SLA-Verifier: Stateful and Quantitative Verification for Service Chaining

Ying Zhang, Hewlett Packard Labs
Wenfei Wu, Tsinghua University
Sujata Banerjee, VMware
Joon-Myung Kang, Hewlett Packard Labs
Mario A Sanchez, Hewlett Packard Labs

***Abstract*—Network verification has been recently proposed to detect network misconfigurations. Existing work focuses on the reachability. This paper proposes a framework that verifies the Service Level Agreement (SLA) compliance of the network using static verification. This work proposes a quantitative model and a set of algorithms for verifying performance properties of a network with switches and middleboxes, i.e., service chains. We develop SLA-Verifier and evaluate its efficiency using simulation on real-world data and testbed experiments. To improve the SLA violation detection accuracy, our system uses verification results to optimize online monitoring.**

I. Introduction

Meeting network Service Level Agreements (SLAs) is critical for network service providers. SLAs specify performance assurance metrics such as packet loss, delay, jitter, and network availability. Failing to meet SLA guarantees can result in poor performance and significant revenue loss. Thus, SLA compliance verification, i.e., verifying whether the network in a given configuration can deliver the performance within the SLA bounds, is critical for the operators. This becomes even more important in emerging new network environments, such as Software-Defined Networks (SDN) and Network Function Virtualization (NFV), which increase the dynamics of network routing and resource allocation. SDN enables fine-grained flow-level dynamic routing, which can be triggered by various network state changes (e.g., failures). NFV enables virtualizing and scaling network services up or down with changes in demand. Upon workload changes or failures, flows may be steered to different paths or through different middleboxes to react to the changes. Thus, it is crucial to verify that SLAs are satisfied in these new dynamic settings.

Recently, network verification has been used to detect configuration errors. Current techniques focused on verifying basic connectivity invariants such as loop free-ness and reachability [1], [2], [3]. However, while connectivity is the basic guarantee that the network should provide, performance guarantees are equally important to customers, for metrics such as latency, packet loss rate, bandwidth, availability, etc. We call verifying these performance properties as *SLA verification*.

Traditionally, SLA compliance/violations are checked via active measurements [4] or passive modeling [5]. However, we propose that a two-step SLA compliance checking mechanism consisting of static verification and online measurements may be more efficient than just measurements alone. Even though the traffic changes fast, by analyzing the traffic distribution and the configuration of the network, we can identify possible SLA violations using static analysis even before traffic arrives. We can combine the static verification and online measurement to accommodate the inaccuracy in traffic distribution estimation.

TABLE I. Example SLA queries

1.Are the minimum bandwidths for all flows from A to B within the bounds defined in SLA?
2.Are the average end-to-end latency for all flows to B within 100ms?
3.Does QoS class X always have higher bandwidth than QoS class Y?
4.Under a single one failure, no link utilization will exceed 95%.
5.The probability of any flow with latency>300ms is below 0.001.

For example, a minimum bandwidth guarantee may not be met because of a misconfiguration of rate limiters or classifying a flow into a low priority class, or wrongly allocating a smaller amount of bandwidth than what is required to the virtual links. Furthermore, in NFV scenarios, the selection of Virtualized Network Functions (VNFs) placements for a particular service chain could be sub-optimal: for example, consider that one VNF is in one Point of Presence (PoP) and the next VNF in the chain is in another PoP. If the propagation delay between the two PoPs is larger than the required latency guarantee, then clearly the latency SLA will not be satisfied even if none of the nodes along the path is congested.

We give a few representative examples of questions related to SLAs that we can verify in Table I. To answer the first question, we need to first verify the reachability from A to B. This can be done using standard techniques like HSA [2] and new techniques that are being developed to verify reachability in the presence of middle-boxes. In addition to reachability, we need to identify the switch and middlebox configurations (e.g., rate limiting, priority setting, buffer sizing, etc.) along the paths and compute the minimum bandwidth along the paths. The second question is verifying the latency metric, requiring a hop-by-hop reverse trace of all the flows destined to B and computing the average delay across all flows. The third question asks the comparison between two classes of packets, which can be represented as two disjoint cubes in the hyper-dimensional header space. The fourth question is about the link utilization after a failure given the current flow rules. Finally, the fifth question checks if the probability of a latency violation is bounded by a threshold. QNA [6] has an initial proposal on quantitative verification. But it does not handle probability and stateful boxes.

We extend our current connectivity verification tool [7] to answer the above questions. However, it is not a trivial extension. First, when computing the performance metric, we need to support the composition of different performance metrics on different header spaces together. Second, in order to verify networks that include middleboxes, we need to account for the dynamic properties of each middlebox. Middleboxes

978-1-5090-5336-0/17/$31.00 ©2017 IEEE

maintain internal state variables and corresponding actions on packets. Therefore, the middlebox may forward packets of a flow to different paths according to its internal states [8]. For example, a cache server forwards a flow along a path with certain probability, depending on whether the object is in its cache. The state changes within the middlebox are often dependent on the prior packets within the flow. The middlebox will make future decisions based on the state changed by current packets. Thus, the performance of a flow could depend on the sequence of packets. Third, the static verification estimates performance based on historical traffic/performance distribution. However, due to the real-time traffic dynamics, the static verification may be inaccurate.

In this paper, we propose a system called SLA-Verifier that performs SLA verification on service chaining: a network with SDN switches and middleboxes (VNF/NFs). It computes the performance metrics by first finding the path that a packet header space traverses. To the best of our knowledge, this is the first paper addressing SLA verification considering middleboxes. We make the following contributions.

- We propose a model of switches and middleboxes that incorporates SLA performance metrics. We then propose a static SLA verification algorithm. It takes the configuration, traffic distribution, and network statistics and then answers various queries. We present three algorithms for queries on flow space, states, and performance.
- To support online update, we propose a graph based online SLA verification algorithm, because a graph data structure is efficient and well suited for online verification.
- To accommodate the inaccuracy of verification due to run-time traffic dynamics, we introduce a verification assisted SLA monitoring component. It uses the verification result to allocate monitoring resources in order to maximize the probability of detecting SLA violations.
- We develop a prototype call SLA-Verifier. We use extensive simulation and testbed evaluation to show that static verification is feasible and our solution is scalable. We can verify a network with 1000+ nodes in less than 20 ms. We have tested it on OpenStack with Neutron-based service chaining implementation, which illustrates its practical usage.

