An NFV Orchestration Framework for Interference-free Policy Enforcement

Xin Li and Chen Qian
Department of Computer Science, University of Kentucky
Email: xin.li@uky.edu, qian@cs.uky.edu

***Abstract*—Network functions virtualization is a new paradigm to offer flexibility of software network function processing on demand. Policy enforcement satisfies network function policies that requires flows to traverse through given sequences of network functions. We summarize three desired properties of virtual network function placement, namely policy enforcement, interference freedom, and resource isolation. However, none of existing solutions can satisfy all of them. In this paper, we present a novel SDN-based NFV orchestration framework, called APPLE, to enforce network function policies while providing the above properties. We present detailed design considerations and prototype implementation. We conduct experiments using representative network topologies, traffic matrices, and policy chains. The results from both prototype experiments and simulations show that APPLE is resource efficient and can quickly react to traffic changes.**

I. Introduction

Network functions (NFs) play an important role in modern computer networks. NF deployment is ubiquitous [39]: from performance-improving applications (e.g. WAN optimizers, proxies, and application gateways) to security appliances (e.g. firewalls, IDS). It has been reported that the deployed network functions in a typical network are often as many as routers and switches [39] [38].

Traditionally, the network operators adopt proprietary hardware middleboxes to provide NFs. The hardware middlebox introduces large capital expenses as well as expensive operational costs. To simplify network management, Network Functions Virtualization (NFV) [20] has been proposed a new paradigm to replace hardware middleboxes with software-based NFs running on generic computing platforms. Besides the great reduction in captital/operational expenses, NFV also brings tremendous flexibility to both the network operators and researchers with regards to the deployment, configuration, launch, and cancel of network functions.

In a network, higher-layer applications or network operators may specify *NF policies* that require traffic flows traverse through given sequences of NFs called a *policy chain* (also known as *service chain*). A network performs *policy enforcement* by satisfying all NF policies of the flows in the network. For example, a network operator may specify a policy that requires all http traffic follow the policy chain: firewall $\rightarrow$ IDS $\rightarrow$ web proxy.

Hardware-based NFs are statically deployed in a network. To enforce polices, paths of flows may be changed in the data plane so that each flow can traverse through the specified sequence of NFs by utilizing software defined networking (SDN) [45] [34], called *traffic steering*. Though being straightforward, traffic steering suffers from the following problems: 1) Flow path changes cause interference to other network applications. For example, traffic engineering may assign a path with sufficient bandwidth to the flow. Changing its path may violate the bandwidth guarantee; 2) Traffic steering introduces extra path length. The situation is aggravated if middleboxes are not properly placed in the network; 3) More forwarding rules are installed at switches to support complex traffic steering [34]. Switch memory such as TCAM is a power-hungry and expensive resource; 4) Forwarding loops may occur due to traffic steering [34].

NFV provides new opportunities to enable automatic and flexible NF placement, configuration, and management. In this work, we present a network-wide NFV orchestration framework that automatically installs virtual network function (VNF) instances to enforce policies. The proposed framework provides the following properties.

1) **Policy enforcement.** The sequential order of NFs of a flow required by network applications or operators must hold.
2) **Interference freedom.** The framework should not change the existing flow forwarding paths determined by other network applications such as routing, access control, and traffic engineering. Policy enforcement is completely orthogonal to routing.
3) **Isolation.** Resource isolation requires every VNF instance does not use the computing resource of others (e.g., memory), even if they are installed at a same physical platform. It is desperately desired for both performance and security concerns, especially for multi-tenant clouds where hardware resources are shared by multiple entities. CPU and memory isolation at a same machine can typically achieved by encapsulating each VNF instance into a virtual machine (VM).

The proposed NFV orchestration framework is called APPLE (Automatic aPProach for poLicy Enforcement). APPLE provides all three desired properties discussed above, namely policy enforcement, interference freedom, and isolation of VNF instances. APPLE estimates the NF demand of network-wide flows and proactively installs VNF instances for potential flows, in order to avoid long waiting time for booting. APPLE aims on minimizing the hardware resources consumed by VNF instances. VNF instances are contained in VMs to guaran-

Framework	Policy Enforcement	Interference Free	Isolation
StEERING [45]	✓	x	✓
SIMPLE [34]	✓	x	✓
PACE [24]	x	✓	✓
CoMb [36]	✓	✓	x
Stratos [19]	✓	x	✓
E2 [32]	✓	x	✓
VNF-OP [15]	✓	x	✓
APPLE	✓	✓	✓

TABLE I
COMPARISON OF NF ORCHESTRATION FRAMEWORKS

tee isolation. We have implemented a prototype system of APPLE using recently developed open-source software tools including OpenStack [11], ClickOS [28], Open vSwitch [9], and OpenDayLight SDN controller [10]. **To our knowledge, APPLE is the first implementation of a interference-free NFV orchestration framework based on VMs.**

The rest of this paper is organized as follows. We discuss the related work in Sec. II. We introduce the overview of APPLE in Sec. III. We detail the optimization engine and policy enforcement approaches in Sec. IV and Sec. V respectively. Implementation details are presented in Sec. VII. We conduct trace-driven simulations and show the results in Sec. IX. We provide some discussion in Sec. X and conclude this work in Sec. XI.

II. Related Work

Policy enforcement of hardware NFs mostly relies on traffic steering. Two typical works are StEERING [45] and SIMPLE [34]. Both of them use the SDN technology to forward flows by assigned paths. As discussed, traffic steering may cause interference to other network applications.

A number of NFV placement or orchestration frameworks have been studied in the literature. PACE [24] proposes to use smart VM placement to deploy NFs. However PACE does not consider policy chains and hence cannot perform desired NF policy enforcement. Stratos [19] and E2 [32] provide efficient and scalable NVF provisioning by combining traffic engineering and NF placement. It utilizes traffic steering to enforce policies and hence is not interference-free. VNF-OP [15] is a set of high-level optimization algorithms for VNF placement towards various objectives. It does not provide interference freedom either. In addition, CoMb [36] deploys multiple VNF instances as threads in a physical machine to reduce the installation time. However thread-based VNFs cannot guarantee isolation. The overall comparison is summarized in Table I. Also, a comprehensive survey about NFV can be found in [27].

III. APPLE System Overview

Network Model. APPLE uses the SDN paradigm [29]. All physical nodes that host VNF instances are connected to one of the SDN-enabled switches. When a flow needs to be processed by a VNF, forwarding rules installed on the switch will guide the packets of the flow to the VNF instance and continue forwarding after receiving the packets again from the VNF instance. A central controller obtains network information and installs forwarding rules onto switches via standard APIs such as OpenFlow [29].

