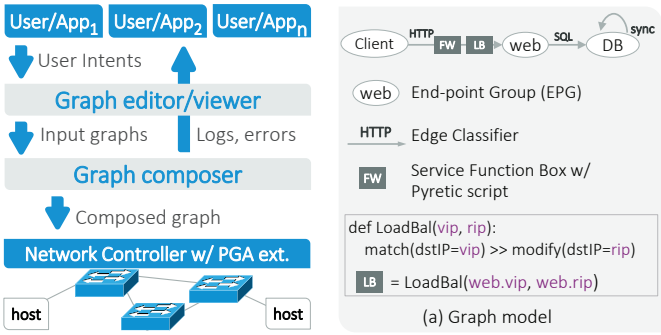

Figure 1: PGA components.

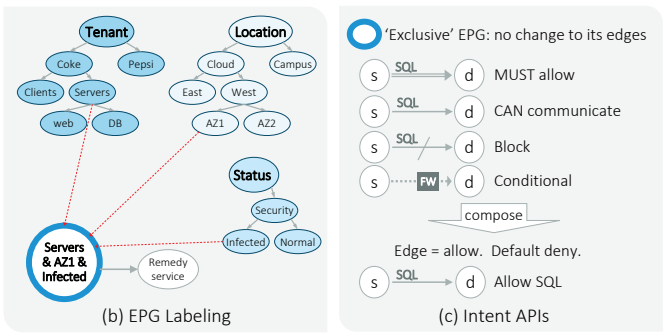

Figure 2: PGA model and APIs.

mutually exclusive relation between labels in *label hierarchy trees* and *label mappings* and uses them to eagerly identify policies to be composed together. A parent label (e.g., 'Servers' label in Fig. 2(b)) is a boolean OR of its children labels (web and DB), thus an EPG defined from the parent label can have overlapping endpoints with EPGs defined from the children labels. Any set of labels in the same tree that do not have an ancestor relationship (e.g., web and DB) are mutually exclusive, thus cannot be assigned on any common endpoint. Two labels from different trees that can be assigned on a common endpoint have a symmetric mapping relation in *label mapping*, e.g., Servers and Infected labels in Fig. 2(b). PGA collects label information from external services (tenant authentication, VM placement, SDN control apps, etc.) and display them in the graph editor to help users easily and correctly create EPGs from the labels.

ACL Intent APIs: Since input policies can be changed during the composition process, PGA provides a few constructs for individual stakeholders to clearly express their intents/invariants in input graphs. A user can choose from four types of edges to express their access control intents: Must Allow, Can Communicate, Block, and Conditional (Fig. 2(c)). These edge types are evolved from the *composition constraint table* construct of [4] that are designed to constrain on the policy changes made by any other policy graph being composed together. We found the edge type representation provides better usability and clarity in GUI than the constraint table form; they achieve the same goal of supporting automatic composition of overlapping ACL policies.

A *Must Allow* edge in an input graph indicates that the communication specified by the edge classifier must be allowed regardless of other policies to be composed with, i.e., an Allow edge must exist for the classifier space in the final composed graph. Management traffic and service-critical communications are examples of this Must Allow type. A *Can Communicate* edge also intends to allow traffic but it can be eventually blocked by other policies. An enterprise IT can use this edge type to 'open' a set of TCP ports for employees to use while allowing them to block some or all the ports per their policies. A *Block* edge type is a constraint that prohibit certain traffic from ever being allowed even if they are allowed by other policy graphs; i.e., the specified traffic must not be allowed in the composed graph. Block edge is useful to implement ACL blacklisting, while using Can type for $\{*\} - BL$ (BL is a set of protocols/ports to block) correctly expresses the weak Allow intent for everything else than BL; using Must Allow for $\{*\} - BL$ might be too strong and might create unnecessary conflicts with Block intents. A *Conditional* edge means there is no need to allow in this graph but it can be allowed in other graphs. By default *no edge* in an input graph is interpreted as Conditional; a policy writer can change the default setting to Block (for whitelisting) or Can (for blacklisting).

With these fine-grained intent APIs, a conflict is clearly detected and resolved: Must/Can edge overrides Conditional while Block wins over Can. A conflict between Must and Block is reported to users or can be resolved based on ranking between policies or stakeholders: e.g., IT admin's Block wins over employee's Must edge. Rejected policies are reported to the policy owners to review, which is possible in PGA's eager policy composition. PGA also provides convenient shorthands such as 'exclusive' EPG that doesn't allow any change or addition of edges of the EPG [4]; e.g, in Fig. 2(b), the EPG of Infected Servers in AZ1 is marked as 'exclusive', preventing other policies from thwarting the intention of the policy writer to redirect all traffic from infected hosts to a remediation server.

Service chaining: An edge of any type except for Block can have a chain of *function boxes*, each representing logical network function (NF) such as firewall, load-balancer or byte-counter. Regardless of its deployment methodology (HW, VM, or SDN programs), each NF can be modeled (by users or vendors) in our variant of Pyretic programming language [3] describing its high-level bounding behavior. This *gray-boxing* approach enables dependency analysis and ordering of NFs in a composed service chains [4]. A service chain on a Conditional edge specifies a pure service chain requirement which is instantiated if and only if the edge's match classifier overlaps the classifier of a Must/Can edge in a different policy graph. E.g, a cloud operator wanting to count bytes of every tenant traffic can use a Conditional edge with a byte-counter function box on it; this edge itself does not allow any traffic, but it counts bytes if any traffic is allowed by tenant policies.

3. COMPOSITION AND DEPLOYMENT

The graph composer computes a union of EPGs of input graphs and merging ACL policies given by edge classifiers and service chain requirements, except when doing so would violate the global invariants and user-given constrains [4]. Our prototype supports two deployment paths: 1) POX OpenFlow controller, reactively generating OpenFlow rules for pkt-in events based on an eagerly composed policy graph, and 2) OpenStack network service, Neutron, for proactively configuring security and service chain rules through Neutron APIs. The demo will show the system running in OpenStack.

4. REFERENCES

[1] OpenDaylight Group Policy. https://wiki.opendaylight.org/view/Group_Policy:Main.
[2] Openstack Congress. https://wiki.openstack.org/wiki/Congress.
[3] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.
[4] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *SIGCOMM*, 2015.
[5] 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.