II. System Overview

We describe a high level overview of SLA-Verifier and list the challenges in this section.

Architecture. The input to SLA-Verifier includes the network topology, SDN flow tables, the Network Function (NF) configurations, NF models, and the performance model generated from historical measurements. Examples of the distribution model include delay distribution and load distribution on each link. For service chaining applications, when flows traverse a sequence of NFs, we also need to have the NF performance models. The main component of SLA-Verifier is the static verification module, which contains two sub modules. The offline module takes a snapshot of the configuration and answers various performance related queries. It checks if there is any SLA violation given the current configuration.

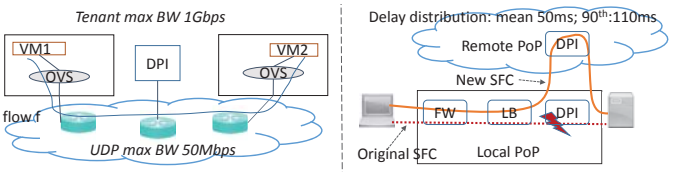

Fig. 1. SLA violation examples

Using the offline analysis results, the online module builds a quantitative and stateful forwarding graph. At run time, upon any configuration or routing changes, this module uses the forwarding graph to identify if the changes will lead to an SLA violation. Its equivalence in reachability testing is NetPlumber [9] or Veriflow [1]. The verification may find some paths that have not violated SLAs yet but could be likely violate it when traffic dynamics change. This is the fundamental problem of static SLA violation because the traffic input may not be accurate. To handle this, we couple the SLA verification with an online monitoring module. Different from general SLA monitoring, it uses the verification results to optimize the monitoring resources.

Target problems. SLA-Verifier aims at detecting two types of problems: *hard mis-configuration* and *soft probabilistic violation*. We illustrate them using two examples in Figure 1. The first example is that the guaranteed SLA for a tenant is a minimum bandwidth of 1 Gbps. This is provided by setting the rate limiter in the hypervisor to 1 Gbps for a VM of this tenant. Along the path of a flow originating out of this VM, there is a Deep Packet Inspection (DPI) NF. The switch connecting to the DPI NF has a configuration to rate limit any UDP flow to 50 Mbps in order to prevent DoS attack to the DPI. Thus, flow f of this tenant, which happens to be a UDP flow, experiences 50 Mbps, which violates the original SLA. In this case, the switch should have a higher priority rule to create an exception for this flow. The second example is shown in the right of Figure 1. A tenant's traffic traverses a service function chain with a Firewall, Load Balancer, and a DPI NF. It requires a maximum latency of 100 ms. This SLA is easier to satisfy when all the NFs are in the same PoP. However, upon detecting a failure or a traffic spike, the network controller decides to use the DPI in a remote PoP. The 90th percentile of the inter-PoP link delay is 110 ms. Thus, the new path will have at least 10% chance of violating the SLA.

Challenges. We believe that the extension of reachability verification to SLA verification is not trivial. First, the performance metric can be defined on different header space. When doing path analysis, we need to compose different performance metrics across the header space. For example, in the top of Figure 1, the QoS configuration for the tenant at OVS needs to be composed with the configuration of UDP at intermediate switch. One composition may yield the following: a UDP flow from VM_1 to VM_2 should have a maximum bandwidth of 50 Mbps. The composition used in HSA focuses on set operations, not quantitative metrics.

Second, advanced network functions may have complex configurations that can affect performance depending on the

states. For example, F5 load balancer performs rate limiting if the number of requests exceeds a certain threshold [10]. Existing work has found that NFs' performance also depends on its internal states [11]. Currently there is no general performance model for NFs and certainly no stateful performance model.

III. Quantitative Network Models

In this section, we first define two kinds of algebra to describe network forwarding and performance behaviors, and then model stateful networks. Finally, we formulate the verification goals of SLA-Verifier. Our key differences with HSA [2] and QNA [6] are the modeling of probability distribution and stateful devices.

A. Operational Algebras

The first kind of algebra is about header space operations, which describes how switch rules on a path are composed as a symbolic flow traverses along the path. The second kind is about performance operations, which describes how to compute a path's end-to-end performance metrics according to the metrics on individual hops along the path.

1) Header Space Algebra: We describe algebra below. A packet header can be viewed as a sequence of 0 and 1, i.e., $\{0, 1\}^+$. While a switch rule's match field is a sequence of 0, 1 and *, i.e., $\{0, 1, *\}^+$, where * is a wildcard for 0 and 1. For example, a switch rule with match field 100xxx means all packet headers with prefix 100 followed by any three bits.

TABLE II. Intersect operation

$\cap$	**0**	**1**	*
0	0	$\emptyset$	0
1	$\emptyset$	1	1
*	0	1	*

Intersection operation "$\cap$" is defined between bits as is shown in Table II, where a wildcard (*) intersecting with a concrete value (0, 1) is the concrete value, a concrete value intersecting with itself is still itself, and 0 intersecting 1 is $\emptyset$. The intersection of two sequences is computed bitwisely; if any bit in the result is $\emptyset$, the whole result is $\emptyset$; otherwise, the result is the sequence of intersecting all bit pairs.

2) Performance Metric Algebra: For each hop or link in the network, its performance is described by a performance vector $P = (p_1, p_2, ..., p_n)$. Each dimension of the vector describes a certain performance metric. In this work, we consider the following metrics: hop count, bandwidth, link load, and latency.

Referring to Network Calculus and QNA [6], we adopt the join operation "$\sqcup$" to merge performance metrics. The join of two performance vectors results in a vector, where each dimension of the result vector is the join of the two original vectors' corresponding dimensions.

$$P_1 = (p_{11}, p_{12}, ..., p_{1n}), P_2 = (p_{21}, p_{22}, ..., p_{2n})$$

$$P_1 \sqcup P2 = (p_{11} \sqcup p_{21}, p_{12} \sqcup p_{22}, ..., p_{1n} \sqcup p_{2n})$$

The join operation of different metrics is defined differently. QNA defines this operation for metrics with concrete values such as hop count, QoS bandwidth. The join operation of the above four metrics is straightforward. For example, hop count is *joined* by summing up all the per-hop counts (=1), and QoS bandwidth is *joined* by computing the minimum bandwidth assignment in QoS policies along the path.