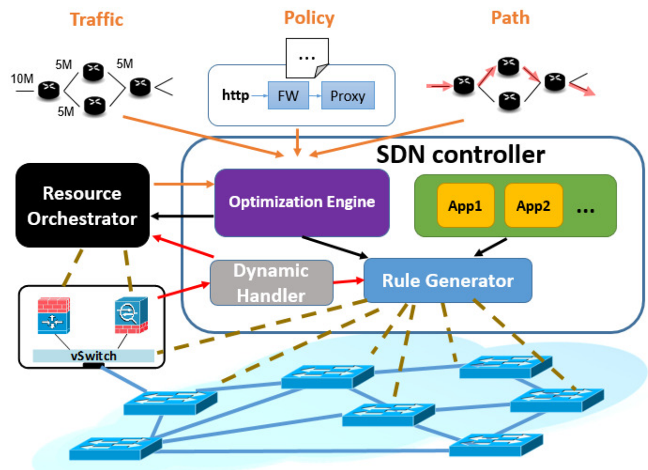

Fig. 1. Overview of APPLE

The overview of APPLE is shown in Fig. 1. APPLE introduces a few new applications, including the Optimization Engine and Dynamic Handler, to the controller other than existing applications such as traffic engineering and access control. The rule generator computes the rules that are installed to the data plane based on the inputs from applications. APPLE also adds a middleware between the control plane and VMs, called Resource Orchestrator. We present the components of APPLE as follows.

APPLE host. Each physical node hosts multiple VNF instances, which is also called an APPLE host. A virtual switch (vSwitch), such as Open vSwitch [9], is installed in the node to switch packets to different VNF instances, which is also connected to the outside network.

VNF Instance. VNF instances run as VMs on physical nodes. VMs guarantee the CPU and memory resource isolation.

Optimization Engine. Like [26], it is a traffic-aware VNF placement algorithm. It runs periodically to make adjustment according to the large time-scale network dynamics. It takes the traffic rate, forwarding path, and policy chain of each flow, together with the available hardware resources, as input. It computes the proper placement of VNF instances and the particular VNF instances each flow is supposed to traverse. The algorithm of the Optimization Engine focuses on optimizing resource efficiency while preserving NF policies under traffic dynamics. The Optimization Engine interacts with the Resource Orchestrator to install VNF instances accordingly and obtains the information about available resources at APPLE hosts from the Resource Orchestrator. It also sends the information about how new flows are assigned to different VNF instances to the Rule Generator for the computation of data plane forwarding rules.

Resource Orchestrator. It allocates sufficient resources

and launches VNF instances according to the result of the Optimization Engine. In addition, it monitors the available resource on APPLE hosts and reports this information to the Optimization Engine.

Rule Generator. It gathers the outputs from different control plane applications, including the Optimization Engine, and generates the data plane forwarding rules. These rules are installed at physical and virtual switches through the SDN API.

Dynamic Handler. It may receive an overloading notification from a APPLE host. It will then re-balance the workload to resolve overloading by requesting the Rule Generator to install new forwarding rules.

Design challenges. There are several challenges involved in the design of APPLE: (1) VNFs consume power and contend for hardware resources with production VMs. Thus, it is highly motivated to find a resource-efficient way to place VNFs (e.g., minimizing the number of VNF instances), while enforcing policies. (2) Traffic is highly dynamic. We should avoid both under-provision during peak load and over-provision during base load to balance the performance and efficiency. However, the provision of VNFs happens in a much larger time-scale. As a result, elastic provision cannot be leveraged to resolve such small time-scale problem. We need to design a scheme to combine proactive VNF provision and dynamic load balancing to achieve the goal. (3) Proper forwarding rules are required to be installed at physical switches and vSwitches, such that all flows traverse the given order of NFs and other network behaviors are retained. The challenge here is how to efficiently use expensive TCAM memory while the semantics are preserved.

IV. Optimization Engine

The Optimization Engine may apply global optimization that computes a VNF placement plan for all current flows or online placement for any new flows. In this section we focus on global optimization for all flows in the network. This mechanism can be applied to ISP or data center networks whose traffic amount and pattern are predictable [16] [13] [43]. Online algorithms are for our future research.

A. Traffic aggregation for scalability

The Optimization Engine of APPLE needs to decide VNF placement for all flows whose number may be huge. Kandula et al. [23] found that 100K flows arrive every second on a 1500-server cluster. To resolve the scalability problem, APPLE aggregates traffic into *equivalence classes*. The flows having the same path and policy chain are aggregated into a class. Using classes as the granularity for the Optimization Engine yields several benefits: 1) The input size of the Optimization Engine can be reduced significantly, and hence the time to solve the optimization problem is shorter. 2) Classes can usually be expressed by wildcard rules. Using wildcard rules instead of exact matching rules will save the TCAM memory in the data plane; 3) Class-based aggregation can smooth the variation of traffic.

The fact that aggregated flows show smaller variance was found by existing work [30]. In addition, the smoothness of aggregated traffic can be derived from the power law form of mean-variance relationship (MVR) of traffic rate [21]. Due to space limit, we skip the derivation here.

In this paper, each class is denoted as $h \in H$, where H is the set of all classes. We use the recently developed atomic predicate based analysis [44] [42] to classify flows into equivalence classes. Detailed explanation can be found in [44] [42].

B. Distribution of VNF processing to different instances

The flows within a same class do not necessarily traverse the same sequence of VNF instances. With a centralized network-wide view, APPLE can spatially distribute the workload such that the responsibility of each VNF instance can be more balanced. Load distribution is also important to handle jumbo classes whose rates are beyond the capacity of any single VNF instance.

C. Algorithm Inputs

VNF capacity. We use Cap_n to denote the capacity of an instance of VNF n, whose metric is the number of packets per second. APPLE can measure the capacity of each VNF instance in advance, which is offline one-shot effort. Using ClickOS [28], when we observe that the packet loss rate soars rapidly, we consider this instance is overloaded.

Available Resource. Each VNF instance consumes hardware resources in its APPLE host. The Optimization Engine needs to ensure there are enough hardware resources to launch new instances. We denote the available hardware resources of the APPLE hosts connected to switch v as A_v. The Optimization Engine polls such information from the Resource Orchestrator. We use R_n to denote a resource requirement vector in which each element is the requirement of a type of resource of VNF n.

Policies. Policies are specified by network operators or applications. They describe the sequence of NFs that each class of flows need to traverse in order. $C_h = < c_h^j >$ represents the policy chain for class $h \in H$, where c_h^j means the the jth NF on the policy chain C_h.

Flow Paths. Forwarding paths are computed by control plane applications. We use $P_h = < p_h^i >$ to donate the path, where p_h^i is the ith switch class h encounters.

Traffic Rate. It can be estimated by other applications [18]. T_h captures the traffic rate of class $h \in H$.

D. Problem formulation and solving

Objectives. The high-level objective of the Optimization Engine is to minimize the total hardware and power resource usage. In this paper, we use the simplest abstraction of this objective: minimizing the number of VNF instances installed in the network. More complicated representation of this objective can also be supported by our system and is left for future study.

Policy enforcement requires the following for each flow:

Notations	Explanation
p_h^i	i-th switch on the path of class h
c_h^j	j-th VNF on the policy chain of class h
$d_{h,j}^i$	portion of class h processed in c_h^j connected to p_h^i
$\sigma_{h,j}^i$	cumulative portion of class h processed in c_h^j until p_h^i
T_h	traffic volume of class h
Cap_n	process capacity of VNF n
q_n^v	quantity of VNF n connected to switch v
R_n	the resource requirement vector of VNF n
A_v	available resource of APPLE host connect to switch v
$\|P(h)\|$	path length of class h
$\|C(h)\|$	number of VNFs at the policy chain of class h
$i(P,h,v)$	index of switch v on the path of class h
$i(C,h,n)$	index of VNF n at the policy chain of class h

TABLE II
NOTATIONS IN THE OPTIMIZATION PROBLEM.

1) For each NF specified by the policy for a flow, at least one instance is on the network path.
2) For any VNF instance n, there should be at lease one instance of the VNFs succeeding n in the policy chain connected to the same switch of n or a downstream switches of the path.

The network topology is represented by a graph $G=(V,E)$, where V is the set of switches in the network. Let N denote the set of all VNFs. The decision variable $d_{h,j}^i$ indicates the portion of traffic of class h to be processed in instances connected to switch p_h^i for VNF c_h^j. Another decision variable $q_v^n \in \{0,1,2,...\}$ quantifies the number of VNF instances needed for VNF n connected to switch v. We also introduce a new derived variable $\sigma_{h,j}^i$, which means the cumulative portion of traffic that has been processed, from beginning to p_h^i on the network path for VNF c_h^j for class h. For the ease of illustration, we also define a bunch of functions. $|P_h|$, $|C_h|$ are the length of P_h and C_h, respectively. $i(P,h,v)$ gets the index of switch v on the sequence $P_h = < p_h^i >$. Likewise, $i(C,h,n)$ is the index of VNF n on the sequence of $C_h = < c_h^j >$. The notations used in this optimization problem is listed in TABLE II. The optimization formulations are stated as follows.

$$Minimize \qquad \sum_{v\in V}\sum_{n\in N} q_n^v \tag{1}$$

s.t.

$$\sigma_{h,j}^i = \sigma_{h,j}^{i-1} + d_{h,j}^i \qquad \forall h,i,j \tag{2}$$

$$\sigma_{h,j-1}^i - \sigma_{h,j}^i \geq 0 \qquad \forall h,i,j \tag{3}$$

$$\sigma_{h,|C(h)|}^{|P(h)|} = 1 \qquad \forall h \tag{4}$$

$$\sum_{h:v\in P_h} T_h d_{h,i(C,h,n)}^{i(P,h,v)} \leq Cap_n \times q_n^v \qquad \forall v,n \tag{5}$$

$$\sum_{n\in N} R_n \times q_n^v \leq A_v \qquad \forall v \tag{6}$$

$$q_n^v \in \{0,1,2,....\} \qquad \forall n,v \tag{7}$$

$$0 \leq d_{h,j}^i \leq 1 \qquad \forall h,i,j \tag{8}$$

Eq. (3) makes sure that the policy chain order is preserved, as the second requirement in Sec. IV-D. Eq. (4) means that 100% traffic of class h need to be properly processed as indicated by the policy chain. Eq. (3), together with Eq. (4), enforces policies. Eq. (5) and Eq. (6) capture VNF capacity limits and resource constraints.

Even though $i(P,h,v)$ and $i(C,h,n)$ in Eq. (5) seem to make the optimization problem nonlinear, actually once the input is given, the values of $i(P,h,v)$ and $i(C,h,n)$ are known immediately, without solving the whole optimization problem. Therefore, this optimization problem is an Integer Linear Program (ILP). This optimization problem can be reduced from *Set Cover Problem*, which is known to be NP-hard. Furthermore, Dinur et al. [17] have proved that Set Cover Problem cannot be approximated within $\big(1-o(1)\big)\cdot \ln n$, unless $P=NP$, where n is the number of subsets. We apply LP relation, an approximation technique, to reduce the complexity and solve it by CPLEX [2] in our implementation. Our experiments using real network topologies and traffic traces show that the computation time is small enough (less than 3.1 sec for global optimization of a 79-switch network). Note that other practical orchestration frameworks [36] [34] use similar time to finish optimization and consider it fast enough for most existing networks. For gigantic networks including hundreds of switches, which are rare in practise, we plan to propose heuristic algorithms to solve it in future work.

V. Enforcing Optimization Results

Once getting the result from the Optimization Engine, APPLE needs to enforce the decisions. The installation of VNF instances are handled by the Resource Orchestrator as ordinary VM placement using existing technology such as OpenStack [11]. However, it is hard to infer the forwarding rules from the result of spatial distribution $d_{c,j}^i$ directly. Thus, we introduce a new concept, *sub-class*, to help APPLE generate forwarding rules.

A. Sub-class

Policy enforcement is on per-flow basis, even though the Optimization Engine operates on classes. It needs to be determined which specific VNF instances to traverse for each flow. Towards this goal, we define the aggregation of flows within a class that traverse the same VNF instances as a sub-class, denoted as s. d_c^s represents the portion of traffic rate of sub-class s in the overall traffic rate of class c. Clearly, $\sum_s d_c^s = 1$. For each class, we enumerate all the possible VNF instance sequences that enforce the policy chain and the assignment of the responsibility for each sub-class is accepted as long as the space distribution from the Optimization Engine is satisfied.

We propose two approaches to assign flows to different sub-classes. The first one leverages consistent hashing. For example, we have a class denoted by $< 10.1.1.0/24 >$, then $< 10.1.1.0/24, h \in [0,0.5) >$ represents a sub-class. If flows are uniformly hashed to $[0,1)$, this sub-class approximately includes 50% flows of this class. However, current hardware switches do not support programmable hash functions. Hence we use an alternative way to determine the sub-class of each flow in our current implementation. Following the previous example, the sub-class can be represented by $< 10.1.1.128/25 >$. The drawback of this method is that it may need multiple

rules to represent a single sub-class, and hence increasing TCAM memory cost. To reduce the TCAM consumption, we design a novel data plane scheme called flow tagging, which is presented in the following subsection.