TABLE III. Definition of $p_1 \sqcup p_2$ for different metrics

Metric	Definition
Hop Count	$p_1 + p_2$, (usually p_1, p_2 is 1 on each hop)
Bandwidth	min(p_1, p_2), p_1 and p_2 are defined in QoS
Latency/Load	$\int_{-\infty}^{+\infty} f_1(x-t)f_2(t)dt$, $p_1 \sim f_1$, $p_2 \sim f_2$

In SLA-Verifier, we extend the scope of this operation to metrics with varying values, i.e., metrics that follow a distribution. For example, the link load and latency are not a constant value in the entire duration of a flow; instead, each of their values follows a given or measured distribution. Two important questions arise: *how to aggregate individual flow's load of a link to get the link's load distribution?* and *how to accumulate per-link latency on a path to get the distribution of end-to-end latency?* SLA-Verifier defines the join operation of two distributions to be the convolution of the two distributions' probability density function (Table III).

B. The Network Model

TABLE IV. Terminologies

Symbol	Meaning
B	the set of network boxes (e.g., switches, middleboxes)
E	the set of directed network links
F	the flow header space (i.e., $\{0, 1, *\}^+$)
P_x	the performance vector of entity x, $x \in B \cup E \cup F$
S_b	the state set of box b

Table IV defines the terminologies in SLA-Verifier to describe a network with quantitative metrics. A network topology is modelled by a set of boxes and directed links between boxes. We use "box" to denote both stateless switches and stateful boxes and introduce unified model to describe their behaviors. In the table, B and E are network entities, F is the set of flow entities. For each entity x, vector P_x is defined to describe its performance behavior. SLA-Verifier verifies stateful networks, thus, it also defines the state set (i.e., S_b in Table IV).

Packet processing is the basic functionality of a network box. In this procedure, a box reads a packet and its internal states, and then possibly transforms the packet, sends it out and updates the internal states. This procedure can be formally expressed as a *processing function*: $h : S \times F \mapsto S \times F \times B$, where the input (S, F) indicates reading states and packets, and output (S, F, B) indicates the updated states, (possibly) transformed packets and next hop in the network respectively. For a stateless device, such as switches, the processing function falls back to traditional rules, i.e., $h(f, *) = (f, *, b)$, where $*$ is a wildcard matching anything, $f \in F$ and $b \in B$.

For boxes that may transform packets, SLA-Verifier introduces transformation function T to express this behavior. Then the packet processing function becomes $h(f, s_{in}) = (T(f), s_{out}, b)$. We also use identity function I to denote packet processing without transformation, where $I(f) = f$.

With these definitions, a *rule* in a network box can be expressed in the format of $(f_{in}, s_{in}, T, s_{out}, nxt)$. For end

hosts that sending packets, the rule is $(*, *, I, *, nxt)$, and stateless switch rules are $(f, *, I, *, nxt)$.

C. Verification Goals

SLA-Verifier aims to verify the reachability, loop-free, no-black-hole and performance properties in a network. We first define a *per-path reachability* for a flow. A path is defined as a sequence of pairs of box and state, i.e., $[(b_1, s_1), (b_2, s_2), ..., (b_n, s_n)]$, where b_i is in state s_i. A flow f is *reachable* on a path if and only if the following three conditions are satisfied. Compared to HSA [2], our description includes quantities.

1) The flow f's header space can be matched and transformed on each hop of the path. That is, on each box b_i, there exists a rule $(f_{in,i}, s_{in,i}, T_i, s_{out,i}, nxt)$, where $f_{out,i-1} = f_{in,i}$, $s_{in,i} = s_i$ and $T_i(f_{in,i}) = f_{out,i}$ ($f = f_{in,0}$).
2) The states of all boxes on the path are satisfied, i.e., $\forall i$, box b_i's states s_i can hold simultaneously.
3) Performance metrics for the flow and all on-path entities (including boxes and links) can be satisfied. Flow-based performance metrics (i.e., hop count, bandwidth, latency) are accumulated along the path, and the end-to-end metric should satisfy the performance requirements; link-based metric (i.e., link load) is computed by accumulating per-flow load on the link, and per-link requirements should be satisfied (e.g., within maximum load threshold).

SLA-Verifier checks the following invariants of a network.

- **Reachability.** For any flow f that is expected to be reachable from source s to destination d, there exists a set of paths, where the flow space of all paths equals f. If the disjunction of all paths box states is always true (i.e., $\vee_{p \in paths} \wedge_{b_i \in p} \mathcal{P}(b_i, s_i)$[1] is true, and paths are from s to d), the flow is *fully reachable*, otherwise it is *conditionally reachable.*
- **No leakage.** For any flow f that is expected not to be reachable from source s to d, there is no path from s to d. That is, there exists no path that satisfies the three per-path reachability requirements.
- **Loop-free.** For any path p that satisfies a flow f, the flow does not traverse the same box with the same state twice. That is, there is no (b_i, s_i) and (b_j, s_j) on p where $b_i = b_j$, $s_i = s_j$ and $i \neq j$.
- **No black holes.** For any maximal path p that satisfies a flow f, if it cannot be forwarded at the last hop of a p, then p should be an end host.

IV. SLA-Verifier Methodology

This section is the main methodology in SLA-Verifier to verify quantitative metrics of stateful networks. SLA-Verifier first preprocesses rules to get a disjoint rule set for each network box, and then verifies flow header space, box state space and performance constraints respectively.

A. Rule Refinement

In an actual box configuration, there are usually multiple rules, which may overlap with each other. To reason about which rule would be applied to a certain flow, it is necessary to refine the rules first. For example, there is a high-priority rule with flow matching and next hop $(11**, n_1)$ and a low-priority rule $(1***, n_2)$ on the same box, the overlapping flow space is $11**$. If a symbolic flow $****$ is input to this box, and rules are matched one by one, the final output would be $(11**, n_1)$ and $(1***, n_2)$ causing incorrect verification result: flows $11**$ to arrive at multiple destinations.