B. Flow tagging

A tag is an identifer written to a packet header. We can customize the header modification in SDN-enabled switches. The unused bits in the packet header can be used as the tag field, such as the 6-bit DS field and 12-bit VLAN ID (if VLANs are not used).

The main idea of the tagging scheme is to avoid duplicated classifications on physical switches, which consume substantial TCAM resource. In the APPLE tagging scheme, each packet contains two tag fields. One field is for the *host ID*, which specifies the next host to process this packet. If one packet has traversed all the required VNF instances, this tagging field is *Fin*. The other field encodes *sub-class ID* within a class. Sub-class ID only has local meanings, thus it can be multiplexed by different classes. The Sub-class tagging field remains unchanged in the network. Fig. 2 shows the data plane framework when a packet arrives at a physical SDN switch. Upon reception of a new packet, the switch needs to check the host ID field. If host ID indicates one APPLE host connected to the switch, it would forward the packet to the APPLE host for NF processing; otherwise, the switch would forwarding the packet to the correct next hop. If this field is empty, it means that this packet just entered the network and the switch should tag a sub-class ID first. After that, if the packet is to be processed in any APPLE host connected to the switch, the switch should forward it to the APPLE host; otherwise, the packet should be tagged with the next host ID that processes it.

Such semantics can be easily encoded in TCAM, when flow table pipelining is supported. TABLE. III illustrates the TCAM layout at the physical switch. Rules of other applications are stored in the next table. Here, the sub-class matching rules are a bunch of wildcard rules to achieve the target distribution computed by the Optimization Engine, as the second method pointed in Sec. V-A. For switches not supporting pipeline processing, the semantics can still be retained by the cross-product of the two tables, but the TCAM consumption would increase. Note that the classification rules are just installed at the corresponding ingress switch for each sub-class to reduce TCAM consumption.

Forwarding rules are also needed in vSwitch embedded in APPLE hosts to direct packets to desired VNF instances. The matching rule is based on three tuples, $< \mathit{IncomePort}, \mathit{class}, \mathit{sub}-\mathit{class} >$, where class is specified by matching rules. A packet may traverse multiple VNF instances in one APPLE host, and *IncomePort* is enough to identify which VNF instances the packet has traversed. (We assume that a packet does not traverse a same instance twice.) We use both class and sub-class IDs to distinguish different sub-classes. Note that sub-class IDs are only adjusted at the ingress switch. When the packet leaves an APPLE host, it also needs be tagged to indicate the next APPLE host to process it. Since we can also install production VMs in APPLE hosts, the vSwitch adopts a similar tagging scheme and the processing pipeline to store rules of other applications. One difference is that *IncomePort* is enough to distinguish whether a packet has been tagged or not: the packets from the ports connect to production VMs are not tagged yet. Fig. 3 gives a concrete example describing how tagging scheme works.

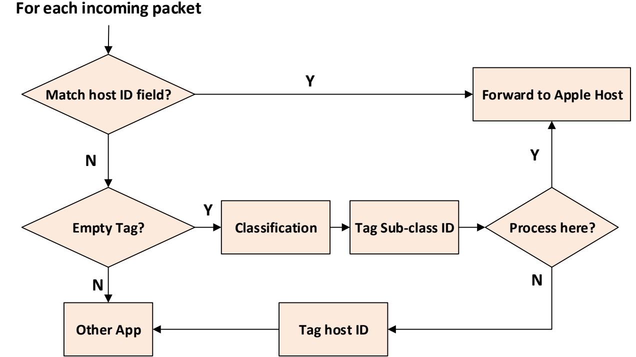

Fig. 2. Data plane framework at an SDN switch.

Type	Host ID field	Match	Action
Host match	Host ID	*	Fwd to APPLE host
Classification	Empty	Sub-classes	Tag sub-class ID, Fwd to APPLE host
	Empty	Sub-classes	Tag sub-class ID, Tag host ID, Go to next table
Pass by	*	*	Go to next table

TABLE III
LAYOUT OF TCAM AT PHYSICAL SWITCHES

VI. Incorporating traffic dynamics

There are two kinds of traffic dynamics which are different in granularity. The large time-scale traffic dynamic shows clear daily or weekly patterns [16]. More importantly, the traffic changes slowly, which can tolerate long VNF installation time. For this kind of traffic dynamics, it can be easily handled by periodically running the Optimization Engine and placing VNF instances accordingly. For planned traffic changes, such as VM migrations, we can also pre-install VNF instances to enforce policies.

The difficult part is to efficiently handle small time-scale traffic dynamics, because the traffic changing rate is both fast and vigorous. In this paper, we propose another mechanism to adapt to the traffic dynamics quickly, called *fast failover*. The core idea is to temporally re-balance the distribution of sub-classes to relieve the overloaded VNF instance. Since overloading is transient, the distribution will roll back to the normal state when the VNF instance is no longer overloaded.

Fast failover can react quickly to small time-scale traffic dynamics, because it only temporarily changes the TCAM

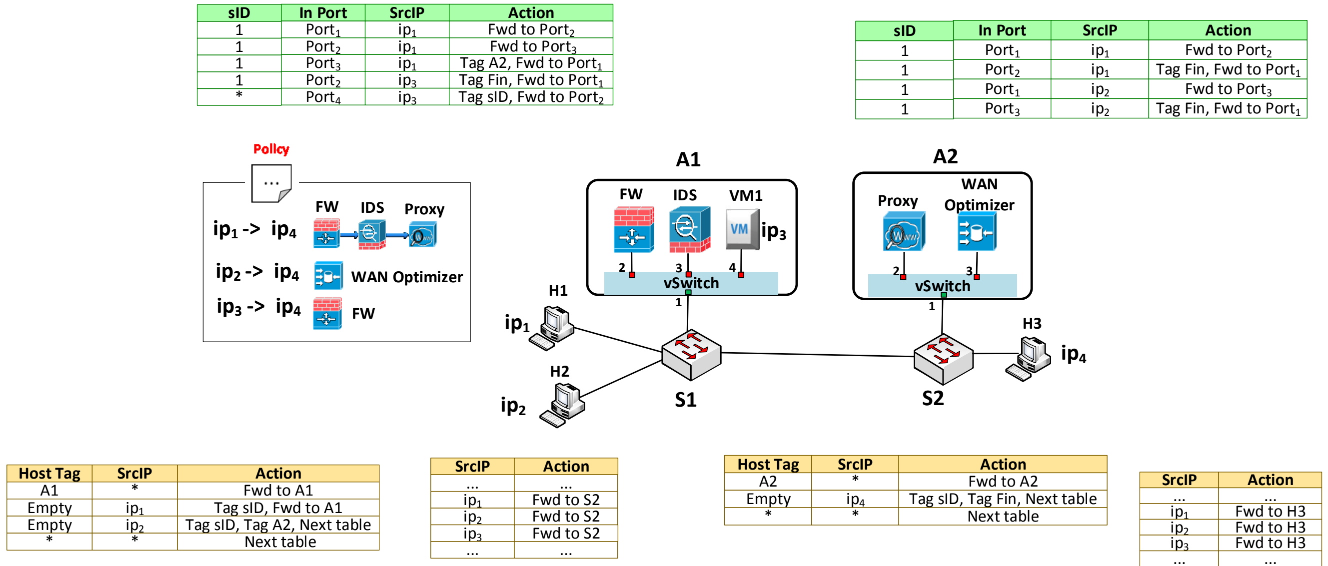

Fig. 3. Illustration of three common scenarios for tagging scheme. There are 3 classes, and they have the same path, $S1 \rightarrow S2$. The classes can be distinguished by the srcIP field. Each class only has only one sub-class (denoted as sID in the figure). The traffic $ip_1 \rightarrow ip_4$ represents the scenario where the packets traverse multiple APPLE hosts. The traffic $ip_2 \rightarrow ip_4$ represents the scenario where the packets are processed in APPLE hosts not connected to the ingress switch. The traffic $ip_3 \rightarrow ip_4$ represents the scenario where the packets originate within an APPLE host.

matching rules and installs light-weight ClickOS instances, both of which happen in tens of milliseconds. When a VNF instance is overloaded, it will send an overloading notification to the Dynamic Handler. The Dynamic Handler in turn will set the workload of all sub-classes that traverse this VNF instance to half as much as before overloading and spread the other half to the least loaded sub-classes within a same class. If such re-balance is expected to result in overloading of another VNF instance, the Dynamic Handler installs new ClickOS instances to create new sub-classes to absorb traffic dynamics. Also, when a VNF instance is no longer overloaded, the newly installed ClickOS instances are cancelled to save hardware resources. Fig. 4 illustrates the steps to achieve fast failover for a particular example, where the firewall is ClickOS-based. In this example, a new sub-class is created and a new ClickOS VM is initiated. Initially, there are two IDS and one firewall instances on the path. When the *master IDS* instance is overloaded, APPLE builds a new sub-class by installing a new firewall to accommodate traffic from previous sub-class.

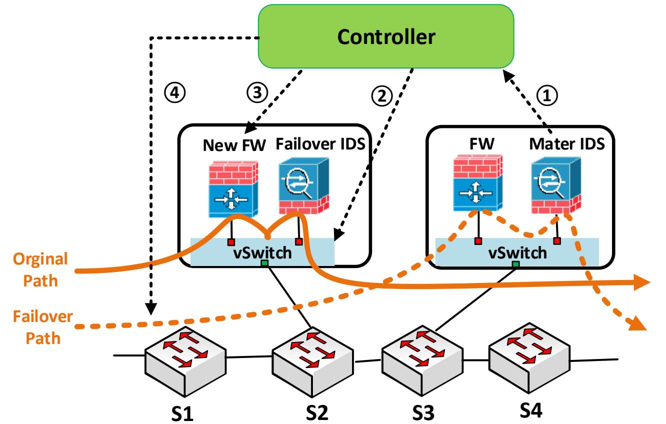

Fig. 4. Steps of fast failover for a sub-class whose policy chain is $FW \rightarrow IDS$: (1) Overloaded VNF instance sends an overloading notification. (2) New ClickOS instances are initiated. (3) Controller installs forwarding rules for the new sub-class. (4) Update rules to forward half traffic to the new sub-class.

VII. Prototype Implementation

A. ClickOS VM Initiation

We have successfully implemented a prototype system of APPLE using recently developed open-source software tools including OpenStack [11], ClickOS [28], Open vSwitch [9], and OpenDayLight SDN controller [10]. Fig. 5 shows the step-by-step procedures to initiate a new ClickOS [28] VM for a VNF instance. In our system, the central controller is a stand-alone application that calls services provided by other orchestration softwares, Openstack [11] and Opendaylight controller [10], via their REST APIs to manage the system. It needs to be specially pointed out why Openstack delegates the networking part to Opendaylight, rather than managing the networking itself. An Openstack controller contains a component called Neutron, which is responsible for the creation, configuration and management of the embedded virtual network. However, Neutron exposes no APIs for the users to install customized forwarding rules to Open vSwitches [9] used in APPLE hosts. On the other hand, if we manually set

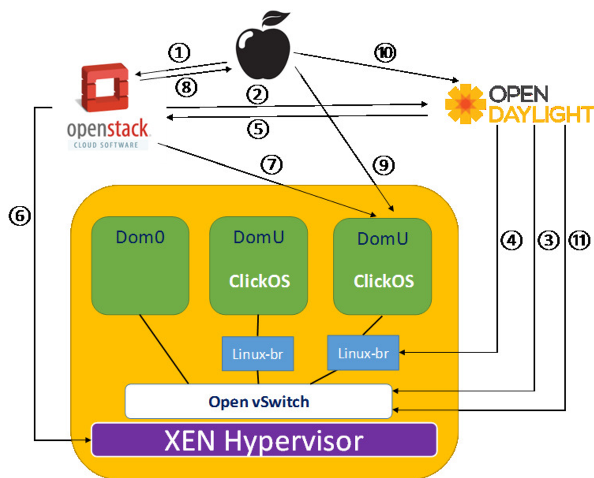

Fig. 5. Implementation of initiating a new ClickOS VM for a VNF instance.

Opendaylight as the controller for Open vSwitches, Openstack will seize control of Open vSwitches intermediately, in order to ensure the connectivity of the VMs. The solution is to explicitly configure that Opendaylight handles the networking for Openstack. When there is any new VM initiation request, Openstack notifies Opendaylight to prepare the networking via REST API (**Step 2**). Opendaylight is more than an OpenFlow controller. It is a platform integrating both customized North-bound and South-bound APIs for control applications and network devices respectively. In **Step 3**, Opendaylight calls OVSDB [33] South-bound RPC to create a new port on the Open vSwitch. Since Xen VMs do not support Open vSwitch directly, we add a Linux Bridge [6] between one Xen [14] VM and the Open vSwitch (**Step 4**). Augmented with the networking information, especially that used for configuring virtual NIC, from Opendaylight (**Step 5**), Openstack leverages libvirt driver [5] to create a new VM (**Step 6**). After that, the newly created VM fetches the ClickOS image from Openstack and installs it (**Step 7**). Once APPLE is notified the completion of the VM creation (**Step 8**), it configures the ClickOS VM into the desired VNF through a customized tool describe in [28] (**Step 9**). Finally, APPLE proactively installs the forwarding rules in the Open vSwitch by calling Opendaylight's REST API (**Step 10&11**).