Therefore, SLA-Verifier first refines all rules in a box. For original rules, it expects to output a new set of rules where (1) new rules do not overlap with each other and (2) cover the same space with the original set, and (3) each flow would be taken same actions on in both rule sets.

The RefineRule function in Algorithm 1 performs this task. It first sorts all original rules in descending order according to priority breaking ties by prefix length. Then the sorted rule set is iterated, and each rule is refined by subtracting the union of all previous rules. Thus a refined rule set that satisfies the three requirements is finally got.

Algorithm 1 SLA-Verifier Methodology

```
1: function RefineRules(B)
2:     for b ∈ B do
3:         refRules := ∅, rules := b.rules.sort()        ▷ Descend by Priority
4:         for r ∈ rules do
5:             newRule := r − ∪refRules, refRules.add(newRule)
6:         b.rules := refRules
7: function VerifyFlowSpace(src)
8:     paths := ∅, cand := ∅
9:     flow := *, path := [(src, *)], perf := *, cand.add( (flow, path, perf) )
10:    while cand ≠ ∅ do
11:        c := cand.pop()
12:        if c.length ≥ K then continue
13:        flow := c.flow, box:= c.path.lastHop()
14:        for r ∈ box.rules do
15:            (f, s_i, T, s_o, n) := r
16:            if flow ∩ f == ∅ then continue
17:            f_o := T(flow ∩ f), path := c.path.append( (n, s_i) )
18:            perf := c.perf ⊔ b.perf
19:            if n == drop or n ∈ EndHosts then
20:                paths.add( (f_o, path, perf) )
21:            else cand.add( (f_o, path, perf) )
22: function VerifyStates( path )
23:    for (b, s) ∈ path do
24:        H_b := GetHistory(b, s)
25:    return ∩ H_b == ∅
26: function VerifyPerformance(paths)
27:    for p ∈ paths do
28:        Verify(p.perf)
29:    for l ∈ E do
30:        l.perf := ⊔_{p∈paths} p.flow.perf, Verify(l.perf)
31: function SLA-Verifier(G(B, E))
32:    RefineRules(B)
33:    paths := ∪_{b∈B} VerifyFlowSpace(b)
34:    for p ∈ paths do
35:        VerifyStates(p)
36:    VerifyPerformance(paths)
```

B. Flow Space Verification

SLA-Verifier models network devices behaviors. Specifically, it statically computes all possible flow paths in the network. To achieve this, SLA-Verifier starts a symbolic flow from each end host, adopt breadth first search (BFS) to find all paths whose length is smaller than K. In each network

[1] $\mathcal{P}(b, s)$ is a predicate meaning box b is in state s.

box, the symbolic flow matches each rule using $\cap$ operation. The symbolic flow is refined and split as it matches with more fine-grained rules, and then forwarded to its next hop.

The VerifyFlowSpace() function (Algorithm 1) computes all K-hop paths originated from src. It initially puts $(src, *)$ into candidate paths, meaning that the flow starts from src with wildcard state $*$. In each round of the search a candidate path is chosen, and then the incoming flow matches with each rule. Once the flow matches a rule, the outgoing flow is computed using the transformation function. The next hop and the outgoing flow is appended into the path, and the internal state and the aggregated performance vector of the box is also recorded. If the next hop is an end host or drop, this path (communication) is complete; otherwise, the flow reaches an intermediate box and is put back into the candidate set.

C. Box States Verification

Flow space verification only guarantees possible data paths for flows in the topology without considering box states along the path. In the final output of flow verification, flow paths are an output together with the box states that satisfy that path (i.e., $\mathcal{P}(b, s)$). For stateful network verification, SLA-Verifier must prove (or disprove) the states of boxes along the path is (un)satisfiable. The internal states of NFs can be built via [12].

The main idea of states verification is to turn states into packet histories. In each box, a certain state would be triggered only after processing certain sequence of packets. For the state s_b of each box b along a data path, SLA-Verifier computes the possible packet history h_b can trigger this state; and then SLA-Verifier checks there exists a packet sequence that satisfies all these histories to confirm whether is data path is satisfiable (i.e., $\cap_b h_b == \emptyset$).

For example, a cache state "cached flow f" can be expressed by history $*f*$, and a firewall state "at least 2 connections of flow f" can be expressed by $*f*f*$. Then, to check whether both states can be satisfied simultaneously is equivalent to check whether $(*f*) \cap (*f*f*)$ is $\emptyset$.

D. Performance Verification

Performance verification includes both hard performance configurations (e.g., QoS bandwidth allocation) and soft probabilistic violation (e.g., possible bursty load). Among the performance metric in Table III, hop count, bandwidth and latency are flow-based metric, that is, the performance metric accumulates (i.e., the join operation $\sqcup$) along the flow's path; while link load is link-based metric, multiple flow's traffic load on each link they traversed.

SLA-Verifier outputs a flow with its accumulated performance vector, thus, the flow-based metrics can be verified easily. For link-based metric, the flow's performance metric (e.g., load) needs to be added to the link, and then the metric is verified per link.

When verifying a hard configuration metric (e.g., hop count, QoS bandwidth), the metric is compared with the pre-set goal. For example, SLA-Verifier user can verify *whether a flow is completed within 10 hops, or whether a flow is allocated 10 Mbps along its path*. When verifying a soft probability (e.g. latency of a flow or load on a link), the accumulated ($\sqcup$) metric is checked by computing the probability of violating a requirement. For example, with latency accumulated along a flow's path (i.e., convolution of the probability density function), SLA-Verifier user can verify *whether 90% of packets can be delivered within 100ms*.