For normal VMs other than ClickOS, the procedures to initiate them are almost the same. The only different is that in **Step 9**, generic configuration tools are utilized (e.g. the tools from Openstack).

B. Overloading Detection

Different VNFs have their own metrics to define overloading. In this paper, we do not use general but expensive load monitoring tools (e.g. Intel Performance Counter Monitor API [3]). We find that for most of the VNFs, the performance is closely related to the packet receiving rate, but not the packet size, as illustrated by Fig. 6 which shows how the loss rate changes for a ClickOS VM that is configured as a passive monitor. Based on measurement results, we set a proper threshold to define overloading. Moreover, it is very convenient to get the packet receiving rate by periodically polling the packet counters of Open vSwitches. In the current implementation, we poll the per-port packet counters instead of the per-flow packet counters, because from our experiment we find that the per-port counters update almost instantly while the per-flow counters update approximately every 1 second.

VIII. Prototype Evaluation

A. Experiment Setup

We install the all-in-one Openstack (Liberty release), Opendaylight (Lithium release), Xen Hypervisor (version 4.4.2) and Open vSwitch (version 2.0.2) on a same VirtualBox VM with quad cores@3.4G and 8GB memory. Two network namespaces [7] in Dom0 and all Xen VMs are connected to a same Open vSwitch. Here, network namespaces, light-weight containers, are created to emulate production hosts or VMs. One network namespace sends packets to the other one via a ClickOS VM that is configured as a passive monitor.

B. ClickOS VM Setup Time

Even though the ClickOS VM can be booted on a Xen Hypervisor in 30 ms as stated in [28], our prototype experiment indicates the booting time is much longer if Openstack is involved. Since the setup time is hard to record directly, we approximate it by measuring the duration which the throughput drops to zero when we emulate failover: new forwarding rules are installed on the Open vSwitch (which consumes only negligible time, as little as 70ms) right before ClickOS VM creation, meanwhile the namespaces are sending UDP packets (Fig. 7). We conduct this experiment 10 times. The approximate booting time ranges from 3.9 seconds to 4.6 seconds, with an average of 4.2 seconds. The main reason for the longer booting time is that Openstack and Opendaylight consume substantial time to orchestrate and prepare the networking before actually initiating a new VM (**Step 1 - Step 5**).

C. Waiting For Five Seconds

One solution to obviate the overhead introduced by failover is to change the forwarding rules after the complete creation of the ClickOS VM. In this subsection, we modify the forwarding rules on the Open vSwitch 5 seconds after we send the VM initiation request to Openstack via REST API. According to our previous VM setup time measurement, 5 seconds is enough to completely boot a new ClickOS VM.

In this subsection, we measure the overhead of failover for TCP and UDP flows, by using Iperf [4] and Netcat [8] to send UDP packets and to transfer a 20MB file via TCP, respectively. We conduct both experiments 10 times. As expected, there is no overhead associated with failover. For all 10 times of the UDP experiment in which we send 1500-Byte UDP packets at 10Kpps, the loss rate for the UDP flow is always 0%. Fig. 8 shows the CDF plot of the time to transmit a 20MB file with and without failover, which indicates that failover does not bring extra overhead. The performances of the three situations

in Fig. 8 are approximately the same and their differences are due to the statistical fluctuation.

D. Reconfiguring Existing VMs

Even though the solution in Sec. VIII-C introduces no performance degradation, the 5-second waiting time constrains the flexibility of the system to adapt to network dynamics. To this end, we propose to reconfigure existing ClickOS VMs to save the booting time. The micro-measurements shows that the time to install forwarding rules is 70ms and reconfiguration only takes 30ms. We conduct a similar experiment to Sec. VIII-C. The only difference is that we just reconfigure an existing ClickOS VM rather than initiating a new one. Still, the UDP packet flow rate is 0% for each time of the UDP experiment. There is also no noticeable difference in TCP performance (Fig. 8).

E. Overloading Detection

One namaspace use pktgen [31] to send 1500-Byte UDP packets to another one via a ClickOS instance that is configured as a passive monitor. The passive monitor is viewed as being overloaded if the receiving packets rate is greater than 8.5 Kpps. The distribution will roll back to the normal state if the packets rate drops to 4 Kpps or lower. Fig. 9 illustrates how fast our system detects overloading. Initially, the source sending rate is 1 Kpps. To mimic network dynamics, the source sending rate soars to 10 Kpps. The overloading is immediately detected. Therefore, another ClickOS instance is quickly configured as a passive monitor and the traffic is evenly split to the two passive monitors. After 5 seconds, the source sending rate becomes 1 Kpps again, which causes the network rolls back to the normal state. During the whole process, the packet loss is 0%.

IX. Simulation Evaluation

We perform extensive simulations using real traffic trace data and real network topologies.

A. Methodology

Topology and data set. We use three representative topologies for campus network, enterprise network, and data center network representatively. For the campus network, we use internet2 research network (12 nodes and 15 links). Time-varying traffic matrices for internet2 are provided in [1], which consist of snapshots of 12×12 traffic matrices. We adopt the totem data set [41] to represent enterprise network. It contains an intradomain network, GEANT (23 nodes and 74 links) and associated time-varying traffic matrices. We use a 2-tier campus data center network, UNIV1 (23 nodes and 43 links) [16]. In this data set, due to the lack of traffic matrices, we replay the corresponding trace between random source-destination pairs. To illustrate that the Optimization Engine is scalable even for large topologies, we also use a Rocketfuel router-level ISP topology, AS-3679 [40]. The traffic matrices for AS-3679 are synthesized using FNSS tools [35].

Using Internet2 and GEANT datasets, we combine 672 snapshots of traffic matrices for each topology. We run the Optimization Engine, whose traffic matrix input is the mean value of the 672 snapshots. After that, we place VNFs in the network according to the result from the Optimization Engine. At last, we replay all the traffic matrices in time order and APPLE will react to traffic changes during this process. Likewise, for UNIV1, we generate the time-varying traffic matrices, from the real trace. Each snapshot lasts for one second. The rest steps are the same with the other two topologies. For each topology, we conduct experiments multiple times with different traffic matrices.