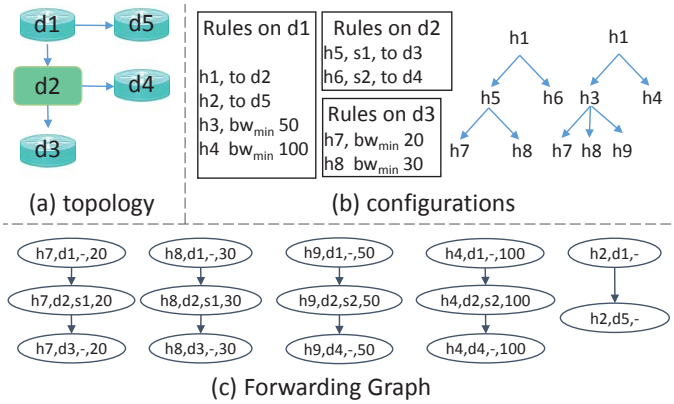

Fig. 2. Quantitative Forwarding Graph example

V. Implementation

In this section, we describe the implementation and algorithms that use the above models to answer various quantitative verification questions. To support efficient online verification, we transform the model to a forwarding graph and then operate on the graph. This idea is similar to NetPlumber [9] extending HSA model [2] to a graph representation in order to support efficient query and incremental update.

A. Quantitative Forwarding Graph Representation

Performing checks in real-time on a large network topology with complex network boxes is challenging. One approach that has been widely used to speed up the checking process is slicing the network to equivalence classes (EC). Each EC is defined as the set of packets that are treated the same across the network. Our EC definition has two key improvements compared to NetPlumber [9] or Veriflow [1]. *1) Quantitative:* packets are in the same equivalent class not only if they traverse the same path but also if they are in the same performance group. We consider the performance group by parsing performance related configurations and by analyzing the performance distribution. *2) Stateful:* packets traverse a sequence of NFs. The path may be changed according to the status of the intermediate NF. This is often known as dynamic service chain. Thus, packets in the same equivalence class will also have the same treatment in any NF states.

We define a *Quantitative Forwarding Graph (QFG)* that represents how packets are forwarded, what performance they are getting, and what NF states they change. QFG is built on top of our existing work on the stateful forwarding graph [7]. In a QFG, each node is denoted as a tuple of packet header space, device, state and performance group, i.e., (H, D, S, G), representing any packet in the packet header space H arriving at a network device (switch or NF) D, when the network device is at a particular state S with performance G. An edge pointing from one node (H_1, D_1, S_1, G_1) to another

(H_2, D_2, S_2, G_2) means when a packet in H_1 arrives at D_1 with state S_1 in performance group G_1, it will be modified to H_2 and forwarded to a device D_2 at state S_2 in performance group G_2. If D_1 does not modify the packet header, then H_1 is equal to H_2. If the packet H_1 does not trigger the state transition, S_1 is equal to S_2. If both devices treat the packet in the same way, then $G_1 = G_2$.

An example of QFG is shown in Figure 2. The topology is shown in Figure 2 (a). There are 4 switches and 1 stateful middlebox d_2. For example, d_2 can be an IDS. If the traffic is normal, it sends to d_3, otherwise to d_5. The tables and relationship between headers are shown in (b). Using them, we can create a QFG in (c).

A key advantage of QFG representation is that whenever the network device configurations are updated, it is easier to find the affected QFG nodes as well as their dependencies, thus, the verification can be limited to only those affected flows/boxes.

B. Verification implementation

Finding EC. We build the QFG graph from parsing the OpenFlow table rules in all switches and NF configurations. For tables in each network device (switch or NF), we first group all the rules based on the actions. Then we create one node for each group, which contains four fields (H, d, s, A), i.e., header, device, state, and action. Next, we compute the path starting from each node by tracing its next hop in the action. For each hop, we create a corresponding node and insert it to the path. In the meanwhile, we need to backtrack to split the parent nodes along the path. For example, as shown in Figure 2, h_1 is split to h_7, h_8, and h_6. Nodes are added to the path iteratively until the next hop is " to drop" or is outside the network.

Verification on a header space. Given a header space h, we find the paths that intersect with h, then we go through the nodes of each path. Along the traversal, we compose the performance metrics. Taking minimum bandwidth as an example, the composition will be the minimum between the current value of the bandwidth of the path and the bandwidth of the nodes in QFG.

VI. Verification assisted SLA monitoring

In this section, we show how the verification results can be used for SLA monitoring. Different from existing work which uses either active measurement [4] or passive measurement [13], we consider using both and use the verification result to optimally pick the right set of measurements.

A. Measurements

In the result of SLA-Verifier includes per-flow performance metrics, the difference of the performance metric and the expected property indicate the *stretch* to accommodate violations. For example, for a flow with expected bandwidth allocation 10Mbps, if SLA-Verifier verifies bandwidth configuration (QoS policies) to be less than 10Mbps, a violation is identified; but if the verification result is larger than 10Mbps, due to the dynamic runtime throughput, it is still possible that there exist violations (e.g., burst traffic). In the case of this possible violation, if the stretch is larger, the violation probability is less. For example, if the verified QoS bandwidth allocation is 100Mbps, the bandwidth violation is less likely to happen than an allocation with exact 10Mpbs.

Therefore, measurement techniques are needed to monitor any SLA violation due to traffic dynamics that cannot be solely discovered by static verification. We consider two types of measurements: *Active end-to-end measurement* actively injects probes into the network, such as ping, traceroute, bandwidth measurement, to measure the network performance in real time. *SDN-based passive measurement* installs rules on SDN switches to passively collect performance counters such as packet drops, packet/byte counts. These counters can be used to detect performance issues.

Our goal is to assign a measurement for each path. Normally a path can be measured by end-to-end active probes, such as ping, traceroute, bandwidth measurement. Active measurement is known to have good accuracy to measure the dynamic performance of the network. However, active probes will introduce overhead to both end host and the network. On the other hand, counters reported by each switch along the path can be used for SLA measurement. However, it is often coarse-grained and is also limited by the switch's available rule space. Given the pros and cons, we will pick the most appropriate one for measuring each path.

B. Analysis

Passive measurements are preferred because they have light-weighted overhead to the system, but there are only a limited amount of them. For example, there are limited number of traffic counters on a switch. Thus, we consider monitoring as many flows by passive counters as possible and leaving the remaining flows using active probing. We show that the "stretch" computed by SLA-Verifier can assist the counter-to-flow assignment.

For a flow i the probability of violation is negative correlated with its stretch, i.e., $P(V_i) \propto 1/S_i$. If a flow i is not assigned a counter, the probability to detect the violation is 0; if it owns a counter exclusively, the probability is 1; if a flow share a counter[2], the probability is a conditional probability where the aggregated flows violate under the condition flow i violates. The equation is

$$P(D_i|V_i) = \begin{cases} 0, & \text{flow } i \text{ is not assigned a counter,} \\ 1, & \text{flow } i \text{ exclusively owns a counter,} \\ P(V_S|V_i), & \text{flow } i \text{ share a counter with flow set } S. \end{cases}$$

The probability $P(V_i)$ can be assumed to known by measuring flows performance profile (e.g., throughput/latency distribution in the history), and $P(V_S|V_i)$ can be got by Monte Carlo simulation if all flows violation probability in S is known.

The counter-to-flow assignment problem can be formulate as an optimization problem as is shown in Figure 3. $\{x_{ijk}\}$ are 0-1 variables representing that flow i is assign to the j-th counter on box k. The goal is maximize the probability of detecting the violations, with the constraints of reasonable

[2]Only flows in the same equivalent class can share a counter.

maximize $\Sigma_i P(D_i\|V_i)$, s.t.,	
$x_{ijk} \in \{0,1\}$	(1)
$\Sigma_{jk} x_{ijk} \leq 1, \forall i$	(2)
$S_{jk} = \{i\|x_{ijk} = 1, \forall i\}, \forall j,k$	(3)
$\|\{S_{jk}\|S_{jk} \neq \emptyset, \forall k\}\| \leq K, \forall j$	(4)

Fig. 3. Optimization for counter-to-flow assignment

Algorithm 2 Find monitoring strategy

```
1: function MONITOR(QFG)
2:     for p ∈ QFG.paths do
3:         p.stretch := p.perf − p.expected
4:     QFG.paths.sort()                          ▷ Ascending by stretch
5:     for p ∈ QFG.paths do
6:         n := arg min_{n∈p} (n.bw)
7:         if n.paths < n.capacity then
8:             n.paths.add(p), p.covered := True
9:             n.highFreq −− ≤ 0 ? p.freq := high : p.freq := low
10:    for p ∈ {p | p∈ QFG.paths and p.covered = False} do
11:        e2eprobe.add(p)
12:        highFreq −− ≤0 ? p.freq := high : p.freq := low
```

assignment (constraints 1, 2 and 3) and limited counters per switch (constraint 4).

This integer program (IP) reveals two key insights, (1) flows with tight stretch should be allocated first to increase the probability of detecting violations, and (2) bottleneck nodes on the path should be considered first to satisfy the monitoring resource constraints (i.e., the number of counters). Due to time complexity (NP-hard for IP, and thousands of flows and boxes), we turn to a heuristic algorithm below based on the two insights.

C. Heuristic Counter Assignment Algorithm

The main idea of the heuristic counter-to-flow assignment is to make flow is loose stretch share counters and assign the counter on the bottleneck node first for each flow. These two heuristics can increase the likelihood of finding out violations.

The algorithm that searches for the best monitoring strategy is shown in Algorithm 2. We use bandwidth as an example performance metric. Using the verification result, it first computes the stretch of each EC. Then we sort all the paths according to their stretch in descending order. For each ranked path, the bottleneck node of each path is identified and used for monitoring this path. If the node has enough capacity, and if the current path is not covered by any existing counter, we assign a rule on the node to measure this path. After all the counters in all the nodes are used up, for the remaining paths that still have a high likelihood of violating SLA, we assign end-to-end (e2e) probes for them.

As another optimization, each node may only satisfy limited polling requirements for counters or support a limited amount of probes. We also assign a measurement frequency for each path. For simplicity, we use two levels of frequency: low and high. Paths with a high violation possibility as assigned a high polling/probing frequency. If a node's polling/probing frequency quota is used up, the remaining paths that are assigned to it would be given a low polling/probing frequency.

We illustrate this process using the same example shown in Figure 2. The input to the algorithm is the constraints of monitoring capacity and the QFG. Assuming the network can only afford 2 e2e probes, because $h7$ and $h8$ have a tighter budget (compared to their SLAs), we use e2e probes to monitor them. The remaining ECs are covered using counters. $h9$ and $h4$ both have bottleneck node $d1$. So we use two counters on $d1$ to measure their bandwidth. Finally, assuming $h2$ is the best effort, we use $d5$ which has the most resource to monitor it.

VII. Evaluation

In this section, we use experiments to answer the following questions: 1) given the dynamic nature of traffic, is it even possible to do static quantitative verification? 2) How accurate is our quantitative model on real-world performance data? 3) how does our algorithm scale? 4) what is the benefit when we use verification results to guide online SLA monitoring?

A. Evaluation methodology

Dataset: To answer the first two questions regarding the model accuracy, we use real-world measurement data and topology to evaluate. We use the traceroute data from iPlane [14]. The data set contains 10 days of traceroute data, from 163 locations to 90,193 destination prefixes, covering 2185 distinct ASes and 925K PoPs.

Given 2800K measurements per day, we parse all the traceroute data and generate PoP level network topology and PoP level delay. More specifically, we map each IP in the traceroute to a PoP using IP to PoP mapping database, and then compute the latency between any two PoPs that appear in the path. At the end, we get 98M PoP pairs, which we call them PoP level links or links in the rest of the paper. Because some links may not have sufficient delay measurements, we pick the top 2000 links that have at least 45K delay measurements. Thus, in our input, each link is represented as (PoP_1, PoP_2) and is associated with a list of delays $\{d_1, d_2, \ldots d_n\}$.

Implementation: We developed a prototype in approximately 3K lines of Java. Our prototype takes as input routing rules from Mininet [15] to create a network topology. We evaluated the time complexity on both a linear topology with generic NFs which we can change the number of states.