Network Function	Core Required	Capacity	ClickOS
Firewall	4	900Mbps	✓
Proxy	4	900Mbps	x
NAT	2	900Mbps	✓
IDS	8	600Mbps	x

TABLE IV
VNF DATA SHEETS

Topology	Nodes	Links	Time
Internet2	12	15	0.029 second
GEANT	23	74	0.1 second
UNIV1	23	43	0.235 second
AS-3679	79	147	3.013 seconds

TABLE V
AVERAGE COMPUTATION TIME OF DIFFERENT TOPOLOGIES.

Policy chains. Due to the lack of publicly available information on NF related policies, we synthesize network function policies based on real-network study by [37] and case studies [12]. The policy chains are the sequences of 4 different NFs: firewall, proxy, NAT and IDS.

VNF specifications. The information on capacity and resource requirements for each NF is from the survey in [15], which is listed in TABLE .IV. We also assume that the firewall and NAT are implemented in ClickOS, while the proxy and IDS are contained in normal VMs. We assume that there are 64 cores at each APPLE host.

Metrics. We measure the following metrics: TCAM and hardware consumption, the algorithm computation time, and packet loss ratio during traffic dynamics.

B. Computation Time

Short computing time is crucial to timely VNF provision. We solve the optimization problem discussed in Sec. IV-D by CPLEX [2] on a quadcore@3.40G desktop with 16GB memory. TABLE. V compares the average time to solve the problem for different topologies. As we can see, for small and medium topologies (Internet2, GEANT and UNIV1), the Optimization Engine is fast: the computation time is less than 1 second. Even for the large topology (e.g. AS-3967), the computation time is acceptable.

C. TCAM Usage

By leverage the tagging scheme, the TCAM consumption is reduced. Fig. 10 gives the boxplot of the TCAM usage reduction ratio compared to that without tagging scheme, for three topologies under different traffic matrices . As we can see, there is a least 4X reduction for all three topologies. The reduction ratio for UNIV1 topology is more impressive

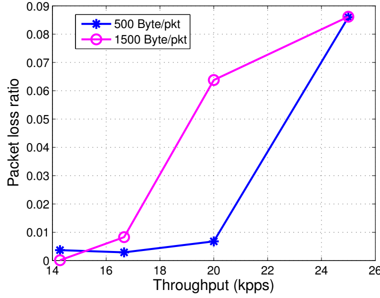

Fig. 6. Performance of VNF

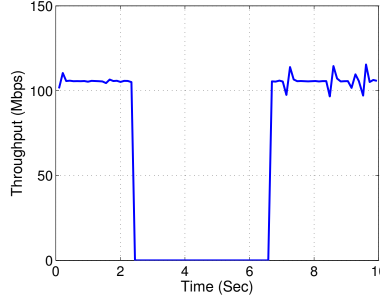

Fig. 7. ClickOS booting time

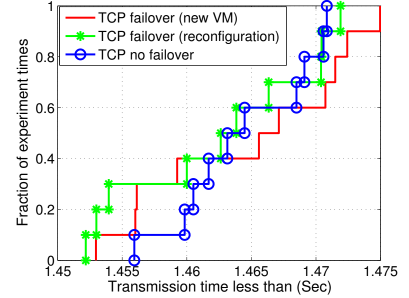

Fig. 8. Distribution of file TX time

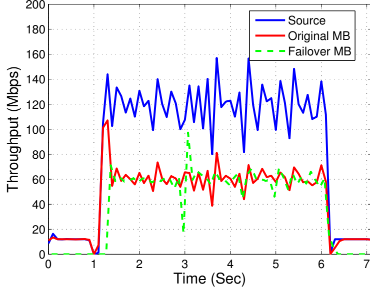

Fig. 9. Illustration of fast failover

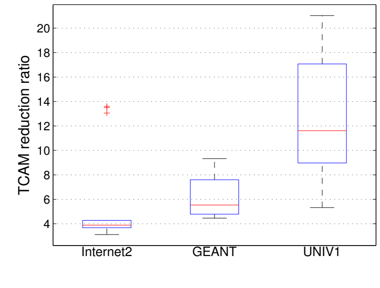

Fig. 10. TCAM usage reduction by tagging

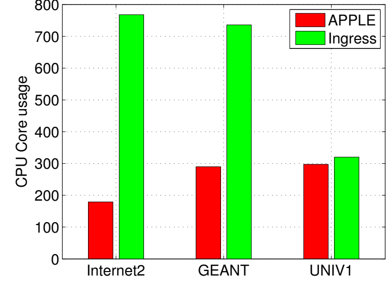

Fig. 11. Avg. CPU Core usage

than the other two, because traffic exploits multi-paths in data center networks. Therefore, we are more motivated to tag these traffic at their ingress switch, rather than match them on all multi-paths, resulting in less TCAM consumption.

D. Hardware Resource Usage

Since APPLE is the first VM-based orchestration framework that introduces no interference to the network, we compare the hardware resource usage for APPLE with an alternative strawman solution called *ingress*, which consolidates all the VNFs of the policy chain in the ingress switch and enforce policy there for each class. Fig. 11 plots the hardware usage for the two solutions. There is 4X reduction for internet2 and 2.5X reduction for GEANT. This benefit comes from the resource multiplexing between different classes. The gap in UNIV1 is not that significant, because UNIV1 only has two core switches. Therefore, the limited hardware capacity at the core switches force APPLE to place VNFs at the ingress switches.

E. React to traffic changes

With fast failover, APPLE can quickly react to traffic changes with low packet loss rate. Fig. 12 depicts the packet loss rate over time for three different topologies. In this plot, we compare APPLE with fast failover to APPLE without fast failover. Thanks to fast failover, the packet loss rate remains much lower for all three topologies even in the face of fiercely changed traffic. This plot illustrates the ability of fast failover to absorb traffic burst efficiently. In the mean time, only a few new ClickOS instances are installed to support fast failover. The average additional cores to support fast failover is less than 17 for all topologies.

X. Discussions

Some NFs may change the packet headers, which makes sub-class classification invalid. If such NFs are in the network, we can tag the global sub-class identification in the affected packets and the forwarding rules would match on such field to address this problem. Due to the page limit, we skip the detailed explanation of the above discussions.

When VNF instances processing packet, they consume multiple hardware resources (e.g. CPU cycles, NIC bandwidth). However, current VM hypervisor's resource scheduler only considers how to statically fairly share CPU and memory [22]. To integrate a max-min fair multi-resource scheduler [25] for policy enforcement would be our future work.

XI. Conclusion

APPLE is the first implementation of an NFV orchestration framework based on VMs that satisfies three requirements, namely policy enforcement, interference freedom, and resource isolation. APPLE introduces additional functional components to current SDN controller and middleware between the control plane and data plane. APPLE applies an optimization engine to determine VNF placement and a flow tagging scheme to reduce TCAM consumption. We present detailed prototype implementation of all APPLE components. Results from both implementation and simulations using real network topologies and traffic matrices show that APPLE satisfies desired properties of VNF deployment.

XII. ACKNOWLEDGEMENT

The authors are supported by University of Kentucky College of Engineering Faculty Startup Grant and National Science Foundation grant CNS-1464335. The authors also thank

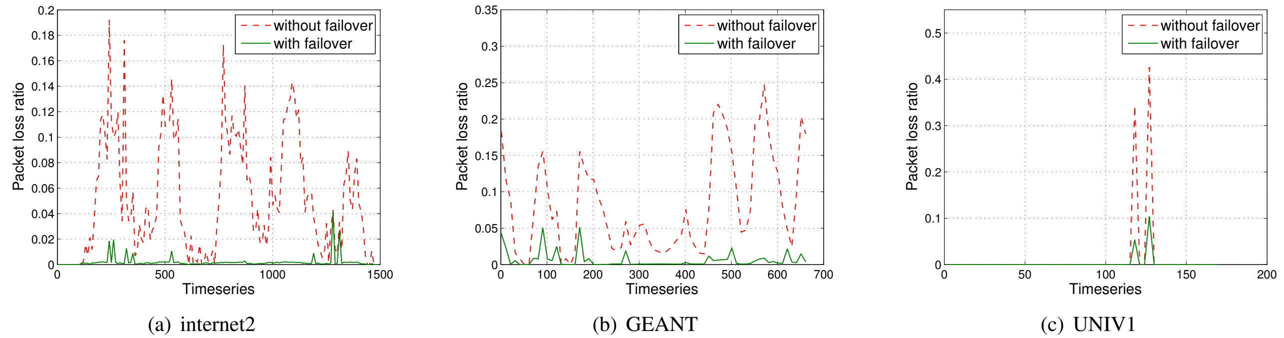

Fig. 12. Packet loss rate over time for APPLE with and without fast failover.

anonymous ICDCS reviews for their constructive comments and suggestions.

References

[1] The abilene observatory data collections. http://www.cs.utexas.edu/~yzhang/research/AbileneTM/.
[2] Ibm ilog cplex optimizer. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.
[3] Intel perf. counter mon. https://goo.gl/MCIhEE.
[4] Iperf. https://iperf.fr/.
[5] Libvirt virtualization api. http://libvirt.org/.
[6] Linux bridge. http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.
[7] Linux network namespace. http://man7.org/linux/man-pages/man8/ip-netns.8.html.
[8] Netcat: the tcp/ip swiss army. http://nc110.sourceforge.net/.
[9] Open vswitch. http://openvswitch.org/.
[10] Opendaylight. https://www.opendaylight.org/.
[11] Openstack. http://www.openstack.org/.
[12] Service function chaining use cases in data centers. http://datatracker.ietf.org/doc/draft-ietf-sfc-dc-use-cases/.
[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. of SIGCOMM*, 2011.
[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.
[15] M. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. On orchestrating virtualized network functions. In *Proc. of IEEE/ACM/IFIP CNSM*, 2015.
[16] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of ACM IMC*, 2010.
[17] I. Dinur and D. Steurer. Analytical approach to parallel repetition. In *Proc. of ACM STOC*, 2014.
[18] A. Feldmann et al. Deriving traffic demands for operational ip networks: Methodology and experience. In *Proc. of ACM SIGCOMM*, 2000.
[19] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.
[20] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.
[21] A. Gunnar et al. Traffic matrix estimation on a large ip backbone: a comparison on real data. In *Proc. of the ACM IMC*, 2004.
[22] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*. 2006.
[23] S. Kandula et al. The nature of data center traffic: measurements & analysis. In *Proc. of the ACM IMC*, 2009.
[24] L. E. Li et al. Pace: policy-aware application cloud embedding. In *Proc. of IEEE INFOCOM*, 2013.
[25] X. Li and C. Qian. Low-complexity multi-resource packet scheduling for network functions virtualization. In *Proc. of IEEE INFOCOM*, 2015.
[26] X. Li and C. Qian. Traffic and failure aware vm placement for multi-tenant cloud computing. In *Proc. of IEEE/ACM IWQoS*, 2015.
[27] X. Li and C. Qian. A survey of network function placement. In *Proc. of IEEE CCNC*, 2016.
[28] J. Martins et al. Clickos and the art of network function virtualization. In *Proc. of USENIX NSDI*, 2014.
[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.
[30] R. Morris and D. Lin. Variance of aggregated web traffic. In *Proc. of IEEE INFOCOM*, 2000.
[31] R. Olsson. Pktgen the linux packet generator. In *Proc. of the Linux Symposium*, 2005.
[32] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for nfv applications. In *Proc. of ACM SOSP*, 2015.
[33] B. Pfaff and E. B. Davie. The open vswitch database management protocol. Technical report, RFC 7047, December, 2013.
[34] Z. A. Qazi and et al. Simple-fying middlebox policy enforcement using sdn. In *Proc. of ACM SIGCOMM*, 2013.
[35] L. Saino, C. Cocora, and G. Pavlou. A toolchain for simplifying network simulation setup. In *Proc.of SIMUTOOLS*, 2013.
[36] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. of USENIX NSDI*, 2012.
[37] V. Sekar et al. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. of ACM HotNets*, 2011.
[38] J. Sherry et al. Rollback-Recovery for Middleboxes. In *Proc. of ACM SIGCOMM*, 2015.
[39] J. Sherry and S. Ratnasamy. A Survey of Enterprise Middlebox Deployments. Technical report, EECS, UC Berkeley, 2012.
[40] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. 2002.
[41] S. Uhlig et al. Providing public intradomain traffic matrices to the research community. *ACM SIGCOMM CCR*, 36(1), 2006.
[42] H. Wang, C. Qian, Y. Yu, H. Yang, and S. Lam. Practical network-wide packet behavior identification by ap classifier. In *Proc. of ACM CoNext*, 2015.
[43] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proc. of ACM SIGCOMM*, 2012.
[44] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *Proc. of IEEE ICNP*, 2013.
[45] Y. Zhang et al. StEERING: A Software-Defined Networking for Inline Service Chaining. In *Proc. of IEEE ICNP*, 2013.