B. Model accuracy

There are two sources of inaccuracy in our quantitative model. First, we assume that the distribution of a metric is given. However, the distribution is derived from history data and it may not capture all the traffic dynamics in the run time. We first divide the data points to two sets: two thirds of them are used for training set and the remaining are the testing set. We compute the distribution from the training set. We use two ways to describe the model: mean+/-2std or the range below 90^{th} percentile. Each method gives a range. We then compute what fraction of the testing set that are covered by this distribution, which is referred as *prediction accuracy*. Figure 4 shows the fraction across all the links. We can that the model is fairly accurate, especially using the first method. All of them are above 92% and half of them are above 97%.

We compute the path performance by combining the link performance together. This is the second source of

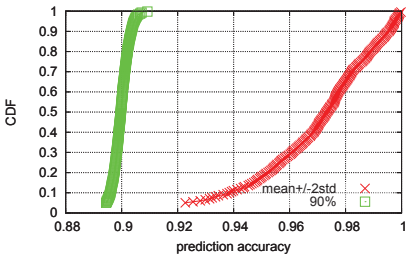

Fig. 4. Model prediction error.

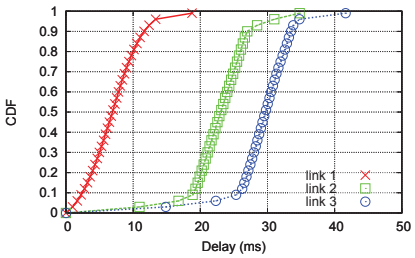

Fig. 5. Compose distribution: link3=link1+link2.

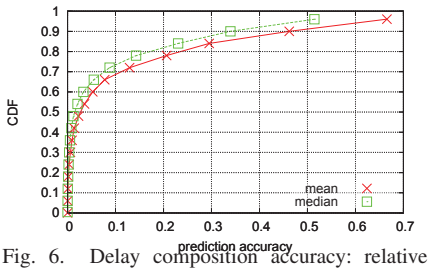

Fig. 6. Delay composition accuracy: relative difference.

inaccuracy. Assuming $link_1 = (PoP_1, PoP_2), link_2 = (PoP_2, PoP_3), link_3 = (PoP_1, PoP2, PoP_3)$, we examine whether the distribution of $link_3$ is indeed a sum of the other two links. Figure 5 shows an example where three links have similar shape of distribution. We systematically examine how accurate the composition is. We take 7380 sets of triple links. For each triple, we compute the $\frac{link_3-(link_1+link_2)}{link_3}$ using both mean and median value. Figure 6 shows the fraction: 70% of the links have a difference less than 10%. Figure 7 shows the absolute difference: 80% are less than 5ms. Going beyond 3 links, we increase the hop count of the path and see that the error is all below 10% in Figure 8. Note that our experiment is different from the network triangular inequality works [16]. In their case, PoP_1 to PoP_3 does not traverse PoP_2.

C. Verification efficiency

We test the scalability of SLA-Verifier as the network size increases. For each network size, we assume each switch has 4 or 8 QoS classes. Each switch randomly assigns one class to each flow. This increases the ECs based on QoS metrics. We first check the time it takes to find the equivalence class (EC) in Figure 9. The EC build time increases as the network size and the number of QoS classes. With 1000 nodes, the time to build the EC graph is 400ms for 4-QoS and 500ms for 8-QoS. Next, we compute the time to answer the first two queries in Table I with both 4-QoS and 8-QoS in Figure 10. Both queries can be answered within 20ms. Query 2 takes longer time because it needs to examine all the paths to a particular destination. 8-QoS has slightly longer time because it has more ECs.

D. Benefit of verification assisted SLA monitoring

We use the same iPlane dataset to quantify the benefit of our monitoring method. We take five ASes that have the largest number of PoPs in our dataset. Their statistics are shown in Table V. The fourth column *path* is the number of PoP level paths found. It is also the amount of probes needed per time slot if we only use active probing. The fifth column *max counter* shows the maximum number of counters on all the nodes if we only use passive counter based measurement. The last two columns show the average probability of detecting SLA violations for all paths. For each path, we randomly assign an maximum delay requirement and compute the gap between it and the estimated value from verification. We assume the detection probability for active probing to be 0.9 and counter based measurement to be 0.6. Comparing the last two columns, we see an 8x improvement. Finally, we vary the delay SLA setting. In Figure 11 we can see that when SLAs are more relaxed, we need fewer active probes and slightly more counters.

E. Testbed evaluation

We test our method on an Openstack service chaining testbed. Our testbed is comprised of 2 HPE ProLiant DL120 Gen9 servers with 128GB of RAM powered by Intel Xeon E5-2600 v3 series processors. We deployed a multi-node DevStack cluster running the Liberty release of OpenStack with one server acting as controller node and the second as compute node. Static SFC chaining is realized via OpenStack's Neutron reference implementation using Open vSwitch.[3]

We spawn four different virtual machines in our single compute node and connect all of them to a single virtual network (via OVS). Two of the VMs act as source (*server VM)* and destination (*client VM*) for video traffic that traverses a service chain. Using the remaining two VMs, we created a 2-VNF service chain using two instances of a commercial network firewall image. Both firewalls have QoS rules. We created two flow classifiers for traffic from both directions. Figure 12 shows a logical diagram of our service chain and the flow of traffic in the network.

We test SLA-Verifier on this testbed in two ways. *path checking:* We manually remove the rules that forward packet to the second VNF. SLA-Verifier reports that the chain reconstructed from the table only traverses one VNF, which conflicts with the SFC requirement. The verification time is 3.2ms. *performance checking:* We further insert a QoS configuration on the firewall, which sets the maximum and minimum bandwidth of a flow to be 5Mbps. Then SLA-Verifier reports an SLA violation because the required minimum bandwidth is 100Mbps. This checking takes 2.8ms.

VIII. Related Work

SLA monitoring ISPs often monitors SLAs by active measurement, which periodically injects probes into the network [18], [19], [20]. From the probes, it computes delay, delay variation, loss rate metrics that are statistically significant. Active measurement may incur additional overhead to the network. And it can only discover problems after it manifests. Our work is orthogonal to active SLA monitoring. We could be used to detect misconfigurations even before deployment.

[3]Neutron's reference implementation is based on programming Open vSwitch with flow table entries that override the default MAC-based forwarding with arbitrary user-defined frame forwarding ordering using the Neutron SFC API. [17]

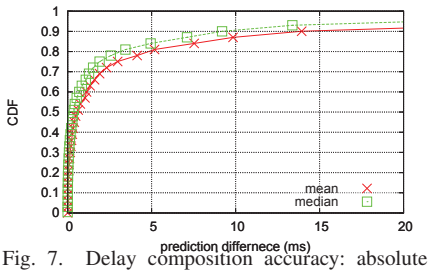

Fig. 7. Delay composition accuracy: absolute difference.

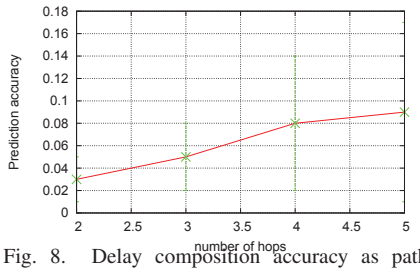

Fig. 8. Delay composition accuracy as path length increases.

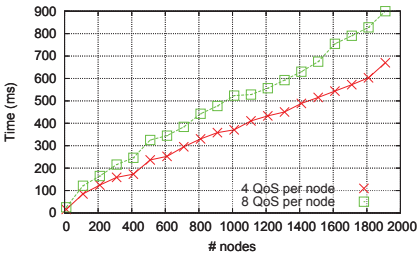

Fig. 9. Time to create Equivalence Class.

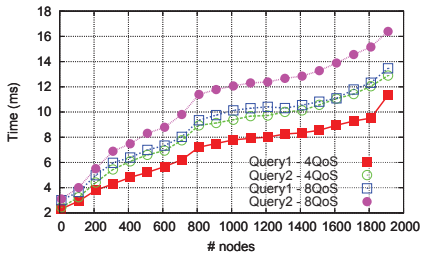

Fig. 10. Verification time.

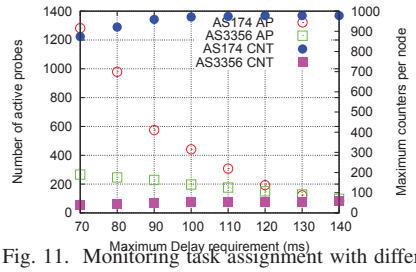

Fig. 11. Monitoring task assignment with different delay SLAs. (AP: active probes; max CNT: max counters per node)

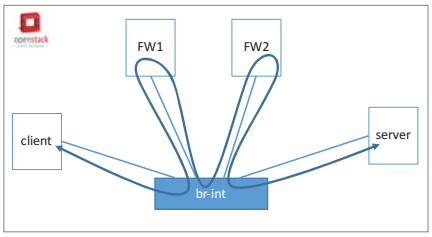

Fig. 12. Logical 2-hop VNF chain running on a single OpenStack compute node.

TABLE V. MONITORING EXPERIMENT SUMMARY. (VAM: VERIFICATION ASSISTED MONITORING)

ASN	Name	# PoPs	# paths	max cnt	detect prob. naive	detect prob. VAM
174	Cogent	105	6752	981	0.3	0.73
4134	ChinaT	77	1242	323	0.22	0.83
3356	Level 3	52	627	78	0.15	0.75
3549	GBLX	41	267	61	0.11	0.92
15557	RIPE	38	132	58	0.09	0.89

Network verification Our work extends existing network verification [2], [1], [9], [21] to quantitative metrics. Our work is also related to middlebox debugging. BUZZ [22] builds the FSM from NFs' source code and then generates testing packets based on the FSM. Our work is orthogonal as we focus on network-wide verification instead of each individual NF.

IX. CONCLUSION

In this paper, we have proposed SLA-Verifier, a system that verifies the high level SLAs by examining the low level performance implementations such as QoS, bandwidth, buffer sizing and rate limiting configurations. While existing work focuses on reachability properties, this paper goes further to handle performance metrics. The method proposed is efficient and general as it handles both stateless and stateful NFs in a uniform way.

REFERENCES

[1] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. NSDI*, 2013.
[2] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012.
[3] N. Bjorner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese, "ddNF: An Efficient Data Structure for Header Spaces," Haifa Verification Conference (HVC), 2016.
[4] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Accurate and efficient sla compliance monitoring," *SIGCOMM CCR*, August 2007.
[5] Y.-W. E. Sung, C. Lund, M. Lyn, S. G. Rao, and S. Sen, "Modeling and understanding end-to-end class of service policies in operational networks," in *Proc. ACM SIGCOMM*, 2009.
[6] N. Bjorner, G. Juniwal, R. Mahajan, S. Seshia, and G. Varghese, "Quantitative network verification." http://cseweb.ucsd.edu/∼varghese/netver.html.
[7] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang, "SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining," in *IEEE SDN-NFV Conference*, 2016.
[8] 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," NSDI'14, 2014.
[9] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. NSDI*, 2013.
[10] "F5 icontrol api." devcentral.f5.com/questions/icontrol-api-for-policies.
[11] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions," in *Proceedings of IEEE NFV-SDN*, 2015.
[12] W. Wu, Y. Zhang, and S. Banerjee, "Automatic synthesis of NF models by program analysis," in *ACM HotNets*, 2016.
[13] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *NSDI 13*, 2013.
[14] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," OSDI'06, 2006.
[15] "Mininet: An Instant Virtual Network on your Laptop." mininet.org/.
[16] G. Wang, B. Zhang, and T. S. E. Ng, "Towards network triangle inequality violation aware distributed systems," IMC'07, 2007.
[17] "Service Function Chaining Extension for OpenStack Networking." http://docs.openstack.org/developer/networking-sfc/.
[18] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Multiobjective monitoring for sla compliance," *IEEE/ACM ToN*, vol. 18, April 2010.
[19] Y. Chen, D. Bindel, H. Song, and R. H. Katz, "An algebraic approach to practical and scalable overlay network monitoring," *SIGCOMM CCR*, vol. 34, August 2004.
[20] B.-Y. Choi, S. Moon, R. Cruz, Z.-L. Zhang, and C. Diot, "Practical delay monitoring for isps," CoNEXT'05, 2005.
[21] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in software-defined networks," in *Proc. NSDI*, 2015.
[22] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *Proc. NSDI*, 2016.